

---

# **GNU Scientific Library**

*Release 2.7*

**Mark Galassi  
Jim Davies  
James Theiler  
Brian Gough  
Gerard Jungman  
Patrick Alken  
Michael Booth  
Fabrice Rossi  
Rhys Ulerich**

Jun 18, 2021



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Routines available in GSL	1
1.2	GSL is Free Software	1
1.3	Obtaining GSL	2
1.4	No Warranty	2
1.5	Reporting Bugs	3
1.6	Further Information	3
1.7	Conventions used in this manual	3
<b>2</b>	<b>Using the Library</b>	<b>5</b>
2.1	An Example Program	5
2.2	Compiling and Linking	5
2.3	Shared Libraries	7
2.4	ANSI C Compliance	7
2.5	Inline functions	8
2.6	Long double	8
2.7	Portability functions	9
2.8	Alternative optimized functions	9
2.9	Support for different numeric types	10
2.10	Compatibility with C++	11
2.11	Aliasing of arrays	11
2.12	Thread-safety	11
2.13	Deprecated Functions	11
2.14	Code Reuse	12
<b>3</b>	<b>Error Handling</b>	<b>13</b>
3.1	Error Reporting	13
3.2	Error Codes	14
3.3	Error Handlers	14
3.4	Using GSL error reporting in your own functions	15
3.5	Examples	16
<b>4</b>	<b>Mathematical Functions</b>	<b>17</b>
4.1	Mathematical Constants	17
4.2	Infinities and Not-a-number	18
4.3	Elementary Functions	18
4.4	Small integer powers	19
4.5	Testing the Sign of Numbers	20
4.6	Testing for Odd and Even Numbers	20
4.7	Maximum and Minimum functions	20

4.8	Approximate Comparison of Floating Point Numbers . . . . .	21
<b>5</b>	<b>Complex Numbers</b>	<b>23</b>
5.1	Representation of complex numbers . . . . .	23
5.2	Complex number macros . . . . .	24
5.3	Assigning complex numbers . . . . .	24
5.4	Properties of complex numbers . . . . .	25
5.5	Complex arithmetic operators . . . . .	25
5.6	Elementary Complex Functions . . . . .	26
5.7	Complex Trigonometric Functions . . . . .	26
5.8	Inverse Complex Trigonometric Functions . . . . .	27
5.9	Complex Hyperbolic Functions . . . . .	27
5.10	Inverse Complex Hyperbolic Functions . . . . .	28
5.11	References and Further Reading . . . . .	28
<b>6</b>	<b>Polynomials</b>	<b>29</b>
6.1	Polynomial Evaluation . . . . .	29
6.2	Divided Difference Representation of Polynomials . . . . .	29
6.3	Quadratic Equations . . . . .	30
6.4	Cubic Equations . . . . .	31
6.5	General Polynomial Equations . . . . .	31
6.6	Examples . . . . .	32
6.7	References and Further Reading . . . . .	33
<b>7</b>	<b>Special Functions</b>	<b>35</b>
7.1	Usage . . . . .	35
7.2	The <code>gsl_sf_result</code> struct . . . . .	36
7.3	Modes . . . . .	36
7.4	Airy Functions and Derivatives . . . . .	37
7.5	Bessel Functions . . . . .	38
7.6	Clausen Functions . . . . .	45
7.7	Coulomb Functions . . . . .	45
7.8	Coupling Coefficients . . . . .	47
7.9	Dawson Function . . . . .	48
7.10	Debye Functions . . . . .	48
7.11	Dilogarithm . . . . .	49
7.12	Elementary Operations . . . . .	49
7.13	Elliptic Integrals . . . . .	50
7.14	Elliptic Functions (Jacobi) . . . . .	52
7.15	Error Functions . . . . .	52
7.16	Exponential Functions . . . . .	53
7.17	Exponential Integrals . . . . .	55
7.18	Fermi-Dirac Function . . . . .	57
7.19	Gamma and Beta Functions . . . . .	58
7.20	Gegenbauer Functions . . . . .	61
7.21	Hermite Polynomials and Functions . . . . .	62
7.22	Hypergeometric Functions . . . . .	65
7.23	Laguerre Functions . . . . .	67
7.24	Lambert W Functions . . . . .	67
7.25	Legendre Functions and Spherical Harmonics . . . . .	68
7.26	Logarithm and Related Functions . . . . .	73
7.27	Mathieu Functions . . . . .	74
7.28	Power Function . . . . .	75
7.29	Psi (Digamma) Function . . . . .	76

7.30	Synchrotron Functions . . . . .	77
7.31	Transport Functions . . . . .	77
7.32	Trigonometric Functions . . . . .	78
7.33	Zeta Functions . . . . .	79
7.34	Examples . . . . .	81
7.35	References and Further Reading . . . . .	82
<b>8</b>	<b>Vectors and Matrices</b>	<b>83</b>
8.1	Data types . . . . .	83
8.2	Blocks . . . . .	84
8.3	Vectors . . . . .	85
8.4	Matrices . . . . .	94
<b>9</b>	<b>Permutations</b>	<b>107</b>
9.1	The Permutation struct . . . . .	107
9.2	Permutation allocation . . . . .	107
9.3	Accessing permutation elements . . . . .	108
9.4	Permutation properties . . . . .	108
9.5	Permutation functions . . . . .	108
9.6	Applying Permutations . . . . .	109
9.7	Reading and writing permutations . . . . .	109
9.8	Permutations in cyclic form . . . . .	110
9.9	Examples . . . . .	110
9.10	References and Further Reading . . . . .	112
<b>10</b>	<b>Combinations</b>	<b>113</b>
10.1	The Combination struct . . . . .	113
10.2	Combination allocation . . . . .	113
10.3	Accessing combination elements . . . . .	114
10.4	Combination properties . . . . .	114
10.5	Combination functions . . . . .	114
10.6	Reading and writing combinations . . . . .	115
10.7	Examples . . . . .	115
10.8	References and Further Reading . . . . .	116
<b>11</b>	<b>Multisets</b>	<b>117</b>
11.1	The Multiset struct . . . . .	117
11.2	Multiset allocation . . . . .	117
11.3	Accessing multiset elements . . . . .	118
11.4	Multiset properties . . . . .	118
11.5	Multiset functions . . . . .	118
11.6	Reading and writing multisets . . . . .	119
11.7	Examples . . . . .	119
<b>12</b>	<b>Sorting</b>	<b>123</b>
12.1	Sorting objects . . . . .	123
12.2	Sorting vectors . . . . .	124
12.3	Selecting the k smallest or largest elements . . . . .	125
12.4	Computing the rank . . . . .	125
12.5	Examples . . . . .	126
12.6	References and Further Reading . . . . .	127
<b>13</b>	<b>BLAS Support</b>	<b>129</b>
13.1	GSL BLAS Interface . . . . .	130
13.2	Examples . . . . .	138

13.3	References and Further Reading	139
<b>14</b>	<b>Linear Algebra</b>	<b>141</b>
14.1	LU Decomposition	141
14.2	QR Decomposition	143
14.3	QR Decomposition with Column Pivoting	149
14.4	LQ Decomposition	150
14.5	QL Decomposition	151
14.6	Complete Orthogonal Decomposition	152
14.7	Singular Value Decomposition	153
14.8	Cholesky Decomposition	154
14.9	Pivoted Cholesky Decomposition	156
14.10	Modified Cholesky Decomposition	157
14.11	LDLT Decomposition	158
14.12	Tridiagonal Decomposition of Real Symmetric Matrices	158
14.13	Tridiagonal Decomposition of Hermitian Matrices	159
14.14	Hessenberg Decomposition of Real Matrices	159
14.15	Hessenberg-Triangular Decomposition of Real Matrices	160
14.16	Bidiagonalization	160
14.17	Givens Rotations	161
14.18	Householder Transformations	161
14.19	Householder solver for linear systems	162
14.20	Tridiagonal Systems	162
14.21	Triangular Systems	163
14.22	Banded Systems	164
14.23	Balancing	168
14.24	Examples	168
14.25	References and Further Reading	169
<b>15</b>	<b>Eigensystems</b>	<b>171</b>
15.1	Real Symmetric Matrices	171
15.2	Complex Hermitian Matrices	172
15.3	Real Nonsymmetric Matrices	172
15.4	Real Generalized Symmetric-Definite Eigensystems	174
15.5	Complex Generalized Hermitian-Definite Eigensystems	175
15.6	Real Generalized Nonsymmetric Eigensystems	176
15.7	Sorting Eigenvalues and Eigenvectors	177
15.8	Examples	178
15.9	References and Further Reading	182
<b>16</b>	<b>Fast Fourier Transforms (FFTs)</b>	<b>183</b>
16.1	Mathematical Definitions	183
16.2	Overview of complex data FFTs	184
16.3	Radix-2 FFT routines for complex data	185
16.4	Mixed-radix FFT routines for complex data	188
16.5	Overview of real data FFTs	191
16.6	Radix-2 FFT routines for real data	192
16.7	Mixed-radix FFT routines for real data	193
16.8	References and Further Reading	198
<b>17</b>	<b>Numerical Integration</b>	<b>199</b>
17.1	Introduction	199
17.2	QNG non-adaptive Gauss-Kronrod integration	201
17.3	QAG adaptive integration	201
17.4	QAGS adaptive integration with singularities	202

17.5	QAGP adaptive integration with known singular points . . . . .	202
17.6	QAGI adaptive integration on infinite intervals . . . . .	202
17.7	QAWC adaptive integration for Cauchy principal values . . . . .	203
17.8	QAWS adaptive integration for singular functions . . . . .	203
17.9	QAWO adaptive integration for oscillatory functions . . . . .	204
17.10	QAWF adaptive integration for Fourier integrals . . . . .	205
17.11	CQUAD doubly-adaptive integration . . . . .	206
17.12	Romberg integration . . . . .	207
17.13	Gauss-Legendre integration . . . . .	207
17.14	Fixed point quadratures . . . . .	208
17.15	Error codes . . . . .	210
17.16	Examples . . . . .	210
17.17	References and Further Reading . . . . .	213
<b>18</b>	<b>Random Number Generation</b>	<b>215</b>
18.1	General comments on random numbers . . . . .	215
18.2	The Random Number Generator Interface . . . . .	215
18.3	Random number generator initialization . . . . .	216
18.4	Sampling from a random number generator . . . . .	216
18.5	Auxiliary random number generator functions . . . . .	217
18.6	Random number environment variables . . . . .	218
18.7	Copying random number generator state . . . . .	219
18.8	Reading and writing random number generator state . . . . .	220
18.9	Random number generator algorithms . . . . .	220
18.10	Unix random number generators . . . . .	223
18.11	Other random number generators . . . . .	224
18.12	Performance . . . . .	226
18.13	Examples . . . . .	227
18.14	References and Further Reading . . . . .	228
18.15	Acknowledgements . . . . .	229
<b>19</b>	<b>Quasi-Random Sequences</b>	<b>231</b>
19.1	Quasi-random number generator initialization . . . . .	231
19.2	Sampling from a quasi-random number generator . . . . .	231
19.3	Auxiliary quasi-random number generator functions . . . . .	232
19.4	Saving and restoring quasi-random number generator state . . . . .	232
19.5	Quasi-random number generator algorithms . . . . .	232
19.6	Examples . . . . .	233
19.7	References . . . . .	233
<b>20</b>	<b>Random Number Distributions</b>	<b>235</b>
20.1	Introduction . . . . .	235
20.2	The Gaussian Distribution . . . . .	237
20.3	The Gaussian Tail Distribution . . . . .	239
20.4	The Bivariate Gaussian Distribution . . . . .	241
20.5	The Multivariate Gaussian Distribution . . . . .	242
20.6	The Exponential Distribution . . . . .	243
20.7	The Laplace Distribution . . . . .	244
20.8	The Exponential Power Distribution . . . . .	245
20.9	The Cauchy Distribution . . . . .	246
20.10	The Rayleigh Distribution . . . . .	247
20.11	The Rayleigh Tail Distribution . . . . .	248
20.12	The Landau Distribution . . . . .	249
20.13	The Levy alpha-Stable Distributions . . . . .	250

20.14	The Levy skew alpha-Stable Distribution . . . . .	251
20.15	The Gamma Distribution . . . . .	252
20.16	The Flat (Uniform) Distribution . . . . .	254
20.17	The Lognormal Distribution . . . . .	255
20.18	The Chi-squared Distribution . . . . .	256
20.19	The F-distribution . . . . .	258
20.20	The t-distribution . . . . .	260
20.21	The Beta Distribution . . . . .	262
20.22	The Logistic Distribution . . . . .	263
20.23	The Pareto Distribution . . . . .	264
20.24	Spherical Vector Distributions . . . . .	265
20.25	The Weibull Distribution . . . . .	266
20.26	The Type-1 Gumbel Distribution . . . . .	267
20.27	The Type-2 Gumbel Distribution . . . . .	268
20.28	The Dirichlet Distribution . . . . .	269
20.29	General Discrete Distributions . . . . .	270
20.30	The Poisson Distribution . . . . .	271
20.31	The Bernoulli Distribution . . . . .	272
20.32	The Binomial Distribution . . . . .	273
20.33	The Multinomial Distribution . . . . .	274
20.34	The Negative Binomial Distribution . . . . .	275
20.35	The Pascal Distribution . . . . .	276
20.36	The Geometric Distribution . . . . .	277
20.37	The Hypergeometric Distribution . . . . .	278
20.38	The Logarithmic Distribution . . . . .	279
20.39	The Wishart Distribution . . . . .	280
20.40	Shuffling and Sampling . . . . .	281
20.41	Examples . . . . .	282
20.42	References and Further Reading . . . . .	284
<b>21</b>	<b>Statistics</b>	<b>287</b>
21.1	Mean, Standard Deviation and Variance . . . . .	287
21.2	Absolute deviation . . . . .	288
21.3	Higher moments (skewness and kurtosis) . . . . .	289
21.4	Autocorrelation . . . . .	289
21.5	Covariance . . . . .	290
21.6	Correlation . . . . .	290
21.7	Weighted Samples . . . . .	290
21.8	Maximum and Minimum values . . . . .	292
21.9	Median and Percentiles . . . . .	293
21.10	Order Statistics . . . . .	293
21.11	Robust Location Estimates . . . . .	294
21.12	Robust Scale Estimates . . . . .	294
21.13	Examples . . . . .	296
21.14	References and Further Reading . . . . .	297
<b>22</b>	<b>Running Statistics</b>	<b>299</b>
22.1	Initializing the Accumulator . . . . .	299
22.2	Adding Data to the Accumulator . . . . .	299
22.3	Current Statistics . . . . .	300
22.4	Quantiles . . . . .	301
22.5	Examples . . . . .	301
22.6	References and Further Reading . . . . .	304



<b>23</b>	<b>Moving Window Statistics</b>	<b>305</b>
23.1	Introduction	305
23.2	Handling Endpoints	305
23.3	Allocation for Moving Window Statistics	306
23.4	Moving Mean	306
23.5	Moving Variance and Standard Deviation	307
23.6	Moving Minimum and Maximum	307
23.7	Moving Sum	308
23.8	Moving Median	308
23.9	Robust Scale Estimation	308
23.10	User-defined Moving Statistics	310
23.11	Accumulators	310
23.12	Examples	312
23.13	References and Further Reading	318
<b>24</b>	<b>Digital Filtering</b>	<b>319</b>
24.1	Introduction	319
24.2	Handling Endpoints	319
24.3	Linear Digital Filters	320
24.4	Nonlinear Digital Filters	321
24.5	Examples	323
24.6	References and Further Reading	333
<b>25</b>	<b>Histograms</b>	<b>335</b>
25.1	The histogram struct	335
25.2	Histogram allocation	336
25.3	Copying Histograms	337
25.4	Updating and accessing histogram elements	337
25.5	Searching histogram ranges	338
25.6	Histogram Statistics	338
25.7	Histogram Operations	338
25.8	Reading and writing histograms	339
25.9	Resampling from histograms	340
25.10	The histogram probability distribution struct	340
25.11	Example programs for histograms	341
25.12	Two dimensional histograms	342
25.13	The 2D histogram struct	342
25.14	2D Histogram allocation	343
25.15	Copying 2D Histograms	343
25.16	Updating and accessing 2D histogram elements	344
25.17	Searching 2D histogram ranges	345
25.18	2D Histogram Statistics	345
25.19	2D Histogram Operations	346
25.20	Reading and writing 2D histograms	346
25.21	Resampling from 2D histograms	347
25.22	Example programs for 2D histograms	348
<b>26</b>	<b>N-tuples</b>	<b>351</b>
26.1	The ntuple struct	351
26.2	Creating ntuples	351
26.3	Opening an existing ntuple file	352
26.4	Writing ntuples	352
26.5	Reading ntuples	352
26.6	Closing an ntuple file	352

26.7	Histogramming ntuple values . . . . .	352
26.8	Examples . . . . .	353
26.9	References and Further Reading . . . . .	356
<b>27</b>	<b>Monte Carlo Integration</b>	<b>357</b>
27.1	Interface . . . . .	357
27.2	PLAIN Monte Carlo . . . . .	358
27.3	MISER . . . . .	359
27.4	VEGAS . . . . .	361
27.5	Examples . . . . .	363
27.6	References and Further Reading . . . . .	366
<b>28</b>	<b>Simulated Annealing</b>	<b>367</b>
28.1	Simulated Annealing algorithm . . . . .	367
28.2	Simulated Annealing functions . . . . .	367
28.3	Examples . . . . .	369
28.4	References and Further Reading . . . . .	381
<b>29</b>	<b>Ordinary Differential Equations</b>	<b>383</b>
29.1	Defining the ODE System . . . . .	383
29.2	Stepping Functions . . . . .	384
29.3	Adaptive Step-size Control . . . . .	386
29.4	Evolution . . . . .	388
29.5	Driver . . . . .	389
29.6	Examples . . . . .	390
29.7	References and Further Reading . . . . .	394
<b>30</b>	<b>Interpolation</b>	<b>395</b>
30.1	Introduction to 1D Interpolation . . . . .	395
30.2	1D Interpolation Functions . . . . .	395
30.3	1D Interpolation Types . . . . .	396
30.4	1D Index Look-up and Acceleration . . . . .	397
30.5	1D Evaluation of Interpolating Functions . . . . .	397
30.6	1D Higher-level Interface . . . . .	398
30.7	1D Interpolation Example Programs . . . . .	399
30.8	Introduction to 2D Interpolation . . . . .	403
30.9	2D Interpolation Functions . . . . .	403
30.10	2D Interpolation Grids . . . . .	404
30.11	2D Interpolation Types . . . . .	404
30.12	2D Evaluation of Interpolating Functions . . . . .	405
30.13	2D Higher-level Interface . . . . .	406
30.14	2D Interpolation Example programs . . . . .	408
30.15	References and Further Reading . . . . .	409
<b>31</b>	<b>Numerical Differentiation</b>	<b>411</b>
31.1	Functions . . . . .	411
31.2	Examples . . . . .	412
31.3	References and Further Reading . . . . .	413
<b>32</b>	<b>Chebyshev Approximations</b>	<b>415</b>
32.1	Definitions . . . . .	415
32.2	Creation and Calculation of Chebyshev Series . . . . .	415
32.3	Auxiliary Functions . . . . .	416
32.4	Chebyshev Series Evaluation . . . . .	416
32.5	Derivatives and Integrals . . . . .	416

32.6	Examples . . . . .	417
32.7	References and Further Reading . . . . .	418
<b>33</b>	<b>Series Acceleration</b>	<b>419</b>
33.1	Acceleration functions . . . . .	419
33.2	Acceleration functions without error estimation . . . . .	420
33.3	Examples . . . . .	420
33.4	References and Further Reading . . . . .	422
<b>34</b>	<b>Wavelet Transforms</b>	<b>423</b>
34.1	Definitions . . . . .	423
34.2	Initialization . . . . .	423
34.3	Transform Functions . . . . .	424
34.4	Examples . . . . .	426
34.5	References and Further Reading . . . . .	428
<b>35</b>	<b>Discrete Hankel Transforms</b>	<b>431</b>
35.1	Definitions . . . . .	431
35.2	Functions . . . . .	432
35.3	References and Further Reading . . . . .	432
<b>36</b>	<b>One Dimensional Root-Finding</b>	<b>433</b>
36.1	Overview . . . . .	433
36.2	Caveats . . . . .	434
36.3	Initializing the Solver . . . . .	434
36.4	Providing the function to solve . . . . .	435
36.5	Search Bounds and Guesses . . . . .	437
36.6	Iteration . . . . .	437
36.7	Search Stopping Parameters . . . . .	438
36.8	Root Bracketing Algorithms . . . . .	438
36.9	Root Finding Algorithms using Derivatives . . . . .	439
36.10	Examples . . . . .	440
36.11	References and Further Reading . . . . .	445
<b>37</b>	<b>One Dimensional Minimization</b>	<b>447</b>
37.1	Overview . . . . .	447
37.2	Caveats . . . . .	448
37.3	Initializing the Minimizer . . . . .	448
37.4	Providing the function to minimize . . . . .	449
37.5	Iteration . . . . .	449
37.6	Stopping Parameters . . . . .	450
37.7	Minimization Algorithms . . . . .	450
37.8	Examples . . . . .	451
37.9	References and Further Reading . . . . .	453
<b>38</b>	<b>Multidimensional Root-Finding</b>	<b>455</b>
38.1	Overview . . . . .	455
38.2	Initializing the Solver . . . . .	456
38.3	Providing the function to solve . . . . .	457
38.4	Iteration . . . . .	459
38.5	Search Stopping Parameters . . . . .	460
38.6	Algorithms using Derivatives . . . . .	460
38.7	Algorithms without Derivatives . . . . .	462
38.8	Examples . . . . .	463
38.9	References and Further Reading . . . . .	467

<b>39</b>	<b>Multidimensional Minimization</b>	<b>469</b>
39.1	Overview . . . . .	469
39.2	Caveats . . . . .	470
39.3	Initializing the Multidimensional Minimizer . . . . .	470
39.4	Providing a function to minimize . . . . .	471
39.5	Iteration . . . . .	473
39.6	Stopping Criteria . . . . .	473
39.7	Algorithms with Derivatives . . . . .	474
39.8	Algorithms without Derivatives . . . . .	475
39.9	Examples . . . . .	476
39.10	References and Further Reading . . . . .	480
<b>40</b>	<b>Linear Least-Squares Fitting</b>	<b>481</b>
40.1	Overview . . . . .	481
40.2	Linear regression . . . . .	482
40.3	Multi-parameter regression . . . . .	483
40.4	Regularized regression . . . . .	485
40.5	Robust linear regression . . . . .	489
40.6	Large dense linear systems . . . . .	493
40.7	Troubleshooting . . . . .	497
40.8	Examples . . . . .	497
40.9	References and Further Reading . . . . .	520
<b>41</b>	<b>Nonlinear Least-Squares Fitting</b>	<b>521</b>
41.1	Overview . . . . .	521
41.2	Solving the Trust Region Subproblem (TRS) . . . . .	522
41.3	Weighted Nonlinear Least-Squares . . . . .	525
41.4	Tunable Parameters . . . . .	525
41.5	Initializing the Solver . . . . .	529
41.6	Providing the Function to be Minimized . . . . .	530
41.7	Iteration . . . . .	532
41.8	Testing for Convergence . . . . .	534
41.9	High Level Driver . . . . .	534
41.10	Covariance matrix of best fit parameters . . . . .	535
41.11	Troubleshooting . . . . .	536
41.12	Examples . . . . .	536
41.13	References and Further Reading . . . . .	565
<b>42</b>	<b>Basis Splines</b>	<b>567</b>
42.1	Overview . . . . .	567
42.2	Allocation of B-splines . . . . .	568
42.3	Initializing the knots vector . . . . .	569
42.4	Properties of B-splines . . . . .	570
42.5	Evaluation of B-splines . . . . .	570
42.6	Evaluation of B-spline derivatives . . . . .	570
42.7	Evaluation of B-spline integrals . . . . .	571
42.8	Least Squares Fitting with B-splines . . . . .	572
42.9	Gram Matrix . . . . .	574
42.10	Interpolation with B-splines . . . . .	575
42.11	Projection onto the B-spline Basis . . . . .	576
42.12	Greville abscissae . . . . .	576
42.13	Examples . . . . .	576
42.14	References and Further Reading . . . . .	605
<b>43</b>	<b>Sparse Matrices</b>	<b>607</b>

43.1	Data types	607
43.2	Sparse Matrix Storage Formats	608
43.3	Overview	609
43.4	Allocation	610
43.5	Accessing Matrix Elements	611
43.6	Initializing Matrix Elements	611
43.7	Reading and Writing Matrices	611
43.8	Copying Matrices	612
43.9	Exchanging Rows and Columns	612
43.10	Matrix Operations	612
43.11	Matrix Properties	613
43.12	Finding Maximum and Minimum Elements	614
43.13	Compressed Format	614
43.14	Conversion Between Sparse and Dense Matrices	614
43.15	Examples	615
43.16	References and Further Reading	617
<b>44</b>	<b>Sparse BLAS Support</b>	<b>619</b>
44.1	Sparse BLAS operations	619
44.2	References and Further Reading	619
<b>45</b>	<b>Sparse Linear Algebra</b>	<b>621</b>
45.1	Overview	621
45.2	Sparse Iterative Solvers	621
45.3	Examples	623
45.4	References and Further Reading	626
<b>46</b>	<b>Physical Constants</b>	<b>627</b>
46.1	Fundamental Constants	627
46.2	Astronomy and Astrophysics	628
46.3	Atomic and Nuclear Physics	628
46.4	Measurement of Time	629
46.5	Imperial Units	629
46.6	Speed and Nautical Units	630
46.7	Printers Units	630
46.8	Volume, Area and Length	630
46.9	Mass and Weight	631
46.10	Thermal Energy and Power	631
46.11	Pressure	632
46.12	Viscosity	632
46.13	Light and Illumination	632
46.14	Radioactivity	633
46.15	Force and Energy	633
46.16	Prefixes	633
46.17	Examples	634
46.18	References and Further Reading	635
<b>47</b>	<b>IEEE floating-point arithmetic</b>	<b>637</b>
47.1	Representation of floating point numbers	637
47.2	Setting up your IEEE environment	639
47.3	References and Further Reading	642
<b>48</b>	<b>Debugging Numerical Programs</b>	<b>643</b>
48.1	Using gdb	643
48.2	Examining floating point registers	644

48.3	Handling floating point exceptions . . . . .	645
48.4	GCC warning options for numerical programs . . . . .	645
48.5	References and Further Reading . . . . .	647
<b>49</b>	<b>Contributors to GSL</b>	<b>649</b>
<b>50</b>	<b>Autoconf Macros</b>	<b>653</b>
<b>51</b>	<b>GSL CBLAS Library</b>	<b>655</b>
51.1	Level 1 . . . . .	655
51.2	Level 2 . . . . .	657
51.3	Level 3 . . . . .	662
51.4	Examples . . . . .	665
<b>52</b>	<b>GNU General Public License</b>	<b>667</b>
<b>53</b>	<b>GNU Free Documentation License</b>	<b>681</b>
	<b>Index</b>	<b>691</b>

## INTRODUCTION

The GNU Scientific Library (GSL) is a collection of routines for numerical computing. The routines have been written from scratch in C, and present a modern Applications Programming Interface (API) for C programmers, allowing wrappers to be written for very high level languages. The source code is distributed under the GNU General Public License.

### 1.1 Routines available in GSL

The library covers a wide range of topics in numerical computing. Routines are available for the following areas,

Complex Numbers	Roots of Polynomials	Special Functions
Vectors and Matrices	Permutations	Combinations
Sorting	BLAS Support	Linear Algebra
CBLAS Library	Fast Fourier Transforms	Eigensystems
Random Numbers	Quadrature	Random Distributions
Quasi-Random Sequences	Histograms	Statistics
Monte Carlo Integration	N-Tuples	Differential Equations
Simulated Annealing	Numerical Differentiation	Interpolation
Series Acceleration	Chebyshev Approximations	Root-Finding
Discrete Hankel Transforms	Least-Squares Fitting	Minimization
IEEE Floating-Point	Physical Constants	Basis Splines
Wavelets	Sparse BLAS Support	Sparse Linear Algebra

The use of these routines is described in this manual. Each chapter provides detailed definitions of the functions, followed by example programs and references to the articles on which the algorithms are based.

Where possible the routines have been based on reliable public-domain packages such as FFTPACK and QUADPACK, which the developers of GSL have reimplemented in C with modern coding conventions.

### 1.2 GSL is Free Software

The subroutines in the GNU Scientific Library are “free software”; this means that everyone is free to use them, and to redistribute them in other free programs. The library is not in the public domain; it is copyrighted and there are conditions on its distribution. These conditions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of the software that they might get from you.

Specifically, we want to make sure that you have the right to share copies of programs that you are given which use the GNU Scientific Library, that you receive their source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of any code which uses the GNU Scientific Library, you must give the recipients all the rights that you have received. You must make sure that they, too, receive or can get the source code, both to the library and the code which uses it. And you must tell them their rights. This means that the library should not be redistributed in proprietary programs.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the GNU Scientific Library. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions for the distribution of software related to the GNU Scientific Library are found in the [GNU General Public License](#). Further information about this license is available from the GNU Project webpage [Frequently Asked Questions about the GNU GPL](#).

The Free Software Foundation also operates a license consulting service for commercial users (contact details available from <http://www.fsf.org>).

### 1.3 Obtaining GSL

The source code for the library can be obtained in different ways, by copying it from a friend, purchasing it on CDROM or downloading it from the internet. A list of public ftp servers which carry the source code can be found on the GNU website, <http://www.gnu.org/software/gsl/>.

The preferred platform for the library is a GNU system, which allows it to take advantage of additional features in the GNU C compiler and GNU C library. However, the library is fully portable and should compile on most systems with a C compiler.

Announcements of new releases, updates and other relevant events are made on the [info-gsl@gnu.org](mailto:info-gsl@gnu.org) mailing list. To subscribe to this low-volume list, send an email of the following form:

To: <code>info-gsl-request@gnu.org</code> Subject: <code>subscribe</code>
--

You will receive a response asking you to reply in order to confirm your subscription.

### 1.4 No Warranty

The software described in this manual has no warranty, it is provided “as is”. It is your responsibility to validate the behavior of the routines and their accuracy using the source code provided, or to purchase support and warranties from commercial redistributors. Consult the [GNU General Public License](#) for further details.



## 1.5 Reporting Bugs

A list of known bugs can be found in the BUGS file included in the GSL distribution or online in the GSL bug tracker.<sup>1</sup> Details of compilation problems can be found in the INSTALL file.

If you find a bug which is not listed in these files, please report it to [bug-gsl@gnu.org](mailto:bug-gsl@gnu.org).

All bug reports should include:

- The version number of GSL
- The hardware and operating system
- The compiler used, including version number and compilation options
- A description of the bug behavior
- A short program which exercises the bug

It is useful if you can check whether the same problem occurs when the library is compiled without optimization. Thank you.

Any errors or omissions in this manual can also be reported to the same address.

## 1.6 Further Information

Additional information, including online copies of this manual, links to related projects, and mailing list archives are available from the website mentioned above.

Any questions about the use and installation of the library can be asked on the mailing list [help-gsl@gnu.org](mailto:help-gsl@gnu.org). To subscribe to this list, send an email of the following form:

To: [help-gsl-request@gnu.org](mailto:help-gsl-request@gnu.org)  
 Subject: subscribe

This mailing list can be used to ask questions not covered by this manual, and to contact the developers of the library.

If you would like to refer to the GNU Scientific Library in a journal article, the recommended way is to cite this reference manual, e.g.:

M. Galassi et al, GNU Scientific Library Reference Manual (3rd Ed.), ISBN 0954612078.

If you want to give a url, use “<http://www.gnu.org/software/gsl/>”.

## 1.7 Conventions used in this manual

This manual contains many examples which can be typed at the keyboard. A command entered at the terminal is shown like this:

\$ command

The first character on the line is the terminal prompt, and should not be typed. The dollar sign \$ is used as the standard prompt in this manual, although some systems may use a different character.

<sup>1</sup> <http://savannah.gnu.org/bugs/?group=gsl>

The examples assume the use of the GNU operating system. There may be minor differences in the output on other systems. The commands for setting environment variables use the Bourne shell syntax of the standard GNU shell (`bash`).

## USING THE LIBRARY

This chapter describes how to compile programs that use GSL, and introduces its conventions.

### 2.1 An Example Program

The following short program demonstrates the use of the library by computing the value of the Bessel function  $J_0(x)$  for  $x = 5$ :

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
    double x = 5.0;
    double y = gsl_sf_bessel_J0 (x);
    printf ("J0(%g) = %.18e\n", x, y);
    return 0;
}
```

The output is shown below, and should be correct to double-precision accuracy<sup>1</sup>,

```
J0(5) = -1.775967713143382642e-01
```

The steps needed to compile this program are described in the following sections.

### 2.2 Compiling and Linking

The library header files are installed in their own `gsl` directory. You should write any preprocessor include statements with a `gsl/` directory prefix thus:

```
#include <gsl/gsl_math.h>
```

If the directory is not installed on the standard search path of your compiler you will also need to provide its location to the preprocessor as a command line flag. The default location of the `gsl` directory is `/usr/local/include/gsl`. A typical compilation command for a source file `example.c` with the GNU C compiler `gcc` is:

---

<sup>1</sup> The last few digits may vary slightly depending on the compiler and platform used—this is normal

```
$ gcc -Wall -I/usr/local/include -c example.c
```

This results in an object file `example.o`. The default include path for `gcc` searches `/usr/local/include` automatically so the `-I` option can actually be omitted when GSL is installed in its default location.

## 2.2.1 Linking programs with the library

The library is installed as a single file, `libgsl.a`. A shared version of the library `libgsl.so` is also installed on systems that support shared libraries. The default location of these files is `/usr/local/lib`. If this directory is not on the standard search path of your linker you will also need to provide its location as a command line flag.

To link against the library you need to specify both the main library and a supporting CBLAS library, which provides standard basic linear algebra subroutines. A suitable CBLAS implementation is provided in the library `libgslcblas.a` if your system does not provide one. The following example shows how to link an application with the library:

```
$ gcc -L/usr/local/lib example.o -lgsl -lgslcblas -lm
```

The default library path for `gcc` searches `/usr/local/lib` automatically so the `-L` option can be omitted when GSL is installed in its default location.

The option `-lm` links with the system math library. On some systems it is not needed.<sup>2</sup>

For a tutorial introduction to the GNU C Compiler and related programs, see “An Introduction to GCC” (ISBN 0954161793).<sup>3</sup>

## 2.2.2 Linking with an alternative BLAS library

The following command line shows how you would link the same application with an alternative CBLAS library `libcblas.a`:

```
$ gcc example.o -lgsl -lcblas -lm
```

For the best performance an optimized platform-specific CBLAS library should be used for `-lcblas`. The library must conform to the CBLAS standard. The ATLAS package provides a portable high-performance BLAS library with a CBLAS interface. It is free software and should be installed for any work requiring fast vector and matrix operations. The following command line will link with the ATLAS library and its CBLAS interface:

```
$ gcc example.o -lgsl -lcblas -latlas -lm
```

If the ATLAS library is installed in a non-standard directory use the `-L` option to add it to the search path, as described above.

For more information about BLAS functions see *BLAS Support*.

---

<sup>2</sup> It is not needed on MacOS X

<sup>3</sup> <http://www.network-theory.co.uk/gcc/intro/>

## 2.3 Shared Libraries

To run a program linked with the shared version of the library the operating system must be able to locate the corresponding `.so` file at runtime. If the library cannot be found, the following error will occur:

```
$ ./a.out
./a.out: error while loading shared libraries:
libgsl.so.0: cannot open shared object file: No such file or directory
```

To avoid this error, either modify the system dynamic linker configuration<sup>4</sup> or define the shell variable `LD_LIBRARY_PATH` to include the directory where the library is installed.

For example, in the Bourne shell (`/bin/sh` or `/bin/bash`), the library search path can be set with the following commands:

```
$ LD_LIBRARY_PATH=/usr/local/lib
$ export LD_LIBRARY_PATH
$ ./example
```

In the C-shell (`/bin/csh` or `/bin/tcsh`) the equivalent command is:

```
% setenv LD_LIBRARY_PATH /usr/local/lib
```

The standard prompt for the C-shell in the example above is the percent character `%`, and should not be typed as part of the command.

To save retyping these commands each session they can be placed in an individual or system-wide login file.

To compile a statically linked version of the program, use the `-static` flag in `gcc`:

```
$ gcc -static example.o -lgsl -lgslcblas -lm
```

## 2.4 ANSI C Compliance

The library is written in ANSI C and is intended to conform to the ANSI C standard (C89). It should be portable to any system with a working ANSI C compiler.

The library does not rely on any non-ANSI extensions in the interface it exports to the user. Programs you write using GSL can be ANSI compliant. Extensions which can be used in a way compatible with pure ANSI C are supported, however, via conditional compilation. This allows the library to take advantage of compiler extensions on those platforms which support them.

When an ANSI C feature is known to be broken on a particular system the library will exclude any related functions at compile-time. This should make it impossible to link a program that would use these functions and give incorrect results.

To avoid namespace conflicts all exported function names and variables have the prefix `gsl_`, while exported macros have the prefix `GSL_`.

<sup>4</sup> `/etc/ld.so.conf` on GNU/Linux systems

## 2.5 Inline functions

The `inline` keyword is not part of the original ANSI C standard (C89) so the library does not export any inline function definitions by default. Inline functions were introduced officially in the newer C99 standard but most C89 compilers have also included `inline` as an extension for a long time.

To allow the use of inline functions, the library provides optional inline versions of performance-critical routines by conditional compilation in the exported header files. The inline versions of these functions can be included by defining the macro `HAVE_INLINE` when compiling an application:

```
$ gcc -Wall -c -DHAVE_INLINE example.c
```

If you use `autoconf` this macro can be defined automatically. If you do not define the macro `HAVE_INLINE` then the slower non-inlined versions of the functions will be used instead.

By default, the actual form of the inline keyword is `extern inline`, which is a `gcc` extension that eliminates unnecessary function definitions. If the form `extern inline` causes problems with other compilers a stricter `autoconf` test can be used, see *Autoconf Macros*.

When compiling with `gcc` in C99 mode (`gcc -std=c99`) the header files automatically switch to C99-compatible inline function declarations instead of `extern inline`. With other C99 compilers, define the macro `GSL_C99_INLINE` to use these declarations.

## 2.6 Long double

In general, the algorithms in the library are written for double precision only. The `long double` type is not supported for actual computation.

One reason for this choice is that the precision of `long double` is platform dependent. The IEEE standard only specifies the minimum precision of extended precision numbers, while the precision of `double` is the same on all platforms.

However, it is sometimes necessary to interact with external data in long-double format, so the vector and matrix datatypes include long-double versions.

It should be noted that in some system libraries the `stdio.h` formatted input/output functions `printf` and `scanf` are not implemented correctly for `long double`. Undefined or incorrect results are avoided by testing these functions during the `configure` stage of library compilation and eliminating certain GSL functions which depend on them if necessary. The corresponding line in the `configure` output looks like this:

```
checking whether printf works with long double... no
```

Consequently when `long double` formatted input/output does not work on a given system it should be impossible to link a program which uses GSL functions dependent on this.

If it is necessary to work on a system which does not support formatted `long double` input/output then the options are to use binary formats or to convert `long double` results into `double` for reading and writing.

## 2.7 Portability functions

To help in writing portable applications GSL provides some implementations of functions that are found in other libraries, such as the BSD math library. You can write your application to use the native versions of these functions, and substitute the GSL versions via a preprocessor macro if they are unavailable on another platform.

For example, after determining whether the BSD function `hypot()` is available you can include the following macro definitions in a file `config.h` with your application:

```
/* Substitute gsl_hypot for missing system hypot */  
  
#ifndef HAVE_HYPOT  
#define hypot gsl_hypot  
#endif
```

The application source files can then use the include command `#include <config.h>` to replace each occurrence of `hypot()` by `gsl_hypot()` when `hypot()` is not available. This substitution can be made automatically if you use `autoconf`, see *Autoconf Macros*.

In most circumstances the best strategy is to use the native versions of these functions when available, and fall back to GSL versions otherwise, since this allows your application to take advantage of any platform-specific optimizations in the system library. This is the strategy used within GSL itself.

## 2.8 Alternative optimized functions

The main implementation of some functions in the library will not be optimal on all architectures. For example, there are several ways to compute a Gaussian random variate and their relative speeds are platform-dependent. In cases like this the library provides alternative implementations of these functions with the same interface. If you write your application using calls to the standard implementation you can select an alternative version later via a preprocessor definition. It is also possible to introduce your own optimized functions this way while retaining portability. The following lines demonstrate the use of a platform-dependent choice of methods for sampling from the Gaussian distribution:

```
#ifdef SPARC  
#define gsl_ran_gaussian gsl_ran_gaussian_ratio_method  
#endif  
#ifdef INTEL  
#define gsl_ran_gaussian my_gaussian  
#endif
```

These lines would be placed in the configuration header file `config.h` of the application, which should then be included by all the source files. Note that the alternative implementations will not produce bit-for-bit identical results, and in the case of random number distributions will produce an entirely different stream of random variates.

## 2.9 Support for different numeric types

Many functions in the library are defined for different numeric types. This feature is implemented by varying the name of the function with a type-related modifier—a primitive form of C++ templates. The modifier is inserted into the function name after the initial module prefix. The following table shows the function names defined for all the numeric types of an imaginary module `gsl_foo` with function `fn()`:

<code>gsl_foo_fn</code>	<code>double</code>
<code>gsl_foo_long_double_fn</code>	<code>long double</code>
<code>gsl_foo_float_fn</code>	<code>float</code>
<code>gsl_foo_long_fn</code>	<code>long</code>
<code>gsl_foo_ulong_fn</code>	<code>unsigned long</code>
<code>gsl_foo_int_fn</code>	<code>int</code>
<code>gsl_foo_uint_fn</code>	<code>unsigned int</code>
<code>gsl_foo_short_fn</code>	<code>short</code>
<code>gsl_foo_ushort_fn</code>	<code>unsigned short</code>
<code>gsl_foo_char_fn</code>	<code>char</code>
<code>gsl_foo_uchar_fn</code>	<code>unsigned char</code>

The normal numeric precision `double` is considered the default and does not require a suffix. For example, the function `gsl_stats_mean()` computes the mean of double precision numbers, while the function `gsl_stats_int_mean()` computes the mean of integers.

A corresponding scheme is used for library defined types, such as `gsl_vector` and `gsl_matrix`. In this case the modifier is appended to the type name. For example, if a module defines a new type-dependent struct or typedef `gsl_foo` it is modified for other types in the following way:

<code>gsl_foo</code>	<code>double</code>
<code>gsl_foo_long_double</code>	<code>long double</code>
<code>gsl_foo_float</code>	<code>float</code>
<code>gsl_foo_long</code>	<code>long</code>
<code>gsl_foo_ulong</code>	<code>unsigned long</code>
<code>gsl_foo_int</code>	<code>int</code>
<code>gsl_foo_uint</code>	<code>unsigned int</code>
<code>gsl_foo_short</code>	<code>short</code>
<code>gsl_foo_ushort</code>	<code>unsigned short</code>
<code>gsl_foo_char</code>	<code>char</code>
<code>gsl_foo_uchar</code>	<code>unsigned char</code>

When a module contains type-dependent definitions the library provides individual header files for each type. The filenames are modified as shown in the below. For convenience the default header includes the definitions for all the types. To include only the double precision header file, or any other specific type, use its individual filename:

```
#include <gsl/gsl_foo.h>           All types
#include <gsl/gsl_foo_double.h>    double
#include <gsl/gsl_foo_long_double.h> long double
#include <gsl/gsl_foo_float.h>     float
#include <gsl/gsl_foo_long.h>      long
#include <gsl/gsl_foo_ulong.h>     unsigned long
#include <gsl/gsl_foo_int.h>       int
#include <gsl/gsl_foo_uint.h>      unsigned int
#include <gsl/gsl_foo_short.h>     short
#include <gsl/gsl_foo_ushort.h>    unsigned short
```

(continues on next page)



(continued from previous page)

```
#include <gsl/gsl_foo_char.h>      char
#include <gsl/gsl_foo_uchar.h>    unsigned char
```

## 2.10 Compatibility with C++

The library header files automatically define functions to have `extern "C"` linkage when included in C++ programs. This allows the functions to be called directly from C++.

To use C++ exception handling within user-defined functions passed to the library as parameters, the library must be built with the additional CFLAGS compilation option `-fexceptions`.

## 2.11 Aliasing of arrays

The library assumes that arrays, vectors and matrices passed as modifiable arguments are not aliased and do not overlap with each other. This removes the need for the library to handle overlapping memory regions as a special case, and allows additional optimizations to be used. If overlapping memory regions are passed as modifiable arguments then the results of such functions will be undefined. If the arguments will not be modified (for example, if a function prototype declares them as `const` arguments) then overlapping or aliased memory regions can be safely used.

## 2.12 Thread-safety

The library can be used in multi-threaded programs. All the functions are thread-safe, in the sense that they do not use static variables. Memory is always associated with objects and not with functions. For functions which use *workspace* objects as temporary storage the workspaces should be allocated on a per-thread basis. For functions which use *table* objects as read-only memory the tables can be used by multiple threads simultaneously. Table arguments are always declared `const` in function prototypes, to indicate that they may be safely accessed by different threads.

There are a small number of static global variables which are used to control the overall behavior of the library (e.g. whether to use range-checking, the function to call on fatal error, etc). These variables are set directly by the user, so they should be initialized once at program startup and not modified by different threads.

## 2.13 Deprecated Functions

From time to time, it may be necessary for the definitions of some functions to be altered or removed from the library. In these circumstances the functions will first be declared *deprecated* and then removed from subsequent versions of the library. Functions that are deprecated can be disabled in the current release by setting the preprocessor definition `GSL_DISABLE_DEPRECATED`. This allows existing code to be tested for forwards compatibility.

## 2.14 Code Reuse

Where possible the routines in the library have been written to avoid dependencies between modules and files. This should make it possible to extract individual functions for use in your own applications, without needing to have the whole library installed. You may need to define certain macros such as `GSL_ERROR` and remove some `#include` statements in order to compile the files as standalone units. Reuse of the library code in this way is encouraged, subject to the terms of the GNU General Public License.

## ERROR HANDLING

This chapter describes the way that GSL functions report and handle errors. By examining the status information returned by every function you can determine whether it succeeded or failed, and if it failed you can find out what the precise cause of failure was. You can also define your own error handling functions to modify the default behavior of the library.

The functions described in this section are declared in the header file `gsl_errno.h`.

### 3.1 Error Reporting

The library follows the thread-safe error reporting conventions of the POSIX Threads library. Functions return a non-zero error code to indicate an error and `0` to indicate success:

```
int status = gsl_function (...)  
  
if (status) { /* an error occurred */  
    .....  
    /* status value specifies the type of error */  
}
```

The routines report an error whenever they cannot perform the task requested of them. For example, a root-finding function would return a non-zero error code if could not converge to the requested accuracy, or exceeded a limit on the number of iterations. Situations like this are a normal occurrence when using any mathematical library and you should check the return status of the functions that you call.

Whenever a routine reports an error the return value specifies the type of error. The return value is analogous to the value of the variable `errno` in the C library. The caller can examine the return code and decide what action to take, including ignoring the error if it is not considered serious.

In addition to reporting errors by return codes the library also has an error handler function `gsl_error()`. This function is called by other library functions when they report an error, just before they return to the caller. The default behavior of the error handler is to print a message and abort the program:

```
gsl: file.c:67: ERROR: invalid argument supplied by user  
Default GSL error handler invoked.  
Aborted
```

The purpose of the `gsl_error()` handler is to provide a function where a breakpoint can be set that will catch library errors when running under the debugger. It is not intended for use in production programs, which should handle any errors using the return codes.

## 3.2 Error Codes

The error code numbers returned by library functions are defined in the file `gsl_errno.h`. They all have the prefix `GSL_` and expand to non-zero constant integer values. Error codes above 1024 are reserved for applications, and are not used by the library. Many of the error codes use the same base name as the corresponding error code in the C library. Here are some of the most common error codes,

int **GSL\_EDOM**

Domain error; used by mathematical functions when an argument value does not fall into the domain over which the function is defined (like `EDOM` in the C library)

int **GSL\_ERANGE**

Range error; used by mathematical functions when the result value is not representable because of overflow or underflow (like `ERANGE` in the C library)

int **GSL\_ENOMEM**

No memory available. The system cannot allocate more virtual memory because its capacity is full (like `ENOMEM` in the C library). This error is reported when a GSL routine encounters problems when trying to allocate memory with `malloc()`.

int **GSL\_EINVAL**

Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function (like `EINVAL` in the C library).

The error codes can be converted into an error message using the function `gsl_strerror()`.

const char \***gsl\_strerror**(const int gsl\_errno)

This function returns a pointer to a string describing the error code `gsl_errno`. For example:

```
printf ("error: %s\n", gsl_strerror (status));
```

would print an error message like `error: output range error` for a status value of `GSL_ERANGE`.

## 3.3 Error Handlers

The default behavior of the GSL error handler is to print a short message and call `abort()`. When this default is in use programs will stop with a core-dump whenever a library routine reports an error. This is intended as a fail-safe default for programs which do not check the return status of library routines (we don't encourage you to write programs this way).

If you turn off the default error handler it is your responsibility to check the return values of routines and handle them yourself. You can also customize the error behavior by providing a new error handler. For example, an alternative error handler could log all errors to a file, ignore certain error conditions (such as underflows), or start the debugger and attach it to the current process when an error occurs.

All GSL error handlers have the type `gsl_error_handler_t`, which is defined in `gsl_errno.h`,

type **gsl\_error\_handler\_t**

This is the type of GSL error handler functions. An error handler will be passed four arguments which specify the reason for the error (a string), the name of the source file in which it occurred (also a string), the line number in that file (an integer) and the error number (an integer). The source file and line number are set at compile time using the `__FILE__` and `__LINE__` directives in the preprocessor. An error handler function returns type `void`. Error handler functions should be defined like this:

```
void handler (const char * reason,
             const char * file,
             int line,
             int gsl_errno)
```

To request the use of your own error handler you need to call the function `gsl_set_error_handler()` which is also declared in `gsl_errno.h`,

`gsl_error_handler_t *gsl_set_error_handler(gsl_error_handler_t *new_handler)`

This function sets a new error handler, `new_handler`, for the GSL library routines. The previous handler is returned (so that you can restore it later). Note that the pointer to a user defined error handler function is stored in a static variable, so there can be only one error handler per program. This function should not be used in multi-threaded programs except to set up a program-wide error handler from a master thread. The following example shows how to set and restore a new error handler:

```
/* save original handler, install new handler */
old_handler = gsl_set_error_handler (&my_handler);

/* code uses new handler */
.....

/* restore original handler */
gsl_set_error_handler (old_handler);
```

To use the default behavior (`abort()` on error) set the error handler to `NULL`:

```
old_handler = gsl_set_error_handler (NULL);
```

`gsl_error_handler_t *gsl_set_error_handler_off()`

This function turns off the error handler by defining an error handler which does nothing. This will cause the program to continue after any error, so the return values from any library routines must be checked. This is the recommended behavior for production programs. The previous handler is returned (so that you can restore it later).

The error behavior can be changed for specific applications by recompiling the library with a customized definition of the `GSL_ERROR` macro in the file `gsl_errno.h`.

## 3.4 Using GSL error reporting in your own functions

If you are writing numerical functions in a program which also uses GSL code you may find it convenient to adopt the same error reporting conventions as in the library.

To report an error you need to call the function `gsl_error()` with a string describing the error and then return an appropriate error code from `gsl_errno.h`, or a special value, such as `NaN`. For convenience the file `gsl_errno.h` defines two macros which carry out these steps:

**GSL\_ERROR**(reason, `gsl_errno`)

This macro reports an error using the GSL conventions and returns a status value of `gsl_errno`. It expands to the following code fragment:

```
gsl_error (reason, __FILE__, __LINE__, gsl_errno);
return gsl_errno;
```

The macro definition in `gsl_errno.h` actually wraps the code in a `do { ... } while (0)` block to prevent possible parsing problems.

Here is an example of how the macro could be used to report that a routine did not achieve a requested tolerance. To report the error the routine needs to return the error code `GSL_ETOL`:

```
if (residual > tolerance)
{
    GSL_ERROR("residual exceeds tolerance", GSL_ETOL);
}
```

**GSL\_ERROR\_VAL**(reason, gsl\_errno, value)

This macro is the same as `GSL_ERROR` but returns a user-defined value of `value` instead of an error code. It can be used for mathematical functions that return a floating point value.

The following example shows how to return a NaN at a mathematical singularity using the `GSL_ERROR_VAL` macro:

```
if (x == 0)
{
    GSL_ERROR_VAL("argument lies on singularity", GSL_ERANGE, GSL_NAN);
}
```

## 3.5 Examples

Here is an example of some code which checks the return value of a function where an error might be reported:

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

...
int status;
size_t n = 37;

gsl_set_error_handler_off();

status = gsl_fft_complex_radix2_forward (data, stride, n);

if (status) {
    if (status == GSL_EINVAL) {
        fprintf (stderr, "invalid argument, n=%d\n", n);
    } else {
        fprintf (stderr, "failed, gsl_errno=%d\n", status);
    }
    exit (-1);
}
...
```

The function `gsl_fft_complex_radix2_forward()` only accepts integer lengths which are a power of two. If the variable `n` is not a power of two then the call to the library function will return `GSL_EINVAL`, indicating that the length argument is invalid. The function call to `gsl_set_error_handler_off()` stops the default error handler from aborting the program. The `else` clause catches any other possible errors.

## MATHEMATICAL FUNCTIONS

This chapter describes basic mathematical functions. Some of these functions are present in system libraries, but the alternative versions given here can be used as a substitute when the system functions are not available.

The functions and macros described in this chapter are defined in the header file `gsl_math.h`.

### 4.1 Mathematical Constants

The library ensures that the standard BSD mathematical constants are defined. For reference, here is a list of the constants:

<code>M_E</code>	The base of exponentials, $e$
<code>M_LOG2E</code>	The base-2 logarithm of $e$ , $\log_2(e)$
<code>M_LOG10E</code>	The base-10 logarithm of $e$ , $\log_{10}(e)$
<code>M_SQRT2</code>	The square root of two, $\sqrt{2}$
<code>M_SQRT1_2</code>	The square root of one-half, $\sqrt{1/2}$
<code>M_SQRT3</code>	The square root of three, $\sqrt{3}$
<code>M_PI</code>	The constant pi, $\pi$
<code>M_PI_2</code>	Pi divided by two, $\pi/2$
<code>M_PI_4</code>	Pi divided by four, $\pi/4$
<code>M_SQRTPI</code>	The square root of pi, $\sqrt{\pi}$
<code>M_2_SQRTPI</code>	Two divided by the square root of pi, $2/\sqrt{\pi}$
<code>M_1_PI</code>	The reciprocal of pi, $1/\pi$
<code>M_2_PI</code>	Twice the reciprocal of pi, $2/\pi$
<code>M_LN10</code>	The natural logarithm of ten, $\ln(10)$
<code>M_LN2</code>	The natural logarithm of two, $\ln(2)$
<code>M_LNPI</code>	The natural logarithm of pi, $\ln(\pi)$
<code>M_EULER</code>	Euler's constant, $\gamma$

## 4.2 Infinities and Not-a-number

### GSL\_POSINF

This macro contains the IEEE representation of positive infinity,  $+\infty$ . It is computed from the expression `+1.0/0.0`.

### GSL\_NEGINF

This macro contains the IEEE representation of negative infinity,  $-\infty$ . It is computed from the expression `-1.0/0.0`.

### GSL\_NAN

This macro contains the IEEE representation of the Not-a-Number symbol, NaN. It is computed from the ratio `0.0/0.0`.

int **gsl\_isnan**(const double x)

This function returns 1 if  $x$  is not-a-number.

int **gsl\_isinf**(const double x)

This function returns +1 if  $x$  is positive infinity, -1 if  $x$  is negative infinity and 0 otherwise.<sup>1</sup>

int **gsl\_finite**(const double x)

This function returns 1 if  $x$  is a real number, and 0 if it is infinite or not-a-number.

## 4.3 Elementary Functions

The following routines provide portable implementations of functions found in the BSD math library. When native versions are not available the functions described here can be used instead. The substitution can be made automatically if you use `autoconf` to compile your application (see *Portability functions*).

double **gsl\_log1p**(const double x)

This function computes the value of  $\log(1+x)$  in a way that is accurate for small  $x$ . It provides an alternative to the BSD math function `log1p(x)`.

double **gsl\_expm1**(const double x)

This function computes the value of  $\exp(x) - 1$  in a way that is accurate for small  $x$ . It provides an alternative to the BSD math function `expm1(x)`.

double **gsl\_hypot**(const double x, const double y)

This function computes the value of  $\sqrt{x^2 + y^2}$  in a way that avoids overflow. It provides an alternative to the BSD math function `hypot(x, y)`.

double **gsl\_hypot3**(const double x, const double y, const double z)

This function computes the value of  $\sqrt{x^2 + y^2 + z^2}$  in a way that avoids overflow.

double **gsl\_acosh**(const double x)

This function computes the value of  $\operatorname{arccosh}(x)$ . It provides an alternative to the standard math function `acosh(x)`.

double **gsl\_asinh**(const double x)

This function computes the value of  $\operatorname{arcsinh}(x)$ . It provides an alternative to the standard math function `asinh(x)`.

---

<sup>1</sup> Note that the C99 standard only requires the system `isinf()` function to return a non-zero value, without the sign of the infinity. The implementation in some earlier versions of GSL used the system `isinf()` function and may have this behavior on some platforms. Therefore, it is advisable to test the sign of  $x$  separately, if needed, rather than relying the sign of the return value from `gsl_isinf()`.



double **gsl\_atanh**(const double x)

This function computes the value of  $\operatorname{arctanh}(x)$ . It provides an alternative to the standard math function `atanh(x)`.

double **gsl\_ldexp**(double x, int e)

This function computes the value of  $x * 2^e$ . It provides an alternative to the standard math function `ldexp(x, e)`.

double **gsl\_frexp**(double x, int \*e)

This function splits the number  $x$  into its normalized fraction  $f$  and exponent  $e$ , such that  $x = f * 2^e$  and  $0.5 \leq f < 1$ . The function returns  $f$  and stores the exponent in  $e$ . If  $x$  is zero, both  $f$  and  $e$  are set to zero. This function provides an alternative to the standard math function `frexp(x, e)`.

## 4.4 Small integer powers

A common complaint about the standard C library is its lack of a function for calculating (small) integer powers. GSL provides some simple functions to fill this gap. For reasons of efficiency, these functions do not check for overflow or underflow conditions.

double **gsl\_pow\_int**(double x, int n)

double **gsl\_pow\_uint**(double x, unsigned int n)

These routines compute the power  $x^n$  for integer  $n$ . The power is computed efficiently—for example,  $x^8$  is computed as  $((x^2)^2)^2$ , requiring only 3 multiplications. A version of this function which also computes the numerical error in the result is available as `gsl_sf_pow_int_e()`.

double **gsl\_pow\_2**(const double x)

double **gsl\_pow\_3**(const double x)

double **gsl\_pow\_4**(const double x)

double **gsl\_pow\_5**(const double x)

double **gsl\_pow\_6**(const double x)

double **gsl\_pow\_7**(const double x)

double **gsl\_pow\_8**(const double x)

double **gsl\_pow\_9**(const double x)

These functions can be used to compute small integer powers  $x^2$ ,  $x^3$ , etc. efficiently. The functions will be inlined when `HAVE_INLINE` is defined, so that use of these functions should be as efficient as explicitly writing the corresponding product expression:

```
#include <gsl/gsl_math.h>
double y = gsl_pow_4 (3.141) /* compute 3.141**4 */
```

## 4.5 Testing the Sign of Numbers

### **GSL\_SIGN**(*x*)

This macro returns the sign of *x*. It is defined as  $((x) >= 0 ? 1 : -1)$ . Note that with this definition the sign of zero is positive (regardless of its IEEE sign bit).

## 4.6 Testing for Odd and Even Numbers

### **GSL\_IS\_ODD**(*n*)

This macro evaluates to 1 if *n* is odd and 0 if *n* is even. The argument *n* must be of integer type.

### **GSL\_IS\_EVEN**(*n*)

This macro is the opposite of *GSL\_IS\_ODD*. It evaluates to 1 if *n* is even and 0 if *n* is odd. The argument *n* must be of integer type.

## 4.7 Maximum and Minimum functions

Note that the following macros perform multiple evaluations of their arguments, so they should not be used with arguments that have side effects (such as a call to a random number generator).

### **GSL\_MAX**(*a*, *b*)

This macro returns the maximum of *a* and *b*. It is defined as  $((a) > (b) ? (a) : (b))$ .

### **GSL\_MIN**(*a*, *b*)

This macro returns the minimum of *a* and *b*. It is defined as  $((a) < (b) ? (a) : (b))$ .

### extern inline double **GSL\_MAX\_DBL**(double *a*, double *b*)

This function returns the maximum of the double precision numbers *a* and *b* using an inline function. The use of a function allows for type checking of the arguments as an extra safety feature. On platforms where inline functions are not available the macro *GSL\_MAX* will be automatically substituted.

### extern inline double **GSL\_MIN\_DBL**(double *a*, double *b*)

This function returns the minimum of the double precision numbers *a* and *b* using an inline function. The use of a function allows for type checking of the arguments as an extra safety feature. On platforms where inline functions are not available the macro *GSL\_MIN* will be automatically substituted.

### extern inline int **GSL\_MAX\_INT**(int *a*, int *b*)

### extern inline int **GSL\_MIN\_INT**(int *a*, int *b*)

These functions return the maximum or minimum of the integers *a* and *b* using an inline function. On platforms where inline functions are not available the macros *GSL\_MAX* or *GSL\_MIN* will be automatically substituted.

### extern inline long double **GSL\_MAX\_LDBL**(long double *a*, long double *b*)

### extern inline long double **GSL\_MIN\_LDBL**(long double *a*, long double *b*)

These functions return the maximum or minimum of the long doubles *a* and *b* using an inline function. On platforms where inline functions are not available the macros *GSL\_MAX* or *GSL\_MIN* will be automatically substituted.

## 4.8 Approximate Comparison of Floating Point Numbers

It is sometimes useful to be able to compare two floating point numbers approximately, to allow for rounding and truncation errors. The following function implements the approximate floating-point comparison algorithm proposed by D.E. Knuth in Section 4.2.2 of “Seminumerical Algorithms” (3rd edition).

int **gsl\_fcmp**(double x, double y, double epsilon)

This function determines whether  $x$  and  $y$  are approximately equal to a relative accuracy *epsilon*.

The relative accuracy is measured using an interval of size  $2\delta$ , where  $\delta = 2^k\epsilon$  and  $k$  is the maximum base-2 exponent of  $x$  and  $y$  as computed by the function `frexp()`.

If  $x$  and  $y$  lie within this interval, they are considered approximately equal and the function returns 0. Otherwise if  $x < y$ , the function returns  $-1$ , or if  $x > y$ , the function returns  $+1$ .

Note that  $x$  and  $y$  are compared to relative accuracy, so this function is not suitable for testing whether a value is approximately zero.

The implementation is based on the package `fcmp` by T.C. Belding.



## COMPLEX NUMBERS

The functions described in this chapter provide support for complex numbers. The algorithms take care to avoid unnecessary intermediate underflows and overflows, allowing the functions to be evaluated over as much of the complex plane as possible.

For multiple-valued functions the branch cuts have been chosen to follow the conventions of Abramowitz and Stegun. The functions return principal values which are the same as those in GNU Calc, which in turn are the same as those in “Common Lisp, The Language (Second Edition)”<sup>1</sup> and the HP-28/48 series of calculators.

The complex types are defined in the header file `gsl_complex.h`, while the corresponding complex functions and arithmetic operations are defined in `gsl_complex_math.h`.

### 5.1 Representation of complex numbers

Complex numbers are represented using the type `gsl_complex`. The default interface defines `gsl_complex` as:

```
typedef struct
{
    double dat[2];
} gsl_complex;
```

The real and imaginary part are stored in contiguous elements of a two element array. This eliminates any padding between the real and imaginary parts, `dat[0]` and `dat[1]`, allowing the struct to be mapped correctly onto packed complex arrays.

If a C compiler is available which supports the C11 standard, and the `<complex.h>` header file is included *prior* to `gsl_complex.h`, then `gsl_complex` will be defined to be the native C complex type:

```
typedef double complex gsl_complex
```

This allows users to use `gsl_complex` in ordinary operations such as:

```
gsl_complex x = 2 + 5 * I;
gsl_complex y = x + (3 - 4*I);
```

---

**Important:** Native C support for complex numbers was introduced in the C99 standard, and additional functionality was added in C11. When `<complex.h>` is included in a user’s program prior to `gsl_complex.h`, GSL uses the new C11 functionality to define the `GSL_REAL` and `GSL_IMAG` macros. It does not appear possible to properly define these macros using the C99 standard, and so using a C99 compiler will not define `gsl_complex` to the native complex type.

---

<sup>1</sup> Note that the first edition uses different definitions.

Some compilers, such as the gcc 4.8 series implement only a portion of the C11 standard and so they may fail to correctly compile GSL code when a user tries to turn on native complex functionality. A workaround for this issue is to either remove `<complex.h>` from the include list, or add `-DGSL_COMPLEX_LEGACY` to the compiler flags, which will use the older struct-based definition of `gsl_complex`.

---

## 5.2 Complex number macros

The following C macros offer convenient ways to manipulate complex numbers.

**GSL\_REAL**(*z*)

**GSL\_IMAG**(*z*)

These macros return a memory location (lvalue) corresponding to the real and imaginary parts respectively of the complex number *z*. This allows users to perform operations like:

```
gsl_complex x, y;

GSL_REAL(x) = 4;
GSL_IMAG(x) = 2;

GSL_REAL(y) = GSL_REAL(x);
GSL_IMAG(y) = GSL_REAL(x);
```

In other words, these macros can both read and write to the real and imaginary parts of a complex variable.

**GSL\_SET\_COMPLEX**(*zp*, *x*, *y*)

This macro uses the Cartesian components (*x*, *y*) to set the real and imaginary parts of the complex number pointed to by *zp*. For example:

```
GSL_SET_COMPLEX(&z, 3, 4)
```

sets *z* to be  $3 + 4i$ .

## 5.3 Assigning complex numbers

`gsl_complex` **gsl\_complex\_rect**(double *x*, double *y*)

This function uses the rectangular Cartesian components (*x*, *y*) to return the complex number  $z = x + iy$ . An inline version of this function is used when `HAVE_INLINE` is defined.

`gsl_complex` **gsl\_complex\_polar**(double *r*, double *theta*)

This function returns the complex number  $z = r \exp(i\theta) = r(\cos(\theta) + i \sin(\theta))$  from the polar representation (*r*, *theta*).

## 5.4 Properties of complex numbers

double **gsl\_complex\_arg**(gsl\_complex z)

This function returns the argument of the complex number  $z$ ,  $\arg(z)$ , where  $-\pi < \arg(z) \leq \pi$ .

double **gsl\_complex\_abs**(gsl\_complex z)

This function returns the magnitude of the complex number  $z$ ,  $|z|$ .

double **gsl\_complex\_abs2**(gsl\_complex z)

This function returns the squared magnitude of the complex number  $z$ ,  $|z|^2$ .

double **gsl\_complex\_logabs**(gsl\_complex z)

This function returns the natural logarithm of the magnitude of the complex number  $z$ ,  $\log|z|$ . It allows an accurate evaluation of  $\log|z|$  when  $|z|$  is close to one. The direct evaluation of  $\log(\text{gsl\_complex\_abs}(z))$  would lead to a loss of precision in this case.

## 5.5 Complex arithmetic operators

gsl\_complex **gsl\_complex\_add**(gsl\_complex a, gsl\_complex b)

This function returns the sum of the complex numbers  $a$  and  $b$ ,  $z = a + b$ .

gsl\_complex **gsl\_complex\_sub**(gsl\_complex a, gsl\_complex b)

This function returns the difference of the complex numbers  $a$  and  $b$ ,  $z = a - b$ .

gsl\_complex **gsl\_complex\_mul**(gsl\_complex a, gsl\_complex b)

This function returns the product of the complex numbers  $a$  and  $b$ ,  $z = ab$ .

gsl\_complex **gsl\_complex\_div**(gsl\_complex a, gsl\_complex b)

This function returns the quotient of the complex numbers  $a$  and  $b$ ,  $z = a/b$ .

gsl\_complex **gsl\_complex\_add\_real**(gsl\_complex a, double x)

This function returns the sum of the complex number  $a$  and the real number  $x$ ,  $z = a + x$ .

gsl\_complex **gsl\_complex\_sub\_real**(gsl\_complex a, double x)

This function returns the difference of the complex number  $a$  and the real number  $x$ ,  $z = a - x$ .

gsl\_complex **gsl\_complex\_mul\_real**(gsl\_complex a, double x)

This function returns the product of the complex number  $a$  and the real number  $x$ ,  $z = ax$ .

gsl\_complex **gsl\_complex\_div\_real**(gsl\_complex a, double x)

This function returns the quotient of the complex number  $a$  and the real number  $x$ ,  $z = a/x$ .

gsl\_complex **gsl\_complex\_add\_imag**(gsl\_complex a, double y)

This function returns the sum of the complex number  $a$  and the imaginary number  $iy$ ,  $z = a + iy$ .

gsl\_complex **gsl\_complex\_sub\_imag**(gsl\_complex a, double y)

This function returns the difference of the complex number  $a$  and the imaginary number  $iy$ ,  $z = a - iy$ .

gsl\_complex **gsl\_complex\_mul\_imag**(gsl\_complex a, double y)

This function returns the product of the complex number  $a$  and the imaginary number  $iy$ ,  $z = a * (iy)$ .

gsl\_complex **gsl\_complex\_div\_imag**(gsl\_complex a, double y)

This function returns the quotient of the complex number  $a$  and the imaginary number  $iy$ ,  $z = a/(iy)$ .

gsl\_complex **gsl\_complex\_conjugate**(gsl\_complex z)

This function returns the complex conjugate of the complex number  $z$ ,  $z^* = x - iy$ .

gsl\_complex **gsl\_complex\_inverse**(gsl\_complex z)

This function returns the inverse, or reciprocal, of the complex number  $z$ ,  $1/z = (x - iy)/(x^2 + y^2)$ .

`gsl_complex` **gsl\_complex\_negative**(`gsl_complex z`)

This function returns the negative of the complex number  $z$ ,  $-z = (-x) + i(-y)$ .

## 5.6 Elementary Complex Functions

`gsl_complex` **gsl\_complex\_sqrt**(`gsl_complex z`)

This function returns the square root of the complex number  $z$ ,  $\sqrt{z}$ . The branch cut is the negative real axis. The result always lies in the right half of the complex plane.

`gsl_complex` **gsl\_complex\_sqrt\_real**(`double x`)

This function returns the complex square root of the real number  $x$ , where  $x$  may be negative.

`gsl_complex` **gsl\_complex\_pow**(`gsl_complex z`, `gsl_complex a`)

The function returns the complex number  $z$  raised to the complex power  $a$ ,  $z^a$ . This is computed as  $\exp(\log(z) * a)$  using complex logarithms and complex exponentials.

`gsl_complex` **gsl\_complex\_pow\_real**(`gsl_complex z`, `double x`)

This function returns the complex number  $z$  raised to the real power  $x$ ,  $z^x$ .

`gsl_complex` **gsl\_complex\_exp**(`gsl_complex z`)

This function returns the complex exponential of the complex number  $z$ ,  $\exp(z)$ .

`gsl_complex` **gsl\_complex\_log**(`gsl_complex z`)

This function returns the complex natural logarithm (base  $e$ ) of the complex number  $z$ ,  $\log(z)$ . The branch cut is the negative real axis.

`gsl_complex` **gsl\_complex\_log10**(`gsl_complex z`)

This function returns the complex base-10 logarithm of the complex number  $z$ ,  $\log_{10}(z)$ .

`gsl_complex` **gsl\_complex\_log\_b**(`gsl_complex z`, `gsl_complex b`)

This function returns the complex base- $b$  logarithm of the complex number  $z$ ,  $\log_b(z)$ . This quantity is computed as the ratio  $\log(z)/\log(b)$ .

## 5.7 Complex Trigonometric Functions

`gsl_complex` **gsl\_complex\_sin**(`gsl_complex z`)

This function returns the complex sine of the complex number  $z$ ,  $\sin(z) = (\exp(iz) - \exp(-iz))/(2i)$ .

`gsl_complex` **gsl\_complex\_cos**(`gsl_complex z`)

This function returns the complex cosine of the complex number  $z$ ,  $\cos(z) = (\exp(iz) + \exp(-iz))/2$ .

`gsl_complex` **gsl\_complex\_tan**(`gsl_complex z`)

This function returns the complex tangent of the complex number  $z$ ,  $\tan(z) = \sin(z)/\cos(z)$ .

`gsl_complex` **gsl\_complex\_sec**(`gsl_complex z`)

This function returns the complex secant of the complex number  $z$ ,  $\sec(z) = 1/\cos(z)$ .

`gsl_complex` **gsl\_complex\_csc**(`gsl_complex z`)

This function returns the complex cosecant of the complex number  $z$ ,  $\csc(z) = 1/\sin(z)$ .

`gsl_complex` **gsl\_complex\_cot**(`gsl_complex z`)

This function returns the complex cotangent of the complex number  $z$ ,  $\cot(z) = 1/\tan(z)$ .



## 5.8 Inverse Complex Trigonometric Functions

`gsl_complex gsl_complex_arcsin(gsl_complex z)`

This function returns the complex arcsine of the complex number  $z$ ,  $\arcsin(z)$ . The branch cuts are on the real axis, less than  $-1$  and greater than  $1$ .

`gsl_complex gsl_complex_arcsin_real(double z)`

This function returns the complex arcsine of the real number  $z$ ,  $\arcsin(z)$ . For  $z$  between  $-1$  and  $1$ , the function returns a real value in the range  $[-\pi/2, \pi/2]$ . For  $z$  less than  $-1$  the result has a real part of  $-\pi/2$  and a positive imaginary part. For  $z$  greater than  $1$  the result has a real part of  $\pi/2$  and a negative imaginary part.

`gsl_complex gsl_complex_arccos(gsl_complex z)`

This function returns the complex arccosine of the complex number  $z$ ,  $\arccos(z)$ . The branch cuts are on the real axis, less than  $-1$  and greater than  $1$ .

`gsl_complex gsl_complex_arccos_real(double z)`

This function returns the complex arccosine of the real number  $z$ ,  $\arccos(z)$ . For  $z$  between  $-1$  and  $1$ , the function returns a real value in the range  $[0, \pi]$ . For  $z$  less than  $-1$  the result has a real part of  $\pi$  and a negative imaginary part. For  $z$  greater than  $1$  the result is purely imaginary and positive.

`gsl_complex gsl_complex_arctan(gsl_complex z)`

This function returns the complex arctangent of the complex number  $z$ ,  $\arctan(z)$ . The branch cuts are on the imaginary axis, below  $-i$  and above  $i$ .

`gsl_complex gsl_complex_arcsec(gsl_complex z)`

This function returns the complex arcsecant of the complex number  $z$ ,  $\operatorname{arcsec}(z) = \arccos(1/z)$ .

`gsl_complex gsl_complex_arcsec_real(double z)`

This function returns the complex arcsecant of the real number  $z$ ,  $\operatorname{arcsec}(z) = \arccos(1/z)$ .

`gsl_complex gsl_complex_arccsc(gsl_complex z)`

This function returns the complex arccosecant of the complex number  $z$ ,  $\operatorname{arccsc}(z) = \arcsin(1/z)$ .

`gsl_complex gsl_complex_arccsc_real(double z)`

This function returns the complex arccosecant of the real number  $z$ ,  $\operatorname{arccsc}(z) = \arcsin(1/z)$ .

`gsl_complex gsl_complex_arccot(gsl_complex z)`

This function returns the complex arccotangent of the complex number  $z$ ,  $\operatorname{arccot}(z) = \arctan(1/z)$ .

## 5.9 Complex Hyperbolic Functions

`gsl_complex gsl_complex_sinh(gsl_complex z)`

This function returns the complex hyperbolic sine of the complex number  $z$ ,  $\sinh(z) = (\exp(z) - \exp(-z))/2$ .

`gsl_complex gsl_complex_cosh(gsl_complex z)`

This function returns the complex hyperbolic cosine of the complex number  $z$ ,  $\cosh(z) = (\exp(z) + \exp(-z))/2$ .

`gsl_complex gsl_complex_tanh(gsl_complex z)`

This function returns the complex hyperbolic tangent of the complex number  $z$ ,  $\tanh(z) = \sinh(z)/\cosh(z)$ .

`gsl_complex gsl_complex_sech(gsl_complex z)`

This function returns the complex hyperbolic secant of the complex number  $z$ ,  $\operatorname{sech}(z) = 1/\cosh(z)$ .

`gsl_complex gsl_complex_csch(gsl_complex z)`

This function returns the complex hyperbolic cosecant of the complex number  $z$ ,  $\operatorname{csch}(z) = 1/\sinh(z)$ .

`gsl_complex gsl_complex_coth(gsl_complex z)`

This function returns the complex hyperbolic cotangent of the complex number  $z$ ,  $\operatorname{coth}(z) = 1/\tanh(z)$ .

## 5.10 Inverse Complex Hyperbolic Functions

`gsl_complex` **`gsl_complex_arcsinh`**(`gsl_complex z`)

This function returns the complex hyperbolic arcsine of the complex number  $z$ ,  $\operatorname{arcsinh}(z)$ . The branch cuts are on the imaginary axis, below  $-i$  and above  $i$ .

`gsl_complex` **`gsl_complex_arccosh`**(`gsl_complex z`)

This function returns the complex hyperbolic arccosine of the complex number  $z$ ,  $\operatorname{arccosh}(z)$ . The branch cut is on the real axis, less than 1. Note that in this case we use the negative square root in formula 4.6.21 of Abramowitz & Stegun giving  $\operatorname{arccosh}(z) = \log(z - \sqrt{z^2 - 1})$ .

`gsl_complex` **`gsl_complex_arccosh_real`**(`double z`)

This function returns the complex hyperbolic arccosine of the real number  $z$ ,  $\operatorname{arccosh}(z)$ .

`gsl_complex` **`gsl_complex_arctanh`**(`gsl_complex z`)

This function returns the complex hyperbolic arctangent of the complex number  $z$ ,  $\operatorname{arctanh}(z)$ . The branch cuts are on the real axis, less than  $-1$  and greater than  $1$ .

`gsl_complex` **`gsl_complex_arctanh_real`**(`double z`)

This function returns the complex hyperbolic arctangent of the real number  $z$ ,  $\operatorname{arctanh}(z)$ .

`gsl_complex` **`gsl_complex_arcsech`**(`gsl_complex z`)

This function returns the complex hyperbolic arcsecant of the complex number  $z$ ,  $\operatorname{arcsech}(z) = \operatorname{arccosh}(1/z)$ .

`gsl_complex` **`gsl_complex_arccsch`**(`gsl_complex z`)

This function returns the complex hyperbolic arccosecant of the complex number  $z$ ,  $\operatorname{arccsch}(z) = \operatorname{arcsinh}(1/z)$ .

`gsl_complex` **`gsl_complex_arccoth`**(`gsl_complex z`)

This function returns the complex hyperbolic arccotangent of the complex number  $z$ ,  $\operatorname{arccoth}(z) = \operatorname{arctanh}(1/z)$ .

## 5.11 References and Further Reading

The implementations of the elementary and trigonometric functions are based on the following papers,

- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, “Implementing Complex Elementary Functions Using Exception Handling”, *ACM Transactions on Mathematical Software*, Volume 20 (1994), pp 215–244, Corrigenda, p553
- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, “Implementing the complex arcsin and arccosine functions using exception handling”, *ACM Transactions on Mathematical Software*, Volume 23 (1997) pp 299–335

The general formulas and details of branch cuts can be found in the following books,

- Abramowitz and Stegun, *Handbook of Mathematical Functions*, “Circular Functions in Terms of Real and Imaginary Parts”, Formulas 4.3.55–58, “Inverse Circular Functions in Terms of Real and Imaginary Parts”, Formulas 4.4.37–39, “Hyperbolic Functions in Terms of Real and Imaginary Parts”, Formulas 4.5.49–52, “Inverse Hyperbolic Functions—relation to Inverse Circular Functions”, Formulas 4.6.14–19.
- Dave Gillespie, *Calc Manual*, Free Software Foundation, ISBN 1-882114-18-3

## POLYNOMIALS

This chapter describes functions for evaluating and solving polynomials. There are routines for finding real and complex roots of quadratic and cubic equations using analytic methods. An iterative polynomial solver is also available for finding the roots of general polynomials with real coefficients (of any order). The functions are declared in the header file `gsl_poly.h`.

### 6.1 Polynomial Evaluation

The functions described here evaluate the polynomial

$$P(x) = c[0] + c[1]x + c[2]x^2 + \cdots + c[*len* - 1]x^{*len*-1}$$

using Horner's method for stability. Inline versions of these functions are used when `HAVE_INLINE` is defined.

double **gsl\_poly\_eval**(const double c[], const int len, const double x)

This function evaluates a polynomial with real coefficients for the real variable  $x$ .

gsl\_complex **gsl\_poly\_complex\_eval**(const double c[], const int len, const gsl\_complex z)

This function evaluates a polynomial with real coefficients for the complex variable  $z$ .

gsl\_complex **gsl\_complex\_poly\_complex\_eval**(const gsl\_complex c[], const int len, const gsl\_complex z)

This function evaluates a polynomial with complex coefficients for the complex variable  $z$ .

int **gsl\_poly\_eval\_derivs**(const double c[], const size\_t lenc, const double x, double res[], const size\_t lenres)

This function evaluates a polynomial and its derivatives storing the results in the array `res` of size `lenres`. The output array contains the values of  $d^k P(x)/dx^k$  for the specified value of  $x$  starting with  $k = 0$ .

### 6.2 Divided Difference Representation of Polynomials

The functions described here manipulate polynomials stored in Newton's divided-difference representation. The use of divided-differences is described in Abramowitz & Stegun sections 25.1.4 and 25.2.26, and Burden and Faires, chapter 3, and discussed briefly below.

Given a function  $f(x)$ , an  $n$ th degree interpolating polynomial  $P_n(x)$  can be constructed which agrees with  $f$  at  $n + 1$  distinct points  $x_0, x_1, \dots, x_n$ . This polynomial can be written in a form known as Newton's divided-difference representation

$$P_n(x) = f(x_0) + \sum_{k=1}^n [x_0, x_1, \dots, x_k](x - x_0)(x - x_1) \cdots (x - x_{k-1})$$

where the divided differences  $[x_0, x_1, \dots, x_k]$  are defined in section 25.1.4 of Abramowitz and Stegun. Additionally, it is possible to construct an interpolating polynomial of degree  $2n + 1$  which also matches the first derivatives of  $f$  at the points  $x_0, x_1, \dots, x_n$ . This is called the Hermite interpolating polynomial and is defined as

$$H_{2n+1}(x) = f(z_0) + \sum_{k=1}^{2n+1} [z_0, z_1, \dots, z_k](x - z_0)(x - z_1) \cdots (x - z_{k-1})$$

where the elements of  $z = \{x_0, x_0, x_1, x_1, \dots, x_n, x_n\}$  are defined by  $z_{2k} = z_{2k+1} = x_k$ . The divided-differences  $[z_0, z_1, \dots, z_k]$  are discussed in Burden and Faires, section 3.4.

int **gsl\_poly\_dd\_init**(double dd[], const double xa[], const double ya[], size\_t size)

This function computes a divided-difference representation of the interpolating polynomial for the points  $(x, y)$  stored in the arrays **xa** and **ya** of length **size**. On output the divided-differences of  $(\mathbf{xa}, \mathbf{ya})$  are stored in the array **dd**, also of length **size**. Using the notation above,  $dd[k] = [x_0, x_1, \dots, x_k]$ .

double **gsl\_poly\_dd\_eval**(const double dd[], const double xa[], const size\_t size, const double x)

This function evaluates the polynomial stored in divided-difference form in the arrays **dd** and **xa** of length **size** at the point **x**. An inline version of this function is used when **HAVE\_INLINE** is defined.

int **gsl\_poly\_dd\_taylor**(double c[], double xp, const double dd[], const double xa[], size\_t size, double w[])

This function converts the divided-difference representation of a polynomial to a Taylor expansion. The divided-difference representation is supplied in the arrays **dd** and **xa** of length **size**. On output the Taylor coefficients of the polynomial expanded about the point **xp** are stored in the array **c** also of length **size**. A workspace of length **size** must be provided in the array **w**.

int **gsl\_poly\_dd\_hermite\_init**(double dd[], double za[], const double xa[], const double ya[], const double dya[], const size\_t size)

This function computes a divided-difference representation of the interpolating Hermite polynomial for the points  $(x, y)$  stored in the arrays **xa** and **ya** of length **size**. Hermite interpolation constructs polynomials which also match first derivatives  $dy/dx$  which are provided in the array **dya** also of length **size**. The first derivatives can be incorporated into the usual divided-difference algorithm by forming a new dataset  $z = \{x_0, x_0, x_1, x_1, \dots\}$ , which is stored in the array **za** of length  $2 * \mathbf{size}$  on output. On output the divided-differences of the Hermite representation are stored in the array **dd**, also of length  $2 * \mathbf{size}$ . Using the notation above,  $dd[k] = [z_0, z_1, \dots, z_k]$ . The resulting Hermite polynomial can be evaluated by calling `gsl_poly_dd_eval()` and using **za** for the input argument **xa**.

## 6.3 Quadratic Equations

int **gsl\_poly\_solve\_quadratic**(double a, double b, double c, double \*x0, double \*x1)

This function finds the real roots of the quadratic equation,

$$ax^2 + bx + c = 0$$

The number of real roots (either zero, one or two) is returned, and their locations are stored in **x0** and **x1**. If no real roots are found then **x0** and **x1** are not modified. If one real root is found (i.e. if  $a = 0$ ) then it is stored in **x0**. When two real roots are found they are stored in **x0** and **x1** in ascending order. The case of coincident roots is not considered special. For example  $(x - 1)^2 = 0$  will have two roots, which happen to have exactly equal values.

The number of roots found depends on the sign of the discriminant  $b^2 - 4ac$ . This will be subject to rounding and cancellation errors when computed in double precision, and will also be subject to errors if the coefficients of the polynomial are inexact. These errors may cause a discrete change in the number of roots. However, for polynomials with small integer coefficients the discriminant can always be computed exactly.

int **gsl\_poly\_complex\_solve\_quadratic**(double a, double b, double c, gsl\_complex \*z0, gsl\_complex \*z1)  
 This function finds the complex roots of the quadratic equation,

$$az^2 + bz + c = 0$$

The number of complex roots is returned (either one or two) and the locations of the roots are stored in **z0** and **z1**. The roots are returned in ascending order, sorted first by their real components and then by their imaginary components. If only one real root is found (i.e. if  $a = 0$ ) then it is stored in **z0**.

## 6.4 Cubic Equations

int **gsl\_poly\_solve\_cubic**(double a, double b, double c, double \*x0, double \*x1, double \*x2)  
 This function finds the real roots of the cubic equation,

$$x^3 + ax^2 + bx + c = 0$$

with a leading coefficient of unity. The number of real roots (either one or three) is returned, and their locations are stored in **x0**, **x1** and **x2**. If one real root is found then only **x0** is modified. When three real roots are found they are stored in **x0**, **x1** and **x2** in ascending order. The case of coincident roots is not considered special. For example, the equation  $(x - 1)^3 = 0$  will have three roots with exactly equal values. As in the quadratic case, finite precision may cause equal or closely-spaced real roots to move off the real axis into the complex plane, leading to a discrete change in the number of real roots.

int **gsl\_poly\_complex\_solve\_cubic**(double a, double b, double c, gsl\_complex \*z0, gsl\_complex \*z1, gsl\_complex \*z2)  
 This function finds the complex roots of the cubic equation,

$$z^3 + az^2 + bz + c = 0$$

The number of complex roots is returned (always three) and the locations of the roots are stored in **z0**, **z1** and **z2**. The roots are returned in ascending order, sorted first by their real components and then by their imaginary components.

## 6.5 General Polynomial Equations

The roots of polynomial equations cannot be found analytically beyond the special cases of the quadratic, cubic and quartic equation. The algorithm described in this section uses an iterative method to find the approximate locations of roots of higher order polynomials.

type **gsl\_poly\_complex\_workspace**

This workspace contains parameters used for finding roots of general polynomials

*gsl\_poly\_complex\_workspace* \***gsl\_poly\_complex\_workspace\_alloc**(size\_t n)

This function allocates space for a *gsl\_poly\_complex\_workspace* struct and a workspace suitable for solving a polynomial with  $n$  coefficients using the routine *gsl\_poly\_complex\_solve()*.

The function returns a pointer to the newly allocated *gsl\_poly\_complex\_workspace* if no errors were detected, and a null pointer in the case of error.

void **gsl\_poly\_complex\_workspace\_free**(*gsl\_poly\_complex\_workspace* \*w)

This function frees all the memory associated with the workspace  $w$ .

```
int gsl_poly_complex_solve(const double *a, size_t n, gsl_poly_complex_workspace *w,
                          gsl_complex_packed_ptr z)
```

This function computes the roots of the general polynomial

$$P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$$

using balanced-QR reduction of the companion matrix. The parameter  $n$  specifies the length of the coefficient array. The coefficient of the highest order term must be non-zero. The function requires a workspace  $w$  of the appropriate size. The  $n - 1$  roots are returned in the packed complex array  $z$  of length  $2(n - 1)$ , alternating real and imaginary parts.

The function returns `GSL_SUCCESS` if all the roots are found. If the QR reduction does not converge, the error handler is invoked with an error code of `GSL_EFAILED`. Note that due to finite precision, roots of higher multiplicity are returned as a cluster of simple roots with reduced accuracy. The solution of polynomials with higher-order roots requires specialized algorithms that take the multiplicity structure into account (see e.g. Z. Zeng, Algorithm 835, ACM Transactions on Mathematical Software, Volume 30, Issue 2 (2004), pp 218–236).

## 6.6 Examples

To demonstrate the use of the general polynomial solver we will take the polynomial  $P(x) = x^5 - 1$  which has these roots:

$$1, e^{2\pi i/5}, e^{4\pi i/5}, e^{6\pi i/5}, e^{8\pi i/5}$$

The following program will find these roots.

```
#include <stdio.h>
#include <gsl/gsl_poly.h>

int
main (void)
{
    int i;
    /* coefficients of P(x) = -1 + x^5 */
    double a[6] = { -1, 0, 0, 0, 0, 1 };
    double z[10];

    gsl_poly_complex_workspace * w
        = gsl_poly_complex_workspace_alloc (6);

    gsl_poly_complex_solve (a, 6, w, z);

    gsl_poly_complex_workspace_free (w);

    for (i = 0; i < 5; i++)
    {
        printf ("z%d = %.18f %.18f\n",
                i, z[2*i], z[2*i+1]);
    }

    return 0;
}
```

The output of the program is

```
z0 = -0.809016994374947673 +0.587785252292473359i
z1 = -0.809016994374947673 -0.587785252292473359i
z2 = +0.309016994374947507 +0.951056516295152976i
z3 = +0.309016994374947507 -0.951056516295152976i
z4 = +0.999999999999999889 +0.000000000000000000i
```

which agrees with the analytic result,  $z_n = \exp(2\pi ni/5)$ .

## 6.7 References and Further Reading

The balanced-QR method and its error analysis are described in the following papers,

- R.S. Martin, G. Peters and J.H. Wilkinson, “The QR Algorithm for Real Hessenberg Matrices”, *Numerische Mathematik*, 14 (1970), 219–231.
- B.N. Parlett and C. Reinsch, “Balancing a Matrix for Calculation of Eigenvalues and Eigenvectors”, *Numerische Mathematik*, 13 (1969), 293–304.
- A. Edelman and H. Murakami, “Polynomial roots from companion matrix eigenvalues”, *Mathematics of Computation*, Vol.: 64, No.: 210 (1995), 763–776.

The formulas for divided differences are given in the following texts,

- Abramowitz and Stegun, *Handbook of Mathematical Functions*, Sections 25.1.4 and 25.2.26.
- R. L. Burden and J. D. Faires, *Numerical Analysis*, 9th edition, ISBN 0-538-73351-9, 2011.





## SPECIAL FUNCTIONS

This chapter describes the GSL special function library. The library includes routines for calculating the values of Airy functions, Bessel functions, Clausen functions, Coulomb wave functions, Coupling coefficients, the Dawson function, Debye functions, Dilogarithms, Elliptic integrals, Jacobi elliptic functions, Error functions, Exponential integrals, Fermi-Dirac functions, Gamma functions, Gegenbauer functions, Hermite polynomials and functions, Hypergeometric functions, Laguerre functions, Legendre functions and Spherical Harmonics, the Psi (Digamma) Function, Synchrotron functions, Transport functions, Trigonometric functions and Zeta functions. Each routine also computes an estimate of the numerical error in the calculated value of the function.

The functions in this chapter are declared in individual header files, such as `gsl_sf_airy.h`, `gsl_sf_bessel.h`, etc. The complete set of header files can be included using the file `gsl_sf.h`.

### 7.1 Usage

The special functions are available in two calling conventions, a *natural form* which returns the numerical value of the function and an *error-handling form* which returns an error code. The two types of function provide alternative ways of accessing the same underlying code.

The *natural form* returns only the value of the function and can be used directly in mathematical expressions. For example, the following function call will compute the value of the Bessel function  $J_0(x)$ :

```
double y = gsl_sf_bessel_J0 (x);
```

There is no way to access an error code or to estimate the error using this method. To allow access to this information the alternative error-handling form stores the value and error in a modifiable argument:

```
gsl_sf_result result;  
int status = gsl_sf_bessel_J0_e (x, &result);
```

The error-handling functions have the suffix `_e`. The returned status value indicates error conditions such as overflow, underflow or loss of precision. If there are no errors the error-handling functions return `GSL_SUCCESS`.

## 7.2 The `gsl_sf_result` struct

The error handling form of the special functions always calculate an error estimate along with the value of the result. Therefore, structures are provided for amalgamating a value and error estimate. These structures are declared in the header file `gsl_sf_result.h`.

The following struct contains value and error fields.

type `gsl_sf_result`

```
typedef struct
{
    double val;
    double err;
} gsl_sf_result;
```

The field `val` contains the value and the field `err` contains an estimate of the absolute error in the value.

In some cases, an overflow or underflow can be detected and handled by a function. In this case, it may be possible to return a scaling exponent as well as an error/value pair in order to save the result from exceeding the dynamic range of the built-in types. The following struct contains value and error fields as well as an exponent field such that the actual result is obtained as `result * 10(e10)`.

type `gsl_sf_result_e10`

```
typedef struct
{
    double val;
    double err;
    int    e10;
} gsl_sf_result_e10;
```

## 7.3 Modes

The goal of the library is to achieve double precision accuracy wherever possible. However the cost of evaluating some special functions to double precision can be significant, particularly where very high order terms are required. In these cases a `mode` argument, of type `gsl_mode_t` allows the accuracy of the function to be reduced in order to improve performance. The following precision levels are available for the mode argument,

type `gsl_mode_t`

**GSL\_PREC\_DOUBLE**

Double-precision, a relative accuracy of approximately  $2 * 10^{-16}$ .

**GSL\_PREC\_SINGLE**

Single-precision, a relative accuracy of approximately  $10^{-7}$ .

**GSL\_PREC\_APPROX**

Approximate values, a relative accuracy of approximately  $5 * 10^{-4}$ .

The approximate mode provides the fastest evaluation at the lowest accuracy.

## 7.4 Airy Functions and Derivatives

The Airy functions  $Ai(x)$  and  $Bi(x)$  are defined by the integral representations,

$$Ai(x) = \frac{1}{\pi} \int_0^{\infty} \cos(t^3/3 + xt) dt$$

$$Bi(x) = \frac{1}{\pi} \int_0^{\infty} (e^{-t^3/3+xt} + \sin(t^3/3 + xt)) dt$$

For further information see Abramowitz & Stegun, Section 10.4. The Airy functions are defined in the header file `gsl_sf_airy.h`.

### 7.4.1 Airy Functions

double `gsl_sf_airy_Ai`(double x, *gsl\_mode\_t* mode)

int `gsl_sf_airy_Ai_e`(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the Airy function  $Ai(x)$  with an accuracy specified by *mode*.

double `gsl_sf_airy_Bi`(double x, *gsl\_mode\_t* mode)

int `gsl_sf_airy_Bi_e`(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the Airy function  $Bi(x)$  with an accuracy specified by *mode*.

double `gsl_sf_airy_Ai_scaled`(double x, *gsl\_mode\_t* mode)

int `gsl_sf_airy_Ai_scaled_e`(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute a scaled version of the Airy function  $S_A(x)Ai(x)$ . For  $x > 0$  the scaling factor  $S_A(x)$  is  $\exp(+ (2/3)x^{3/2})$ , and is 1 for  $x < 0$ .

double `gsl_sf_airy_Bi_scaled`(double x, *gsl\_mode\_t* mode)

int `gsl_sf_airy_Bi_scaled_e`(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute a scaled version of the Airy function  $S_B(x)Bi(x)$ . For  $x > 0$  the scaling factor  $S_B(x)$  is  $\exp(- (2/3)x^{3/2})$ , and is 1 for  $x < 0$ .

### 7.4.2 Derivatives of Airy Functions

double `gsl_sf_airy_Ai_deriv`(double x, *gsl\_mode\_t* mode)

int `gsl_sf_airy_Ai_deriv_e`(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the Airy function derivative  $Ai'(x)$  with an accuracy specified by *mode*.

double `gsl_sf_airy_Bi_deriv`(double x, *gsl\_mode\_t* mode)

int `gsl_sf_airy_Bi_deriv_e`(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the Airy function derivative  $Bi'(x)$  with an accuracy specified by *mode*.

double `gsl_sf_airy_Ai_deriv_scaled`(double x, *gsl\_mode\_t* mode)

int `gsl_sf_airy_Ai_deriv_scaled_e`(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the scaled Airy function derivative  $S_A(x)Ai'(x)$ . For  $x > 0$  the scaling factor  $S_A(x)$  is  $\exp(+ (2/3)x^{3/2})$ , and is 1 for  $x < 0$ .

double **gsl\_sf\_airy\_Bi\_deriv\_scaled**(double x, *gsl\_mode\_t* mode)

int **gsl\_sf\_airy\_Bi\_deriv\_scaled\_e**(double x, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the scaled Airy function derivative  $S_B(x)Bi'(x)$ . For  $x > 0$  the scaling factor  $S_B(x)$  is  $\exp(-(2/3)x^{3/2})$ , and is 1 for  $x < 0$ .

### 7.4.3 Zeros of Airy Functions

double **gsl\_sf\_airy\_zero\_Ai**(unsigned int s)

int **gsl\_sf\_airy\_zero\_Ai\_e**(unsigned int s, *gsl\_sf\_result* \*result)

These routines compute the location of the  $s$ -th zero of the Airy function  $Ai(x)$ .

double **gsl\_sf\_airy\_zero\_Bi**(unsigned int s)

int **gsl\_sf\_airy\_zero\_Bi\_e**(unsigned int s, *gsl\_sf\_result* \*result)

These routines compute the location of the  $s$ -th zero of the Airy function  $Bi(x)$ .

### 7.4.4 Zeros of Derivatives of Airy Functions

double **gsl\_sf\_airy\_zero\_Ai\_deriv**(unsigned int s)

int **gsl\_sf\_airy\_zero\_Ai\_deriv\_e**(unsigned int s, *gsl\_sf\_result* \*result)

These routines compute the location of the  $s$ -th zero of the Airy function derivative  $Ai'(x)$ .

double **gsl\_sf\_airy\_zero\_Bi\_deriv**(unsigned int s)

int **gsl\_sf\_airy\_zero\_Bi\_deriv\_e**(unsigned int s, *gsl\_sf\_result* \*result)

These routines compute the location of the  $s$ -th zero of the Airy function derivative  $Bi'(x)$ .

## 7.5 Bessel Functions

The routines described in this section compute the Cylindrical Bessel functions  $J_n(x)$ ,  $Y_n(x)$ , Modified cylindrical Bessel functions  $I_n(x)$ ,  $K_n(x)$ , Spherical Bessel functions  $j_l(x)$ ,  $y_l(x)$ , and Modified Spherical Bessel functions  $i_l(x)$ ,  $k_l(x)$ . For more information see Abramowitz & Stegun, Chapters 9 and 10. The Bessel functions are defined in the header file `gsl_sf_bessel.h`.

### 7.5.1 Regular Cylindrical Bessel Functions

double **gsl\_sf\_bessel\_J0**(double x)

int **gsl\_sf\_bessel\_J0\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regular cylindrical Bessel function of zeroth order,  $J_0(x)$ .

double **gsl\_sf\_bessel\_J1**(double x)

int **gsl\_sf\_bessel\_J1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regular cylindrical Bessel function of first order,  $J_1(x)$ .

double **gsl\_sf\_bessel\_Jn**(int n, double x)

int **gsl\_sf\_bessel\_Jn\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the regular cylindrical Bessel function of order  $n$ ,  $J_n(x)$ .

int **gsl\_sf\_bessel\_Jn\_array**(int nmin, int nmax, double x, double result\_array[])

This routine computes the values of the regular cylindrical Bessel functions  $J_n(x)$  for  $n$  from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

## 7.5.2 Irregular Cylindrical Bessel Functions

double **gsl\_sf\_bessel\_Y0**(double x)

int **gsl\_sf\_bessel\_Y0\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the irregular cylindrical Bessel function of zeroth order,  $Y_0(x)$ , for  $x > 0$ .

double **gsl\_sf\_bessel\_Y1**(double x)

int **gsl\_sf\_bessel\_Y1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the irregular cylindrical Bessel function of first order,  $Y_1(x)$ , for  $x > 0$ .

double **gsl\_sf\_bessel\_Yn**(int n, double x)

int **gsl\_sf\_bessel\_Yn\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the irregular cylindrical Bessel function of order  $n$ ,  $Y_n(x)$ , for  $x > 0$ .

int **gsl\_sf\_bessel\_Yn\_array**(int nmin, int nmax, double x, double result\_array[])

This routine computes the values of the irregular cylindrical Bessel functions  $Y_n(x)$  for  $n$  from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*. The domain of the function is  $x > 0$ . The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

## 7.5.3 Regular Modified Cylindrical Bessel Functions

double **gsl\_sf\_bessel\_I0**(double x)

int **gsl\_sf\_bessel\_I0\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regular modified cylindrical Bessel function of zeroth order,  $I_0(x)$ .

double **gsl\_sf\_bessel\_I1**(double x)

int **gsl\_sf\_bessel\_I1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regular modified cylindrical Bessel function of first order,  $I_1(x)$ .

double **gsl\_sf\_bessel\_In**(int n, double x)

int **gsl\_sf\_bessel\_In\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the regular modified cylindrical Bessel function of order  $n$ ,  $I_n(x)$ .

int **gsl\_sf\_bessel\_In\_array**(int nmin, int nmax, double x, double result\_array[])

This routine computes the values of the regular modified cylindrical Bessel functions  $I_n(x)$  for  $n$  from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*. The start of the range *nmin* must be positive or zero. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

double **gsl\_sf\_bessel\_I0\_scaled**(double x)

int **gsl\_sf\_bessel\_I0\_scaled\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified cylindrical Bessel function of zeroth order  $\exp(-|x|)I_0(x)$ .

double **gsl\_sf\_bessel\_I1\_scaled**(double x)

int **gsl\_sf\_bessel\_I1\_scaled\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified cylindrical Bessel function of first order  $\exp(-|x|)I_1(x)$ .

double **gsl\_sf\_bessel\_In\_scaled**(int n, double x)

int **gsl\_sf\_bessel\_In\_scaled\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified cylindrical Bessel function of order  $n$ ,  $\exp(-|x|)I_n(x)$

int **gsl\_sf\_bessel\_In\_scaled\_array**(int nmin, int nmax, double x, double result\_array[])

This routine computes the values of the scaled regular cylindrical Bessel functions  $\exp(-|x|)I_n(x)$  for  $n$  from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*. The start of the range *nmin* must be positive or zero. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

## 7.5.4 Irregular Modified Cylindrical Bessel Functions

double **gsl\_sf\_bessel\_K0**(double x)

int **gsl\_sf\_bessel\_K0\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the irregular modified cylindrical Bessel function of zeroth order,  $K_0(x)$ , for  $x > 0$ .

double **gsl\_sf\_bessel\_K1**(double x)

int **gsl\_sf\_bessel\_K1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the irregular modified cylindrical Bessel function of first order,  $K_1(x)$ , for  $x > 0$ .

double **gsl\_sf\_bessel\_Kn**(int n, double x)

int **gsl\_sf\_bessel\_Kn\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the irregular modified cylindrical Bessel function of order  $n$ ,  $K_n(x)$ , for  $x > 0$ .

int **gsl\_sf\_bessel\_Kn\_array**(int nmin, int nmax, double x, double result\_array[])

This routine computes the values of the irregular modified cylindrical Bessel functions  $K_n(x)$  for  $n$  from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*. The start of the range *nmin* must be positive or zero. The domain of the function is  $x > 0$ . The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

double **gsl\_sf\_bessel\_K0\_scaled**(double x)

int **gsl\_sf\_bessel\_K0\_scaled\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the scaled irregular modified cylindrical Bessel function of zeroth order  $\exp(x)K_0(x)$  for  $x > 0$ .

double **gsl\_sf\_bessel\_K1\_scaled**(double x)

int **gsl\_sf\_bessel\_K1\_scaled\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the scaled irregular modified cylindrical Bessel function of first order  $\exp(x)K_1(x)$  for  $x > 0$ .

double **gsl\_sf\_bessel\_Kn\_scaled**(int n, double x)

int **gsl\_sf\_bessel\_Kn\_scaled\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the scaled irregular modified cylindrical Bessel function of order  $n$ ,  $\exp(x)K_n(x)$ , for  $x > 0$ .

int **gsl\_sf\_bessel\_Kn\_scaled\_array**(int nmin, int nmax, double x, double result\_array[])

This routine computes the values of the scaled irregular cylindrical Bessel functions  $\exp(x)K_n(x)$  for  $n$  from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*. The start of the range *nmin* must be positive or zero. The domain of the function is  $x > 0$ . The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

## 7.5.5 Regular Spherical Bessel Functions

double **gsl\_sf\_bessel\_j0**(double x)

int **gsl\_sf\_bessel\_j0\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regular spherical Bessel function of zeroth order,  $j_0(x) = \sin(x)/x$ .

double **gsl\_sf\_bessel\_j1**(double x)

int **gsl\_sf\_bessel\_j1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regular spherical Bessel function of first order,  $j_1(x) = (\sin(x)/x - \cos(x))/x$ .

double **gsl\_sf\_bessel\_j2**(double x)

int **gsl\_sf\_bessel\_j2\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regular spherical Bessel function of second order,  $j_2(x) = ((3/x^2 - 1)\sin(x) - 3\cos(x)/x)/x$ .

double **gsl\_sf\_bessel\_jl**(int l, double x)

int **gsl\_sf\_bessel\_jl\_e**(int l, double x, *gsl\_sf\_result* \*result)

These routines compute the regular spherical Bessel function of order  $l$ ,  $j_l(x)$ , for  $l \geq 0$  and  $x \geq 0$ .

int **gsl\_sf\_bessel\_jl\_array**(int lmax, double x, double result\_array[])

This routine computes the values of the regular spherical Bessel functions  $j_l(x)$  for  $l$  from 0 to *lmax* inclusive for  $lmax \geq 0$  and  $x \geq 0$ , storing the results in the array *result\_array*. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

int **gsl\_sf\_bessel\_jl\_steed\_array**(int lmax, double x, double \*result\_array)

This routine uses Steed's method to compute the values of the regular spherical Bessel functions  $j_l(x)$  for  $l$  from 0 to *lmax* inclusive for  $lmax \geq 0$  and  $x \geq 0$ , storing the results in the array *result\_array*. The Steed/Barnett algorithm is described in Comp. Phys. Comm. 21, 297 (1981). Steed's method is more stable than the recurrence used in the other functions but is also slower.

## 7.5.6 Irregular Spherical Bessel Functions

double **gsl\_sf\_bessel\_y0**(double x)

int **gsl\_sf\_bessel\_y0\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the irregular spherical Bessel function of zeroth order,  $y_0(x) = -\cos(x)/x$ .

double **gsl\_sf\_bessel\_y1**(double x)

int **gsl\_sf\_bessel\_y1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the irregular spherical Bessel function of first order,  $y_1(x) = -(\cos(x)/x + \sin(x))/x$ .

double **gsl\_sf\_bessel\_y2**(double x)

int **gsl\_sf\_bessel\_y2\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the irregular spherical Bessel function of second order,  $y_2(x) = (-3/x^3 + 1/x)\cos(x) - (3/x^2)\sin(x)$ .

double **gsl\_sf\_bessel\_y1**(int l, double x)

int **gsl\_sf\_bessel\_y1\_e**(int l, double x, *gsl\_sf\_result* \*result)

These routines compute the irregular spherical Bessel function of order  $l$ ,  $y_l(x)$ , for  $l \geq 0$ .

int **gsl\_sf\_bessel\_y1\_array**(int lmax, double x, double result\_array[])

This routine computes the values of the irregular spherical Bessel functions  $y_l(x)$  for  $l$  from 0 to  $lmax$  inclusive for  $lmax \geq 0$ , storing the results in the array *result\_array*. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

## 7.5.7 Regular Modified Spherical Bessel Functions

The regular modified spherical Bessel functions  $i_l(x)$  are related to the modified Bessel functions of fractional order,  $i_l(x) = \sqrt{\pi/(2x)}I_{l+1/2}(x)$

double **gsl\_sf\_bessel\_i0\_scaled**(double x)

int **gsl\_sf\_bessel\_i0\_scaled\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified spherical Bessel function of zeroth order,  $\exp(-|x|)i_0(x)$ .

double **gsl\_sf\_bessel\_i1\_scaled**(double x)

int **gsl\_sf\_bessel\_i1\_scaled\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified spherical Bessel function of first order,  $\exp(-|x|)i_1(x)$ .

double **gsl\_sf\_bessel\_i2\_scaled**(double x)

int **gsl\_sf\_bessel\_i2\_scaled\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified spherical Bessel function of second order,  $\exp(-|x|)i_2(x)$

double **gsl\_sf\_bessel\_il\_scaled**(int l, double x)

int **gsl\_sf\_bessel\_il\_scaled\_e**(int l, double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified spherical Bessel function of order  $l$ ,  $\exp(-|x|)i_l(x)$



int **gsl\_sf\_bessel\_il\_scaled\_array**(int *lmax*, double *x*, double *result\_array*[])

This routine computes the values of the scaled regular modified spherical Bessel functions  $\exp(-|x|)i_l(x)$  for  $l$  from 0 to *lmax* inclusive for  $lmax \geq 0$ , storing the results in the array *result\_array*. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

## 7.5.8 Irregular Modified Spherical Bessel Functions

The irregular modified spherical Bessel functions  $k_l(x)$  are related to the irregular modified Bessel functions of fractional order,  $k_l(x) = \sqrt{\pi/(2x)}K_{l+1/2}(x)$ .

double **gsl\_sf\_bessel\_k0\_scaled**(double *x*)

int **gsl\_sf\_bessel\_k0\_scaled\_e**(double *x*, *gsl\_sf\_result* \**result*)

These routines compute the scaled irregular modified spherical Bessel function of zeroth order,  $\exp(x)k_0(x)$ , for  $x > 0$ .

double **gsl\_sf\_bessel\_k1\_scaled**(double *x*)

int **gsl\_sf\_bessel\_k1\_scaled\_e**(double *x*, *gsl\_sf\_result* \**result*)

These routines compute the scaled irregular modified spherical Bessel function of first order,  $\exp(x)k_1(x)$ , for  $x > 0$ .

double **gsl\_sf\_bessel\_k2\_scaled**(double *x*)

int **gsl\_sf\_bessel\_k2\_scaled\_e**(double *x*, *gsl\_sf\_result* \**result*)

These routines compute the scaled irregular modified spherical Bessel function of second order,  $\exp(x)k_2(x)$ , for  $x > 0$ .

double **gsl\_sf\_bessel\_kl\_scaled**(int *l*, double *x*)

int **gsl\_sf\_bessel\_kl\_scaled\_e**(int *l*, double *x*, *gsl\_sf\_result* \**result*)

These routines compute the scaled irregular modified spherical Bessel function of order *l*,  $\exp(x)k_l(x)$ , for  $x > 0$ .

int **gsl\_sf\_bessel\_kl\_scaled\_array**(int *lmax*, double *x*, double *result\_array*[])

This routine computes the values of the scaled irregular modified spherical Bessel functions  $\exp(x)k_l(x)$  for  $l$  from 0 to *lmax* inclusive for  $lmax \geq 0$  and  $x > 0$ , storing the results in the array *result\_array*. The values are computed using recurrence relations for efficiency, and therefore may differ slightly from the exact values.

## 7.5.9 Regular Bessel Function—Fractional Order

double **gsl\_sf\_bessel\_Jnu**(double *nu*, double *x*)

int **gsl\_sf\_bessel\_Jnu\_e**(double *nu*, double *x*, *gsl\_sf\_result* \**result*)

These routines compute the regular cylindrical Bessel function of fractional order  $\nu$ ,  $J_\nu(x)$ .

int **gsl\_sf\_bessel\_sequence\_Jnu\_e**(double *nu*, *gsl\_mode\_t* *mode*, *size\_t* *size*, double *v*[])

This function computes the regular cylindrical Bessel function of fractional order  $\nu$ ,  $J_\nu(x)$ , evaluated at a series of  $x$  values. The array *v* of length *size* contains the  $x$  values. They are assumed to be strictly ordered and positive. The array is over-written with the values of  $J_\nu(x_i)$ .

### 7.5.10 Irregular Bessel Functions—Fractional Order

double **gsl\_sf\_bessel\_Ynu**(double nu, double x)

int **gsl\_sf\_bessel\_Ynu\_e**(double nu, double x, *gsl\_sf\_result* \*result)

These routines compute the irregular cylindrical Bessel function of fractional order  $\nu$ ,  $Y_\nu(x)$ .

### 7.5.11 Regular Modified Bessel Functions—Fractional Order

double **gsl\_sf\_bessel\_Inu**(double nu, double x)

int **gsl\_sf\_bessel\_Inu\_e**(double nu, double x, *gsl\_sf\_result* \*result)

These routines compute the regular modified Bessel function of fractional order  $\nu$ ,  $I_\nu(x)$  for  $x > 0$ ,  $\nu > 0$ .

double **gsl\_sf\_bessel\_Inu\_scaled**(double nu, double x)

int **gsl\_sf\_bessel\_Inu\_scaled\_e**(double nu, double x, *gsl\_sf\_result* \*result)

These routines compute the scaled regular modified Bessel function of fractional order  $\nu$ ,  $\exp(-|x|)I_\nu(x)$  for  $x > 0$ ,  $\nu > 0$ .

### 7.5.12 Irregular Modified Bessel Functions—Fractional Order

double **gsl\_sf\_bessel\_Knu**(double nu, double x)

int **gsl\_sf\_bessel\_Knu\_e**(double nu, double x, *gsl\_sf\_result* \*result)

These routines compute the irregular modified Bessel function of fractional order  $\nu$ ,  $K_\nu(x)$  for  $x > 0$ ,  $\nu > 0$ .

double **gsl\_sf\_bessel\_lnKnu**(double nu, double x)

int **gsl\_sf\_bessel\_lnKnu\_e**(double nu, double x, *gsl\_sf\_result* \*result)

These routines compute the logarithm of the irregular modified Bessel function of fractional order  $\nu$ ,  $\ln(K_\nu(x))$  for  $x > 0$ ,  $\nu > 0$ .

double **gsl\_sf\_bessel\_Knu\_scaled**(double nu, double x)

int **gsl\_sf\_bessel\_Knu\_scaled\_e**(double nu, double x, *gsl\_sf\_result* \*result)

These routines compute the scaled irregular modified Bessel function of fractional order  $\nu$ ,  $\exp(+|x|)K_\nu(x)$  for  $x > 0$ ,  $\nu > 0$ .

### 7.5.13 Zeros of Regular Bessel Functions

double **gsl\_sf\_bessel\_zero\_J0**(unsigned int s)

int **gsl\_sf\_bessel\_zero\_J0\_e**(unsigned int s, *gsl\_sf\_result* \*result)

These routines compute the location of the  $s$ -th positive zero of the Bessel function  $J_0(x)$ .

double **gsl\_sf\_bessel\_zero\_J1**(unsigned int s)

int **gsl\_sf\_bessel\_zero\_J1\_e**(unsigned int s, *gsl\_sf\_result* \*result)

These routines compute the location of the  $s$ -th positive zero of the Bessel function  $J_1(x)$ .

double **gsl\_sf\_bessel\_zero\_Jnu**(double nu, unsigned int s)

int **gsl\_sf\_bessel\_zero\_Jnu\_e**(double nu, unsigned int s, *gsl\_sf\_result* \*result)

These routines compute the location of the  $s$ -th positive zero of the Bessel function  $J_\nu(x)$ . The current implementation does not support negative values of  $nu$ .

## 7.6 Clausen Functions

The Clausen function is defined by the following integral,

$$Cl_2(x) = - \int_0^x dt \log(2 \sin(t/2))$$

It is related to the *dilogarithm* by  $Cl_2(\theta) = \Im Li_2(\exp(i\theta))$ . The Clausen functions are declared in the header file `gsl_sf_clausen.h`.

double **gsl\_sf\_clausen**(double x)

int **gsl\_sf\_clausen\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Clausen integral  $Cl_2(x)$ .

## 7.7 Coulomb Functions

The prototypes of the Coulomb functions are declared in the header file `gsl_sf_coulomb.h`. Both bound state and scattering solutions are available.

### 7.7.1 Normalized Hydrogenic Bound States

double **gsl\_sf\_hydrogenicR\_1**(double Z, double r)

int **gsl\_sf\_hydrogenicR\_1\_e**(double Z, double r, *gsl\_sf\_result* \*result)

These routines compute the lowest-order normalized hydrogenic bound state radial wavefunction  $R_1 := 2Z\sqrt{Z} \exp(-Zr)$ .

double **gsl\_sf\_hydrogenicR**(int n, int l, double Z, double r)

int **gsl\_sf\_hydrogenicR\_e**(int n, int l, double Z, double r, *gsl\_sf\_result* \*result)

These routines compute the  $n$ -th normalized hydrogenic bound state radial wavefunction,

$$R_n := \frac{2Z^{3/2}}{n^2} \left( \frac{2Zr}{n} \right)^l \sqrt{\frac{(n-l-1)!}{(n+l)!}} \exp(-Zr/n) L_{n-l-1}^{2l+1}(2Zr/n).$$

where  $L_b^a(x)$  is the *generalized Laguerre polynomial*. The normalization is chosen such that the wavefunction  $\psi$  is given by  $\psi(n, l, r) = R_n Y_{lm}$ .

## 7.7.2 Coulomb Wave Functions

The Coulomb wave functions  $F_L(\eta, x)$ ,  $G_L(\eta, x)$  are described in Abramowitz & Stegun, Chapter 14. Because there can be a large dynamic range of values for these functions, overflows are handled gracefully. If an overflow occurs, `GSL_EOVRFLW` is signalled and exponent(s) are returned through the modifiable parameters `exp_F`, `exp_G`. The full solution can be reconstructed from the following relations,

$$F_L(\eta, x) = fc[k_L] * \exp(exp_F)$$

$$G_L(\eta, x) = gc[k_L] * \exp(exp_G)$$

$$F'_L(\eta, x) = fcp[k_L] * \exp(exp_F)$$

$$G'_L(\eta, x) = gcp[k_L] * \exp(exp_G)$$

int `gsl_sf_coulomb_wave_FG_e`(double eta, double x, double L\_F, int k, *gsl\_sf\_result* \*F, *gsl\_sf\_result* \*Fp, *gsl\_sf\_result* \*G, *gsl\_sf\_result* \*Gp, double \*exp\_F, double \*exp\_G)

This function computes the Coulomb wave functions  $F_L(\eta, x)$ ,  $G_{L-k}(\eta, x)$  and their derivatives  $F'_L(\eta, x)$ ,  $G'_{L-k}(\eta, x)$  with respect to  $x$ . The parameters are restricted to  $L, L - k > -1/2, x > 0$  and integer  $k$ . Note that  $L$  itself is not restricted to being an integer. The results are stored in the parameters `F`, `G` for the function values and `Fp`, `Gp` for the derivative values. If an overflow occurs, `GSL_EOVRFLW` is returned and scaling exponents are stored in the modifiable parameters `exp_F`, `exp_G`.

int `gsl_sf_coulomb_wave_F_array`(double L\_min, int kmax, double eta, double x, double fc\_array[], double \*F\_exponent)

This function computes the Coulomb wave function  $F_L(\eta, x)$  for  $L = L_{min} \dots L_{min} + k_{max}$ , storing the results in `fc_array`. In the case of overflow the exponent is stored in `F_exponent`.

int `gsl_sf_coulomb_wave_FG_array`(double L\_min, int kmax, double eta, double x, double fc\_array[], double gc\_array[], double \*F\_exponent, double \*G\_exponent)

This function computes the functions  $F_L(\eta, x)$ ,  $G_L(\eta, x)$  for  $L = L_{min} \dots L_{min} + k_{max}$  storing the results in `fc_array` and `gc_array`. In the case of overflow the exponents are stored in `F_exponent` and `G_exponent`.

int `gsl_sf_coulomb_wave_FGp_array`(double L\_min, int kmax, double eta, double x, double fc\_array[], double fcp\_array[], double gc\_array[], double gcp\_array[], double \*F\_exponent, double \*G\_exponent)

This function computes the functions  $F_L(\eta, x)$ ,  $G_L(\eta, x)$  and their derivatives  $F'_L(\eta, x)$ ,  $G'_L(\eta, x)$  for  $L = L_{min} \dots L_{min} + k_{max}$  storing the results in `fc_array`, `gc_array`, `fcp_array` and `gcp_array`. In the case of overflow the exponents are stored in `F_exponent` and `G_exponent`.

int `gsl_sf_coulomb_wave_sphF_array`(double L\_min, int kmax, double eta, double x, double fc\_array[], double F\_exponent[])

This function computes the Coulomb wave function divided by the argument  $F_L(\eta, x)/x$  for  $L = L_{min} \dots L_{min} + k_{max}$ , storing the results in `fc_array`. In the case of overflow the exponent is stored in `F_exponent`. This function reduces to spherical Bessel functions in the limit  $\eta \rightarrow 0$ .

## 7.7.3 Coulomb Wave Function Normalization Constant

The Coulomb wave function normalization constant is defined in Abramowitz 14.1.7.

int `gsl_sf_coulomb_CL_e`(double L, double eta, *gsl\_sf\_result* \*result)

This function computes the Coulomb wave function normalization constant  $C_L(\eta)$  for  $L > -1$ .

int `gsl_sf_coulomb_CL_array`(double Lmin, int kmax, double eta, double cl[])

This function computes the Coulomb wave function normalization constant  $C_L(\eta)$  for  $L = L_{min} \dots L_{min} + k_{max}$ ,  $L_{min} > -1$ .

## 7.8 Coupling Coefficients

The Wigner 3-j, 6-j and 9-j symbols give the coupling coefficients for combined angular momentum vectors. Since the arguments of the standard coupling coefficient functions are integer or half-integer, the arguments of the following functions are, by convention, integers equal to twice the actual spin value. For information on the 3-j coefficients see Abramowitz & Stegun, Section 27.9. The functions described in this section are declared in the header file `gsl_sf_coupling.h`.

### 7.8.1 3-j Symbols

double `gsl_sf_coupling_3j`(int two\_ja, int two\_jb, int two\_jc, int two\_ma, int two\_mb, int two\_mc)

int `gsl_sf_coupling_3j_e`(int two\_ja, int two\_jb, int two\_jc, int two\_ma, int two\_mb, int two\_mc, *gsl\_sf\_result* \*result)

These routines compute the Wigner 3-j coefficient,

$$\begin{pmatrix} ja & jb & jc \\ ma & mb & mc \end{pmatrix}$$

where the arguments are given in half-integer units,  $ja = \text{two\_ja}/2$ ,  $ma = \text{two\_ma}/2$ , etc.

### 7.8.2 6-j Symbols

double `gsl_sf_coupling_6j`(int two\_ja, int two\_jb, int two\_jc, int two\_jd, int two\_je, int two\_jf)

int `gsl_sf_coupling_6j_e`(int two\_ja, int two\_jb, int two\_jc, int two\_jd, int two\_je, int two\_jf, *gsl\_sf\_result* \*result)

These routines compute the Wigner 6-j coefficient,

$$\left\{ \begin{matrix} ja & jb & jc \\ jd & je & jf \end{matrix} \right\}$$

where the arguments are given in half-integer units,  $ja = \text{two\_ja}/2$ ,  $ma = \text{two\_ma}/2$ , etc.

### 7.8.3 9-j Symbols

double `gsl_sf_coupling_9j`(int two\_ja, int two\_jb, int two\_jc, int two\_jd, int two\_je, int two\_jf, int two\_jg, int two\_jh, int two\_ji)

int `gsl_sf_coupling_9j_e`(int two\_ja, int two\_jb, int two\_jc, int two\_jd, int two\_je, int two\_jf, int two\_jg, int two\_jh, int two\_ji, *gsl\_sf\_result* \*result)

These routines compute the Wigner 9-j coefficient,

$$\left\{ \begin{matrix} ja & jb & jc \\ jd & je & jf \\ jg & jh & ji \end{matrix} \right\}$$

where the arguments are given in half-integer units,  $ja = \text{two\_ja}/2$ ,  $ma = \text{two\_ma}/2$ , etc.

## 7.9 Dawson Function

The Dawson integral is defined by

$$\exp(-x^2) \int_0^x dt \exp(t^2)$$

A table of Dawson's integral can be found in Abramowitz & Stegun, Table 7.5. The Dawson functions are declared in the header file `gsl_sf_dawson.h`.

double **gsl\_sf\_dawson**(double x)

int **gsl\_sf\_dawson\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the value of Dawson's integral for  $x$ .

## 7.10 Debye Functions

The Debye functions  $D_n(x)$  are defined by the following integral,

$$D_n(x) = \frac{n}{x^n} \int_0^x dt \frac{t^n}{e^t - 1}$$

For further information see Abramowitz & Stegun, Section 27.1. The Debye functions are declared in the header file `gsl_sf_debye.h`.

double **gsl\_sf\_debye\_1**(double x)

int **gsl\_sf\_debye\_1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the first-order Debye function  $D_1(x)$ .

double **gsl\_sf\_debye\_2**(double x)

int **gsl\_sf\_debye\_2\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the second-order Debye function  $D_2(x)$ .

double **gsl\_sf\_debye\_3**(double x)

int **gsl\_sf\_debye\_3\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the third-order Debye function  $D_3(x)$ .

double **gsl\_sf\_debye\_4**(double x)

int **gsl\_sf\_debye\_4\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the fourth-order Debye function  $D_4(x)$ .

double **gsl\_sf\_debye\_5**(double x)

int **gsl\_sf\_debye\_5\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the fifth-order Debye function  $D_5(x)$ .

double **gsl\_sf\_debye\_6**(double x)

int **gsl\_sf\_debye\_6\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the sixth-order Debye function  $D_6(x)$ .

## 7.11 Dilogarithm

The dilogarithm is defined as

$$Li_2(z) = - \int_0^z ds \frac{\log(1-s)}{s}$$

The functions described in this section are declared in the header file `gsl_sf_dilog.h`.

### 7.11.1 Real Argument

double `gsl_sf_dilog`(double *x*)

int `gsl_sf_dilog_e`(double *x*, *gsl\_sf\_result* \**result*)

These routines compute the dilogarithm for a real argument. In Lewin's notation this is  $Li_2(x)$ , the real part of the dilogarithm of a real  $x$ . It is defined by the integral representation

$$Li_2(x) = -\Re \int_0^x ds \log(1-s)/s$$

Note that  $\Im(Li_2(x)) = 0$  for  $x \leq 1$ , and  $-\pi \log(x)$  for  $x > 1$ .

Note that Abramowitz & Stegun refer to the Spence integral  $S(x) = Li_2(1-x)$  as the dilogarithm rather than  $Li_2(x)$ .

### 7.11.2 Complex Argument

int `gsl_sf_complex_dilog_e`(double *r*, double *theta*, *gsl\_sf\_result* \**result\_re*, *gsl\_sf\_result* \**result\_im*)

This function computes the full complex-valued dilogarithm for the complex argument  $z = r \exp(i\theta)$ . The real and imaginary parts of the result are returned in *result\_re*, *result\_im*.

## 7.12 Elementary Operations

The following functions allow for the propagation of errors when combining quantities by multiplication. The functions are declared in the header file `gsl_sf_elementary.h`.

double `gsl_sf_multiply`(double *x*, double *y*)

int `gsl_sf_multiply_e`(double *x*, double *y*, *gsl\_sf\_result* \**result*)

This function multiplies *x* and *y* storing the product and its associated error in *result*.

int `gsl_sf_multiply_err_e`(double *x*, double *dx*, double *y*, double *dy*, *gsl\_sf\_result* \**result*)

This function multiplies *x* and *y* with associated absolute errors *dx* and *dy*. The product  $xy \pm xy\sqrt{(dx/x)^2 + (dy/y)^2}$  is stored in *result*.

## 7.13 Elliptic Integrals

The functions described in this section are declared in the header file `gsl_sf_ellint.h`. Further information about the elliptic integrals can be found in Abramowitz & Stegun, Chapter 17.

### 7.13.1 Definition of Legendre Forms

The Legendre forms of elliptic integrals  $F(\phi, k)$ ,  $E(\phi, k)$  and  $\Pi(\phi, k, n)$  are defined by,

$$F(\phi, k) = \int_0^\phi dt \frac{1}{\sqrt{1 - k^2 \sin^2(t)}}$$

$$E(\phi, k) = \int_0^\phi dt \sqrt{1 - k^2 \sin^2(t)}$$

$$\Pi(\phi, k, n) = \int_0^\phi dt \frac{1}{(1 + n \sin^2(t)) \sqrt{1 - k^2 \sin^2(t)}}$$

The complete Legendre forms are denoted by  $K(k) = F(\pi/2, k)$  and  $E(k) = E(\pi/2, k)$ .

The notation used here is based on Carlson, “Numerische Mathematik” 33 (1979) 1 and differs slightly from that used by Abramowitz & Stegun, where the functions are given in terms of the parameter  $m = k^2$  and  $n$  is replaced by  $-n$ .

### 7.13.2 Definition of Carlson Forms

The Carlson symmetric forms of elliptical integrals  $RC(x, y)$ ,  $RD(x, y, z)$ ,  $RF(x, y, z)$  and  $RJ(x, y, z, p)$  are defined by,

$$RC(x, y) = 1/2 \int_0^\infty dt (t+x)^{-1/2} (t+y)^{-1}$$

$$RD(x, y, z) = 3/2 \int_0^\infty dt (t+x)^{-1/2} (t+y)^{-1/2} (t+z)^{-3/2}$$

$$RF(x, y, z) = 1/2 \int_0^\infty dt (t+x)^{-1/2} (t+y)^{-1/2} (t+z)^{-1/2}$$

$$RJ(x, y, z, p) = 3/2 \int_0^\infty dt (t+x)^{-1/2} (t+y)^{-1/2} (t+z)^{-1/2} (t+p)^{-1}$$

### 7.13.3 Legendre Form of Complete Elliptic Integrals

double `gsl_sf_ellint_Kcomp`(double `k`, `gsl_mode_t` `mode`)

int `gsl_sf_ellint_Kcomp_e`(double `k`, `gsl_mode_t` `mode`, `gsl_sf_result` \*`result`)

These routines compute the complete elliptic integral  $K(k)$  to the accuracy specified by the mode variable `mode`. Note that Abramowitz & Stegun define this function in terms of the parameter  $m = k^2$ .

double `gsl_sf_ellint_Ecomp`(double `k`, `gsl_mode_t` `mode`)

int `gsl_sf_ellint_Ecomp_e`(double `k`, `gsl_mode_t` `mode`, `gsl_sf_result` \*`result`)

These routines compute the complete elliptic integral  $E(k)$  to the accuracy specified by the mode variable `mode`. Note that Abramowitz & Stegun define this function in terms of the parameter  $m = k^2$ .



double **gsl\_sf\_ellint\_Pcomp**(double k, double n, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_Pcomp\_e**(double k, double n, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the complete elliptic integral  $\Pi(k, n)$  to the accuracy specified by the mode variable *mode*. Note that Abramowitz & Stegun define this function in terms of the parameters  $m = k^2$  and  $\sin^2(\alpha) = k^2$ , with the change of sign  $n \rightarrow -n$ .

### 7.13.4 Legendre Form of Incomplete Elliptic Integrals

double **gsl\_sf\_ellint\_F**(double phi, double k, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_F\_e**(double phi, double k, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the incomplete elliptic integral  $F(\phi, k)$  to the accuracy specified by the mode variable *mode*. Note that Abramowitz & Stegun define this function in terms of the parameter  $m = k^2$ .

double **gsl\_sf\_ellint\_E**(double phi, double k, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_E\_e**(double phi, double k, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the incomplete elliptic integral  $E(\phi, k)$  to the accuracy specified by the mode variable *mode*. Note that Abramowitz & Stegun define this function in terms of the parameter  $m = k^2$ .

double **gsl\_sf\_ellint\_P**(double phi, double k, double n, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_P\_e**(double phi, double k, double n, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the incomplete elliptic integral  $\Pi(\phi, k, n)$  to the accuracy specified by the mode variable *mode*. Note that Abramowitz & Stegun define this function in terms of the parameters  $m = k^2$  and  $\sin^2(\alpha) = k^2$ , with the change of sign  $n \rightarrow -n$ .

double **gsl\_sf\_ellint\_D**(double phi, double k, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_D\_e**(double phi, double k, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These functions compute the incomplete elliptic integral  $D(\phi, k)$  which is defined through the Carlson form  $RD(x, y, z)$  by the following relation,

$$D(\phi, k) = \frac{1}{3}(\sin \phi)^3 RD(1 - \sin^2(\phi), 1 - k^2 \sin^2(\phi), 1)$$

### 7.13.5 Carlson Forms

double **gsl\_sf\_ellint\_RC**(double x, double y, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_RC\_e**(double x, double y, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the incomplete elliptic integral  $RC(x, y)$  to the accuracy specified by the mode variable *mode*.

double **gsl\_sf\_ellint\_RD**(double x, double y, double z, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_RD\_e**(double x, double y, double z, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the incomplete elliptic integral  $RD(x, y, z)$  to the accuracy specified by the mode variable *mode*.

double **gsl\_sf\_ellint\_RF**(double x, double y, double z, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_RF\_e**(double x, double y, double z, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the incomplete elliptic integral  $RF(x, y, z)$  to the accuracy specified by the mode variable *mode*.

double **gsl\_sf\_ellint\_RJ**(double x, double y, double z, double p, *gsl\_mode\_t* mode)

int **gsl\_sf\_ellint\_RJ\_e**(double x, double y, double z, double p, *gsl\_mode\_t* mode, *gsl\_sf\_result* \*result)

These routines compute the incomplete elliptic integral  $RJ(x, y, z, p)$  to the accuracy specified by the mode variable *mode*.

## 7.14 Elliptic Functions (Jacobi)

The Jacobian Elliptic functions are defined in Abramowitz & Stegun, Chapter 16. The functions are declared in the header file `gsl_sf_elljac.h`.

int **gsl\_sf\_elljac\_e**(double u, double m, double \*sn, double \*cn, double \*dn)

This function computes the Jacobian elliptic functions  $sn(u|m)$ ,  $cn(u|m)$ ,  $dn(u|m)$  by descending Landen transformations.

## 7.15 Error Functions

The error function is described in Abramowitz & Stegun, Chapter 7. The functions in this section are declared in the header file `gsl_sf_erf.h`.

### 7.15.1 Error Function

double **gsl\_sf\_erf**(double x)

int **gsl\_sf\_erf\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the error function  $\operatorname{erf}(x)$ , where  $\operatorname{erf}(x) = (2/\sqrt{\pi}) \int_0^x dt \exp(-t^2)$ .

### 7.15.2 Complementary Error Function

double **gsl\_sf\_erfc**(double x)

int **gsl\_sf\_erfc\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the complementary error function  $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = (2/\sqrt{\pi}) \int_x^\infty \exp(-t^2)$

### 7.15.3 Log Complementary Error Function

double **gsl\_sf\_log\_erfc**(double x)

int **gsl\_sf\_log\_erfc\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the logarithm of the complementary error function  $\log(\operatorname{erfc}(x))$ .

## 7.15.4 Probability functions

The probability functions for the Normal or Gaussian distribution are described in Abramowitz & Stegun, Section 26.2.

double **gsl\_sf\_erf\_Z**(double x)

int **gsl\_sf\_erf\_Z\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Gaussian probability density function  $Z(x) = (1/\sqrt{2\pi}) \exp(-x^2/2)$

double **gsl\_sf\_erf\_Q**(double x)

int **gsl\_sf\_erf\_Q\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the upper tail of the Gaussian probability function  $Q(x) = (1/\sqrt{2\pi}) \int_x^\infty dt \exp(-t^2/2)$

The *hazard function* for the normal distribution, also known as the inverse Mills' ratio, is defined as,

$$h(x) = \frac{Z(x)}{Q(x)} = \sqrt{\frac{2}{\pi}} \frac{\exp(-x^2/2)}{\operatorname{erfc}(x/\sqrt{2})}$$

It decreases rapidly as  $x$  approaches  $-\infty$  and asymptotes to  $h(x) \sim x$  as  $x$  approaches  $+\infty$ .

double **gsl\_sf\_hazard**(double x)

int **gsl\_sf\_hazard\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the hazard function for the normal distribution.

## 7.16 Exponential Functions

The functions described in this section are declared in the header file `gsl_sf_exp.h`.

### 7.16.1 Exponential Function

double **gsl\_sf\_exp**(double x)

int **gsl\_sf\_exp\_e**(double x, *gsl\_sf\_result* \*result)

These routines provide an exponential function  $\exp(x)$  using GSL semantics and error checking.

int **gsl\_sf\_exp\_e10\_e**(double x, *gsl\_sf\_result\_e10* \*result)

This function computes the exponential  $\exp(x)$  using the *gsl\_sf\_result\_e10* type to return a result with extended range. This function may be useful if the value of  $\exp(x)$  would overflow the numeric range of double.

double **gsl\_sf\_exp\_mult**(double x, double y)

int **gsl\_sf\_exp\_mult\_e**(double x, double y, *gsl\_sf\_result* \*result)

These routines exponentiate  $x$  and multiply by the factor  $y$  to return the product  $y \exp(x)$ .

int **gsl\_sf\_exp\_mult\_e10\_e**(const double x, const double y, *gsl\_sf\_result\_e10* \*result)

This function computes the product  $y \exp(x)$  using the *gsl\_sf\_result\_e10* type to return a result with extended numeric range.

## 7.16.2 Relative Exponential Functions

double **gsl\_sf\_expm1**(double x)

int **gsl\_sf\_expm1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the quantity  $\exp(x) - 1$  using an algorithm that is accurate for small  $x$ .

double **gsl\_sf\_exprel**(double x)

int **gsl\_sf\_exprel\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the quantity  $(\exp(x) - 1)/x$  using an algorithm that is accurate for small  $x$ . For small  $x$  the algorithm is based on the expansion  $(\exp(x) - 1)/x = 1 + x/2 + x^2/(2 * 3) + x^3/(2 * 3 * 4) + \dots$

double **gsl\_sf\_exprel\_2**(double x)

int **gsl\_sf\_exprel\_2\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the quantity  $2(\exp(x) - 1 - x)/x^2$  using an algorithm that is accurate for small  $x$ . For small  $x$  the algorithm is based on the expansion  $2(\exp(x) - 1 - x)/x^2 = 1 + x/3 + x^2/(3*4) + x^3/(3*4*5) + \dots$

double **gsl\_sf\_exprel\_n**(int n, double x)

int **gsl\_sf\_exprel\_n\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the  $N$ -relative exponential, which is the  $n$ -th generalization of the functions *gsl\_sf\_exprel()* and *gsl\_sf\_exprel\_2()*. The  $N$ -relative exponential is given by,

$$\begin{aligned} \text{exprel}_N(x) &= N!/x^N \left( \exp(x) - \sum_{k=0}^{N-1} x^k/k! \right) \\ &= 1 + x/(N + 1) + x^2/((N + 1)(N + 2)) + \dots \\ &= {}_1F_1(1, 1 + N, x) \end{aligned}$$

## 7.16.3 Exponentiation With Error Estimate

int **gsl\_sf\_exp\_err\_e**(double x, double dx, *gsl\_sf\_result* \*result)

This function exponentiates  $x$  with an associated absolute error  $dx$ .

int **gsl\_sf\_exp\_err\_e10\_e**(double x, double dx, *gsl\_sf\_result\_e10* \*result)

This function exponentiates a quantity  $x$  with an associated absolute error  $dx$  using the *gsl\_sf\_result\_e10* type to return a result with extended range.

int **gsl\_sf\_exp\_mult\_err\_e**(double x, double dx, double y, double dy, *gsl\_sf\_result* \*result)

This routine computes the product  $y \exp(x)$  for the quantities  $x, y$  with associated absolute errors  $dx, dy$ .

int **gsl\_sf\_exp\_mult\_err\_e10\_e**(double x, double dx, double y, double dy, *gsl\_sf\_result\_e10* \*result)

This routine computes the product  $y \exp(x)$  for the quantities  $x, y$  with associated absolute errors  $dx, dy$  using the *gsl\_sf\_result\_e10* type to return a result with extended range.

## 7.17 Exponential Integrals

Information on the exponential integrals can be found in Abramowitz & Stegun, Chapter 5. These functions are declared in the header file `gsl_sf_expint.h`.

### 7.17.1 Exponential Integral

double `gsl_sf_expint_E1`(double x)

int `gsl_sf_expint_E1_e`(double x, *gsl\_sf\_result* \*result)

These routines compute the exponential integral  $E_1(x)$ ,

$$E_1(x) := \Re \int_1^{\infty} dt \exp(-xt)/t.$$

double `gsl_sf_expint_E2`(double x)

int `gsl_sf_expint_E2_e`(double x, *gsl\_sf\_result* \*result)

These routines compute the second-order exponential integral  $E_2(x)$ ,

$$E_2(x) := \Re \int_1^{\infty} dt \exp(-xt)/t^2$$

double `gsl_sf_expint_En`(int n, double x)

int `gsl_sf_expint_En_e`(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the exponential integral  $E_n(x)$  of order  $n$ ,

$$E_n(x) := \Re \int_1^{\infty} dt \exp(-xt)/t^n.$$

### 7.17.2 Ei(x)

double `gsl_sf_expint_Ei`(double x)

int `gsl_sf_expint_Ei_e`(double x, *gsl\_sf\_result* \*result)

These routines compute the exponential integral  $Ei(x)$ ,

$$Ei(x) = -PV \left( \int_{-x}^{\infty} dt \exp(-t)/t \right)$$

where  $PV$  denotes the principal value of the integral.

### 7.17.3 Hyperbolic Integrals

double `gsl_sf_Shi`(double x)

int `gsl_sf_Shi_e`(double x, *gsl\_sf\_result* \*result)

These routines compute the integral

$$\text{Shi}(x) = \int_0^x dt \sinh(t)/t$$

double **gsl\_sf\_Chi**(double x)

int **gsl\_sf\_Chi\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the integral

$$\text{Chi}(x) := \Re \left[ \gamma_E + \log(x) + \int_0^x dt (\cosh(t) - 1)/t \right]$$

where  $\gamma_E$  is the Euler constant (available as the macro `M_EULER`).

### 7.17.4 **Ei\_3(x)**

double **gsl\_sf\_expint\_3**(double x)

int **gsl\_sf\_expint\_3\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the third-order exponential integral

$$\text{Ei}_3(x) = \int_0^x dt \exp(-t^3)$$

for  $x \geq 0$ .

### 7.17.5 Trigonometric Integrals

double **gsl\_sf\_Si**(const double x)

int **gsl\_sf\_Si\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Sine integral

$$\text{Si}(x) = \int_0^x dt \sin(t)/t$$

double **gsl\_sf\_Ci**(const double x)

int **gsl\_sf\_Ci\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Cosine integral

$$\text{Ci}(x) = - \int_x^\infty dt \cos(t)/t$$

for  $x > 0$

### 7.17.6 Arctangent Integral

double **gsl\_sf\_atanint**(double x)

int **gsl\_sf\_atanint\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Arctangent integral, which is defined as

$$\text{AtanInt}(x) = \int_0^x dt \arctan(t)/t$$

## 7.18 Fermi-Dirac Function

The functions described in this section are declared in the header file `gsl_sf_fermi_dirac.h`.

### 7.18.1 Complete Fermi-Dirac Integrals

The complete Fermi-Dirac integral  $F_j(x)$  is given by,

$$F_j(x) := \frac{1}{\Gamma(j+1)} \int_0^\infty dt \frac{t^j}{(\exp(t-x) + 1)}$$

Note that the Fermi-Dirac integral is sometimes defined without the normalisation factor in other texts.

`double gsl_sf_fermi_dirac_m1(double x)`

`int gsl_sf_fermi_dirac_m1_e(double x, gsl_sf_result *result)`

These routines compute the complete Fermi-Dirac integral with an index of  $-1$ . This integral is given by  $F_{-1}(x) = e^x/(1 + e^x)$ .

`double gsl_sf_fermi_dirac_0(double x)`

`int gsl_sf_fermi_dirac_0_e(double x, gsl_sf_result *result)`

These routines compute the complete Fermi-Dirac integral with an index of  $0$ . This integral is given by  $F_0(x) = \ln(1 + e^x)$ .

`double gsl_sf_fermi_dirac_1(double x)`

`int gsl_sf_fermi_dirac_1_e(double x, gsl_sf_result *result)`

These routines compute the complete Fermi-Dirac integral with an index of  $1$ ,  $F_1(x) = \int_0^\infty dt(t/(\exp(t-x) + 1))$ .

`double gsl_sf_fermi_dirac_2(double x)`

`int gsl_sf_fermi_dirac_2_e(double x, gsl_sf_result *result)`

These routines compute the complete Fermi-Dirac integral with an index of  $2$ ,  $F_2(x) = (1/2) \int_0^\infty dt(t^2/(\exp(t-x) + 1))$ .

`double gsl_sf_fermi_dirac_int(int j, double x)`

`int gsl_sf_fermi_dirac_int_e(int j, double x, gsl_sf_result *result)`

These routines compute the complete Fermi-Dirac integral with an integer index of  $j$ ,  $F_j(x) = (1/\Gamma(j+1)) \int_0^\infty dt(t^j/(\exp(t-x) + 1))$ .

`double gsl_sf_fermi_dirac_mhalf(double x)`

`int gsl_sf_fermi_dirac_mhalf_e(double x, gsl_sf_result *result)`

These routines compute the complete Fermi-Dirac integral  $F_{-1/2}(x)$ .

`double gsl_sf_fermi_dirac_half(double x)`

`int gsl_sf_fermi_dirac_half_e(double x, gsl_sf_result *result)`

These routines compute the complete Fermi-Dirac integral  $F_{1/2}(x)$ .

`double gsl_sf_fermi_dirac_3half(double x)`

int **gsl\_sf\_fermi\_dirac\_3half\_e**(double x, *gsl\_sf\_result* \*result)  
These routines compute the complete Fermi-Dirac integral  $F_{3/2}(x)$ .

## 7.18.2 Incomplete Fermi-Dirac Integrals

The incomplete Fermi-Dirac integral  $F_j(x, b)$  is given by,

$$F_j(x, b) := \frac{1}{\Gamma(j+1)} \int_b^\infty dt \frac{t^j}{(\exp(t-x)+1)}$$

double **gsl\_sf\_fermi\_dirac\_inc\_0**(double x, double b)

int **gsl\_sf\_fermi\_dirac\_inc\_0\_e**(double x, double b, *gsl\_sf\_result* \*result)  
These routines compute the incomplete Fermi-Dirac integral with an index of zero,  $F_0(x, b) = \ln(1 + e^{b-x}) - (b - x)$

## 7.19 Gamma and Beta Functions

The following routines compute the gamma and beta functions in their full and incomplete forms, as well as various kinds of factorials. The functions described in this section are declared in the header file `gsl_sf_gamma.h`.

### 7.19.1 Gamma Functions

The Gamma function is defined by the following integral,

$$\Gamma(x) = \int_0^\infty dt t^{x-1} \exp(-t)$$

It is related to the factorial function by  $\Gamma(n) = (n-1)!$  for positive integer  $n$ . Further information on the Gamma function can be found in Abramowitz & Stegun, Chapter 6.

double **gsl\_sf\_gamma**(double x)

int **gsl\_sf\_gamma\_e**(double x, *gsl\_sf\_result* \*result)  
These routines compute the Gamma function  $\Gamma(x)$ , subject to  $x$  not being a negative integer or zero. The function is computed using the real Lanczos method. The maximum value of  $x$  such that  $\Gamma(x)$  is not considered an overflow is given by the macro `GSL_SF_GAMMA_XMAX` and is 171.0.

double **gsl\_sf\_lngamma**(double x)

int **gsl\_sf\_lngamma\_e**(double x, *gsl\_sf\_result* \*result)  
These routines compute the logarithm of the Gamma function,  $\log(\Gamma(x))$ , subject to  $x$  not being a negative integer or zero. For  $x < 0$  the real part of  $\log(\Gamma(x))$  is returned, which is equivalent to  $\log(|\Gamma(x)|)$ . The function is computed using the real Lanczos method.

int **gsl\_sf\_lngamma\_sgn\_e**(double x, *gsl\_sf\_result* \*result\_lg, double \*sgn)  
This routine computes the sign of the gamma function and the logarithm of its magnitude, subject to  $x$  not being a negative integer or zero. The function is computed using the real Lanczos method. The value of the gamma function and its error can be reconstructed using the relation  $\Gamma(x) = \text{sgn} * \exp(\text{result\_lg})$ , taking into account the two components of *result\_lg*.



double **gsl\_sf\_gammastar**(double x)

int **gsl\_sf\_gammastar\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the regulated Gamma Function  $\Gamma^*(x)$  for  $x > 0$ . The regulated gamma function is given by,

$$\begin{aligned}\Gamma^*(x) &= \Gamma(x) / (\sqrt{2\pi} x^{(x-1/2)} \exp(-x)) \\ &= \left(1 + \frac{1}{12x} + \dots\right) \quad \text{for } x \rightarrow \infty\end{aligned}$$

and is a useful suggestion of Temme.

double **gsl\_sf\_gammainv**(double x)

int **gsl\_sf\_gammainv\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the reciprocal of the gamma function,  $1/\Gamma(x)$  using the real Lanczos method.

int **gsl\_sf\_lngamma\_complex\_e**(double zr, double zi, *gsl\_sf\_result* \*lnr, *gsl\_sf\_result* \*arg)

This routine computes  $\log(\Gamma(z))$  for complex  $z = z_r + iz_i$  and  $z$  not a negative integer or zero, using the complex Lanczos method. The returned parameters are  $lnr = \log|\Gamma(z)|$  and  $arg = \arg(\Gamma(z))$  in  $(-\pi, \pi]$ . Note that the phase part (*arg*) is not well-determined when  $|z|$  is very large, due to inevitable roundoff in restricting to  $(-\pi, \pi]$ . This will result in a GSL\_ELOSS error when it occurs. The absolute value part (*lnr*), however, never suffers from loss of precision.

## 7.19.2 Factorials

Although factorials can be computed from the Gamma function, using the relation  $n! = \Gamma(n + 1)$  for non-negative integer  $n$ , it is usually more efficient to call the functions in this section, particularly for small values of  $n$ , whose factorial values are maintained in hardcoded tables.

double **gsl\_sf\_fact**(unsigned int n)

int **gsl\_sf\_fact\_e**(unsigned int n, *gsl\_sf\_result* \*result)

These routines compute the factorial  $n!$ . The factorial is related to the Gamma function by  $n! = \Gamma(n + 1)$ . The maximum value of  $n$  such that  $n!$  is not considered an overflow is given by the macro `GSL_SF_FACT_NMAX` and is 170.

double **gsl\_sf\_doublefact**(unsigned int n)

int **gsl\_sf\_doublefact\_e**(unsigned int n, *gsl\_sf\_result* \*result)

These routines compute the double factorial  $n!! = n(n-2)(n-4)\dots$ . The maximum value of  $n$  such that  $n!!$  is not considered an overflow is given by the macro `GSL_SF_DOUBLEFACT_NMAX` and is 297.

double **gsl\_sf\_lnfact**(unsigned int n)

int **gsl\_sf\_lnfact\_e**(unsigned int n, *gsl\_sf\_result* \*result)

These routines compute the logarithm of the factorial of  $n$ ,  $\log(n!)$ . The algorithm is faster than computing  $\ln(\Gamma(n + 1))$  via `gsl_sf_lngamma()` for  $n < 170$ , but defers for larger  $n$ .

double **gsl\_sf\_lndoublefact**(unsigned int n)

int **gsl\_sf\_lndoublefact\_e**(unsigned int n, *gsl\_sf\_result* \*result)

These routines compute the logarithm of the double factorial of  $n$ ,  $\log(n!!)$ .

double **gsl\_sf\_choose**(unsigned int n, unsigned int m)

int **gsl\_sf\_choose\_e**(unsigned int n, unsigned int m, *gsl\_sf\_result* \*result)

These routines compute the combinatorial factor  $n \text{ choose } m = n!/(m!(n-m)!)$

double **gsl\_sf\_lnchoose**(unsigned int n, unsigned int m)

int **gsl\_sf\_lnchoose\_e**(unsigned int n, unsigned int m, *gsl\_sf\_result* \*result)

These routines compute the logarithm of  $n \text{ choose } m$ . This is equivalent to the sum  $\log(n!) - \log(m!) - \log((n-m)!)$ .

double **gsl\_sf\_taylorcoeff**(int n, double x)

int **gsl\_sf\_taylorcoeff\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the Taylor coefficient  $x^n/n!$  for  $x \geq 0, n \geq 0$

### 7.19.3 Pochhammer Symbol

double **gsl\_sf\_poch**(double a, double x)

int **gsl\_sf\_poch\_e**(double a, double x, *gsl\_sf\_result* \*result)

These routines compute the Pochhammer symbol  $(a)_x = \Gamma(a+x)/\Gamma(a)$ . The Pochhammer symbol is also known as the Apell symbol and sometimes written as  $(a, x)$ . When  $a$  and  $a+x$  are negative integers or zero, the limiting value of the ratio is returned.

double **gsl\_sf\_lnpoch**(double a, double x)

int **gsl\_sf\_lnpoch\_e**(double a, double x, *gsl\_sf\_result* \*result)

These routines compute the logarithm of the Pochhammer symbol,  $\log((a)_x) = \log(\Gamma(a+x)/\Gamma(a))$ .

int **gsl\_sf\_lnpoch\_sgn\_e**(double a, double x, *gsl\_sf\_result* \*result, double \*sgn)

These routines compute the sign of the Pochhammer symbol and the logarithm of its magnitude. The computed parameters are  $result = \log(|(a)_x|)$  with a corresponding error term, and  $sgn = \text{sgn}((a)_x)$  where  $(a)_x = \Gamma(a+x)/\Gamma(a)$ .

double **gsl\_sf\_pochrel**(double a, double x)

int **gsl\_sf\_pochrel\_e**(double a, double x, *gsl\_sf\_result* \*result)

These routines compute the relative Pochhammer symbol  $((a)_x - 1)/x$  where  $(a)_x = \Gamma(a+x)/\Gamma(a)$ .

### 7.19.4 Incomplete Gamma Functions

double **gsl\_sf\_gamma\_inc**(double a, double x)

int **gsl\_sf\_gamma\_inc\_e**(double a, double x, *gsl\_sf\_result* \*result)

These functions compute the unnormalized incomplete Gamma Function  $\Gamma(a, x) = \int_x^\infty dt t^{(a-1)} \exp(-t)$  for  $a$  real and  $x \geq 0$ .

double **gsl\_sf\_gamma\_inc\_Q**(double a, double x)

int **gsl\_sf\_gamma\_inc\_Q\_e**(double a, double x, *gsl\_sf\_result* \*result)

These routines compute the normalized incomplete Gamma Function  $Q(a, x) = 1/\Gamma(a) \int_x^\infty dt t^{(a-1)} \exp(-t)$  for  $a > 0, x \geq 0$ .

double **gsl\_sf\_gamma\_inc\_P**(double a, double x)

int **gsl\_sf\_gamma\_inc\_P\_e**(double a, double x, *gsl\_sf\_result* \*result)

These routines compute the complementary normalized incomplete Gamma Function  $P(a, x) = 1 - Q(a, x) = 1/\Gamma(a) \int_0^x dt t^{a-1} \exp(-t)$  for  $a > 0, x \geq 0$ .

Note that Abramowitz & Stegun call  $P(a, x)$  the incomplete gamma function (section 6.5).

## 7.19.5 Beta Functions

double **gsl\_sf\_beta**(double a, double b)

int **gsl\_sf\_beta\_e**(double a, double b, *gsl\_sf\_result* \*result)

These routines compute the Beta Function,  $B(a, b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$  subject to  $a$  and  $b$  not being negative integers.

double **gsl\_sf\_lnbeta**(double a, double b)

int **gsl\_sf\_lnbeta\_e**(double a, double b, *gsl\_sf\_result* \*result)

These routines compute the logarithm of the Beta Function,  $\log(B(a, b))$  subject to  $a$  and  $b$  not being negative integers.

## 7.19.6 Incomplete Beta Function

double **gsl\_sf\_beta\_inc**(double a, double b, double x)

int **gsl\_sf\_beta\_inc\_e**(double a, double b, double x, *gsl\_sf\_result* \*result)

These routines compute the normalized incomplete Beta function  $I_x(a, b) = B_x(a, b)/B(a, b)$  where

$$B_x(a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt$$

for  $0 \leq x \leq 1$ . For  $a > 0, b > 0$  the value is computed using a continued fraction expansion. For all other values it is computed using the relation

$$I_x(a, b, x) = (1/a)x^a {}_2F_1(a, 1-b, a+1, x)/B(a, b)$$

## 7.20 Gegenbauer Functions

The Gegenbauer polynomials are defined in Abramowitz & Stegun, Chapter 22, where they are known as Ultraspherical polynomials. The functions described in this section are declared in the header file `gsl_sf_gegenbauer.h`.

double **gsl\_sf\_gegenpoly\_1**(double lambda, double x)

double **gsl\_sf\_gegenpoly\_2**(double lambda, double x)

double **gsl\_sf\_gegenpoly\_3**(double lambda, double x)

int **gsl\_sf\_gegenpoly\_1\_e**(double lambda, double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_gegenpoly\_2\_e**(double lambda, double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_gegenpoly\_3\_e**(double lambda, double x, *gsl\_sf\_result* \*result)

These functions evaluate the Gegenbauer polynomials  $C_n^{(\lambda)}(x)$  using explicit representations for  $n = 1, 2, 3$ .

double **gsl\_sf\_gegenpoly\_n**(int n, double lambda, double x)

int **gsl\_sf\_gegenpoly\_n\_e**(int n, double lambda, double x, *gsl\_sf\_result* \*result)

These functions evaluate the Gegenbauer polynomial  $C_n^{(\lambda)}(x)$  for a specific value of  $n$ ,  $\lambda$ ,  $x$  subject to  $\lambda > -1/2$ ,  $n \geq 0$ .

int **gsl\_sf\_gegenpoly\_array**(int nmax, double lambda, double x, double result\_array[])

This function computes an array of Gegenbauer polynomials  $C_n^{(\lambda)}(x)$  for  $n = 0, 1, 2, \dots, nmax$ , subject to  $\lambda > -1/2$ ,  $nmax \geq 0$ .

## 7.21 Hermite Polynomials and Functions

Hermite polynomials and functions are discussed in Abramowitz & Stegun, Chapter 22 and Szego, Gabor (1939, 1958, 1967), Orthogonal Polynomials, American Mathematical Society. The Hermite polynomials and functions are defined in the header file `gsl_sf_hermite.h`.

### 7.21.1 Hermite Polynomials

The Hermite polynomials exist in two variants: the physicist version  $H_n(x)$  and the probabilist version  $He_n(x)$ . They are defined by the derivatives

$$H_n(x) = (-1)^n e^{x^2} \left( \frac{d}{dx} \right)^n e^{-x^2}$$
$$He_n(x) = (-1)^n e^{x^2/2} \left( \frac{d}{dx} \right)^n e^{-x^2/2}$$

They are connected via

$$H_n(x) = 2^{n/2} He_n(\sqrt{2}x)$$
$$He_n(x) = 2^{-n/2} H_n\left(\frac{x}{\sqrt{2}}\right)$$

and satisfy the ordinary differential equations

$$H_n''(x) - 2xH_n'(x) + 2nH_n(x) = 0$$
$$He_n''(x) - xHe_n'(x) + nHe_n(x) = 0$$

double **gsl\_sf\_hermite**(const int n, const double x)

int **gsl\_sf\_hermite\_e**(const int n, const double x, *gsl\_sf\_result* \*result)

These routines evaluate the physicist Hermite polynomial  $H_n(x)$  of order  $n$  at position  $x$ . If an overflow is detected, `GSL_EOVRFLW` is returned without calling the error handler.

int **gsl\_sf\_hermite\_array**(const int nmax, const double x, double \*result\_array)

This routine evaluates all physicist Hermite polynomials  $H_n$  up to order  $nmax$  at position  $x$ . The results are stored in `result_array`.

double **gsl\_sf\_hermite\_series**(const int n, const double x, const double \*a)

int **gsl\_sf\_hermite\_series\_e**(const int n, const double x, const double \*a, *gsl\_sf\_result* \*result)  
 These routines evaluate the series  $\sum_{j=0}^n a_j H_j(x)$  with  $H_j$  being the  $j$ -th physicist Hermite polynomial using the Clenshaw algorithm.

double **gsl\_sf\_hermite\_prob**(const int n, const double x)

int **gsl\_sf\_hermite\_prob\_e**(const int n, const double x, *gsl\_sf\_result* \*result)  
 These routines evaluate the probabilist Hermite polynomial  $He_n(x)$  of order  $n$  at position  $x$ . If an overflow is detected, `GSL_EOVRFLW` is returned without calling the error handler.

int **gsl\_sf\_hermite\_prob\_array**(const int nmax, const double x, double \*result\_array)  
 This routine evaluates all probabilist Hermite polynomials  $He_n(x)$  up to order  $nmax$  at position  $x$ . The results are stored in *result\_array*.

double **gsl\_sf\_hermite\_prob\_series**(const int n, const double x, const double \*a)

int **gsl\_sf\_hermite\_prob\_series\_e**(const int n, const double x, const double \*a, *gsl\_sf\_result* \*result)  
 These routines evaluate the series  $\sum_{j=0}^n a_j He_j(x)$  with  $He_j$  being the  $j$ -th probabilist Hermite polynomial using the Clenshaw algorithm.

## 7.21.2 Derivatives of Hermite Polynomials

double **gsl\_sf\_hermite\_deriv**(const int m, const int n, const double x)

int **gsl\_sf\_hermite\_deriv\_e**(const int m, const int n, const double x, *gsl\_sf\_result* \*result)  
 These routines evaluate the  $m$ -th derivative of the physicist Hermite polynomial  $H_n(x)$  of order  $n$  at position  $x$ .

int **gsl\_sf\_hermite\_array\_deriv**(const int m, const int nmax, const double x, double \*result\_array)  
 This routine evaluates the  $m$ -th derivative of all physicist Hermite polynomials  $H_n(x)$  from orders  $0, \dots, nmax$  at position  $x$ . The result  $d^m/dx^m H_n(x)$  is stored in `result_array[n]`. The output *result\_array* must have length at least `nmax + 1`.

int **gsl\_sf\_hermite\_deriv\_array**(const int mmax, const int n, const double x, double \*result\_array)  
 This routine evaluates all derivative orders from  $0, \dots, mmax$  of the physicist Hermite polynomial of order  $n$ ,  $H_n$ , at position  $x$ . The result  $d^m/dx^m H_n(x)$  is stored in `result_array[m]`. The output *result\_array* must have length at least `mmax + 1`.

double **gsl\_sf\_hermite\_prob\_deriv**(const int m, const int n, const double x)

int **gsl\_sf\_hermite\_prob\_deriv\_e**(const int m, const int n, const double x, *gsl\_sf\_result* \*result)  
 These routines evaluate the  $m$ -th derivative of the probabilist Hermite polynomial  $He_n(x)$  of order  $n$  at position  $x$ .

int **gsl\_sf\_hermite\_prob\_array\_deriv**(const int m, const int nmax, const double x, double \*result\_array)  
 This routine evaluates the  $m$ -th derivative of all probabilist Hermite polynomials  $He_n(x)$  from orders  $0, \dots, nmax$  at position  $x$ . The result  $d^m/dx^m He_n(x)$  is stored in `result_array[n]`. The output *result\_array* must have length at least `nmax + 1`.

int **gsl\_sf\_hermite\_prob\_deriv\_array**(const int mmax, const int n, const double x, double \*result\_array)  
 This routine evaluates all derivative orders from  $0, \dots, mmax$  of the probabilist Hermite polynomial of order  $n$ ,  $He_n$ , at position  $x$ . The result  $d^m/dx^m He_n(x)$  is stored in `result_array[m]`. The output *result\_array* must have length at least `mmax + 1`.

### 7.21.3 Hermite Functions

The Hermite functions are defined by

$$\psi_n(x) = (2^n n! \sqrt{\pi})^{-1/2} e^{-x^2/2} H_n(x)$$

and satisfy the Schrödinger equation for a quantum mechanical harmonic oscillator

$$\psi_n''(x) + (2n + 1 - x^2)\psi_n(x) = 0$$

They are orthonormal,

$$\int_{-\infty}^{\infty} \psi_m(x)\psi_n(x)dx = \delta_{mn}$$

and form an orthonormal basis of  $L^2(\mathbb{R})$ . The Hermite functions are also eigenfunctions of the continuous Fourier transform. GSL offers two methods for evaluating the Hermite functions. The first uses the standard three-term recurrence relation which has  $O(n)$  complexity and is the most accurate. The second uses a Cauchy integral approach due to Bunck (2009) which has  $O(\sqrt{n})$  complexity which represents a significant speed improvement for large  $n$ , although it is slightly less accurate.

double **gsl\_sf\_hermite\_func**(const int n, const double x)

int **gsl\_sf\_hermite\_func\_e**(const int n, const double x, *gsl\_sf\_result* \*result)

These routines evaluate the Hermite function  $\psi_n(x)$  of order  $n$  at position  $x$  using a three term recurrence relation. The algorithm complexity is  $O(n)$ .

double **gsl\_sf\_hermite\_func\_fast**(const int n, const double x)

int **gsl\_sf\_hermite\_func\_fast\_e**(const int n, const double x, *gsl\_sf\_result* \*result)

These routines evaluate the Hermite function  $\psi_n(x)$  of order  $n$  at position  $x$  using a the Cauchy integral algorithm due to Bunck, 2009. The algorithm complexity is  $O(\sqrt{n})$ .

int **gsl\_sf\_hermite\_func\_array**(const int nmax, const double x, double \*result\_array)

This routine evaluates all Hermite functions  $\psi_n(x)$  for orders  $n = 0, \dots, nmax$  at position  $x$ , using the recurrence relation algorithm. The results are stored in *result\_array* which has length at least  $nmax + 1$ .

double **gsl\_sf\_hermite\_func\_series**(const int n, const double x, const double \*a)

int **gsl\_sf\_hermite\_func\_series\_e**(const int n, const double x, const double \*a, *gsl\_sf\_result* \*result)

These routines evaluate the series  $\sum_{j=0}^n a_j \psi_j(x)$  with  $\psi_j$  being the  $j$ -th Hermite function using the Clenshaw algorithm.

### 7.21.4 Derivatives of Hermite Functions

double **gsl\_sf\_hermite\_func\_der**(const int m, const int n, const double x)

int **gsl\_sf\_hermite\_func\_der\_e**(const int m, const int n, const double x, *gsl\_sf\_result* \*result)

These routines evaluate the  $m$ -th derivative of the Hermite function  $\psi_n(x)$  of order  $n$  at position  $x$ .

## 7.21.5 Zeros of Hermite Polynomials and Hermite Functions

These routines calculate the  $s$ -th zero of the Hermite polynomial/function of order  $n$ . Since the zeros are symmetrical around zero, only positive zeros are calculated, ordered from smallest to largest, starting from index 1. Only for odd polynomial orders a zeroth zero exists, its value always being zero.

double **gsl\_sf\_hermite\_zero**(const int n, const int s)

int **gsl\_sf\_hermite\_zero\_e**(const int n, const int s, *gsl\_sf\_result* \*result)

These routines evaluate the  $s$ -th zero of the physicist Hermite polynomial  $H_n(x)$  of order  $n$ .

double **gsl\_sf\_hermite\_prob\_zero**(const int n, const int s)

int **gsl\_sf\_hermite\_prob\_zero\_e**(const int n, const int s, *gsl\_sf\_result* \*result)

These routines evaluate the  $s$ -th zero of the probabilist Hermite polynomial  $He_n(x)$  of order  $n$ .

double **gsl\_sf\_hermite\_func\_zero**(const int n, const int s)

int **gsl\_sf\_hermite\_func\_zero\_e**(const int n, const int s, *gsl\_sf\_result* \*result)

These routines evaluate the  $s$ -th zero of the Hermite function  $\psi_n(x)$  of order  $n$ .

## 7.22 Hypergeometric Functions

Hypergeometric functions are described in Abramowitz & Stegun, Chapters 13 and 15. These functions are declared in the header file `gsl_sf_hyperg.h`.

double **gsl\_sf\_hyperg\_0F1**(double c, double x)

int **gsl\_sf\_hyperg\_0F1\_e**(double c, double x, *gsl\_sf\_result* \*result)

These routines compute the hypergeometric function

$${}_0F_1(c, x)$$

double **gsl\_sf\_hyperg\_1F1\_int**(int m, int n, double x)

int **gsl\_sf\_hyperg\_1F1\_int\_e**(int m, int n, double x, *gsl\_sf\_result* \*result)

These routines compute the confluent hypergeometric function

$${}_1F_1(m, n, x) = M(m, n, x)$$

for integer parameters  $m, n$ .

double **gsl\_sf\_hyperg\_1F1**(double a, double b, double x)

int **gsl\_sf\_hyperg\_1F1\_e**(double a, double b, double x, *gsl\_sf\_result* \*result)

These routines compute the confluent hypergeometric function

$${}_1F_1(a, b, x) = M(a, b, x)$$

for general parameters  $a, b$ .

double **gsl\_sf\_hyperg\_U\_int**(int m, int n, double x)

int **gsl\_sf\_hyperg\_U\_int\_e**(int m, int n, double x, *gsl\_sf\_result* \*result)

These routines compute the confluent hypergeometric function  $U(m, n, x)$  for integer parameters  $m, n$ .

int **gsl\_sf\_hyperg\_U\_int\_e10\_e**(int m, int n, double x, *gsl\_sf\_result\_e10* \*result)

This routine computes the confluent hypergeometric function  $U(m, n, x)$  for integer parameters  $m, n$  using the *gsl\_sf\_result\_e10* type to return a result with extended range.

double **gsl\_sf\_hyperg\_U**(double a, double b, double x)

int **gsl\_sf\_hyperg\_U\_e**(double a, double b, double x, *gsl\_sf\_result* \*result)

These routines compute the confluent hypergeometric function  $U(a, b, x)$ .

int **gsl\_sf\_hyperg\_U\_e10\_e**(double a, double b, double x, *gsl\_sf\_result\_e10* \*result)

This routine computes the confluent hypergeometric function  $U(a, b, x)$  using the *gsl\_sf\_result\_e10* type to return a result with extended range.

double **gsl\_sf\_hyperg\_2F1**(double a, double b, double c, double x)

int **gsl\_sf\_hyperg\_2F1\_e**(double a, double b, double c, double x, *gsl\_sf\_result* \*result)

These routines compute the Gauss hypergeometric function

$${}_2F_1(a, b, c, x) = F(a, b, c, x)$$

for  $|x| < 1$ . If the arguments  $(a, b, c, x)$  are too close to a singularity then the function can return the error code `GSL_EMAXITER` when the series approximation converges too slowly. This occurs in the region of  $x = 1$ ,  $c - a - b = m$  for integer  $m$ .

double **gsl\_sf\_hyperg\_2F1\_conj**(double aR, double aI, double c, double x)

int **gsl\_sf\_hyperg\_2F1\_conj\_e**(double aR, double aI, double c, double x, *gsl\_sf\_result* \*result)

These routines compute the Gauss hypergeometric function

$${}_2F_1(a_R + ia_I, a_R - ia_I, c, x)$$

with complex parameters for  $|x| < 1$ .

double **gsl\_sf\_hyperg\_2F1\_renorm**(double a, double b, double c, double x)

int **gsl\_sf\_hyperg\_2F1\_renorm\_e**(double a, double b, double c, double x, *gsl\_sf\_result* \*result)

These routines compute the renormalized Gauss hypergeometric function

$${}_2F_1(a, b, c, x)/\Gamma(c)$$

for  $|x| < 1$ .

double **gsl\_sf\_hyperg\_2F1\_conj\_renorm**(double aR, double aI, double c, double x)

int **gsl\_sf\_hyperg\_2F1\_conj\_renorm\_e**(double aR, double aI, double c, double x, *gsl\_sf\_result* \*result)

These routines compute the renormalized Gauss hypergeometric function

$${}_2F_1(a_R + ia_I, a_R - ia_I, c, x)/\Gamma(c)$$

for  $|x| < 1$ .

double **gsl\_sf\_hyperg\_2F0**(double a, double b, double x)



int **gsl\_sf\_hyperg\_2F0\_e**(double a, double b, double x, *gsl\_sf\_result* \*result)

These routines compute the hypergeometric function

$${}_2F_0(a, b, x)$$

The series representation is a divergent hypergeometric series. However, for  $x < 0$  we have

$${}_2F_0(a, b, x) = (-1/x)^a U(a, 1 + a - b, -1/x)$$

## 7.23 Laguerre Functions

The generalized Laguerre polynomials, sometimes referred to as associated Laguerre polynomials, are defined in terms of confluent hypergeometric functions as

$$L_n^a(x) = \frac{(a+1)_n}{n!} {}_1F_1(-n, a+1, x)$$

where  $(a)_n$  is the *Pochhammer symbol* (rising factorial). They are related to the plain Laguerre polynomials  $L_n(x)$  by  $L_n^0(x) = L_n(x)$  and  $L_n^k(x) = (-1)^k (d^k/dx^k) L_{(n+k)}(x)$ . For more information see Abramowitz & Stegun, Chapter 22.

The functions described in this section are declared in the header file `gsl_sf_laguerre.h`.

double **gsl\_sf\_laguerre\_1**(double a, double x)

double **gsl\_sf\_laguerre\_2**(double a, double x)

double **gsl\_sf\_laguerre\_3**(double a, double x)

int **gsl\_sf\_laguerre\_1\_e**(double a, double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_laguerre\_2\_e**(double a, double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_laguerre\_3\_e**(double a, double x, *gsl\_sf\_result* \*result)

These routines evaluate the generalized Laguerre polynomials  $L_1^a(x)$ ,  $L_2^a(x)$ ,  $L_3^a(x)$  using explicit representations.

double **gsl\_sf\_laguerre\_n**(const int n, const double a, const double x)

int **gsl\_sf\_laguerre\_n\_e**(int n, double a, double x, *gsl\_sf\_result* \*result)

These routines evaluate the generalized Laguerre polynomials  $L_n^a(x)$  for  $a > -1$ ,  $n \geq 0$ .

## 7.24 Lambert W Functions

Lambert's W functions,  $W(x)$ , are defined to be solutions of the equation  $W(x) \exp(W(x)) = x$ . This function has multiple branches for  $x < 0$ ; however, it has only two real-valued branches. We define  $W_0(x)$  to be the principal branch, where  $W > -1$  for  $x < 0$ , and  $W_{-1}(x)$  to be the other real branch, where  $W < -1$  for  $x < 0$ . The Lambert functions are declared in the header file `gsl_sf_lambert.h`.

double **gsl\_sf\_lambert\_W0**(double x)

int **gsl\_sf\_lambert\_W0\_e**(double x, *gsl\_sf\_result* \*result)  
These compute the principal branch of the Lambert W function,  $W_0(x)$ .

double **gsl\_sf\_lambert\_Wm1**(double x)

int **gsl\_sf\_lambert\_Wm1\_e**(double x, *gsl\_sf\_result* \*result)  
These compute the secondary real-valued branch of the Lambert W function,  $W_{-1}(x)$ .

## 7.25 Legendre Functions and Spherical Harmonics

The Legendre Functions and Legendre Polynomials are described in Abramowitz & Stegun, Chapter 8. These functions are declared in the header file `gsl_sf_legendre.h`.

### 7.25.1 Legendre Polynomials

double **gsl\_sf\_legendre\_P1**(double x)

double **gsl\_sf\_legendre\_P2**(double x)

double **gsl\_sf\_legendre\_P3**(double x)

int **gsl\_sf\_legendre\_P1\_e**(double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_legendre\_P2\_e**(double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_legendre\_P3\_e**(double x, *gsl\_sf\_result* \*result)

These functions evaluate the Legendre polynomials  $P_l(x)$  using explicit representations for  $l = 1, 2, 3$ .

double **gsl\_sf\_legendre\_P1**(int l, double x)

int **gsl\_sf\_legendre\_P1\_e**(int l, double x, *gsl\_sf\_result* \*result)

These functions evaluate the Legendre polynomial  $P_l(x)$  for a specific value of  $l$ ,  $x$  subject to  $l \geq 0$  and  $|x| \leq 1$ .

int **gsl\_sf\_legendre\_P1\_array**(int lmax, double x, double result\_array[])

int **gsl\_sf\_legendre\_P1\_deriv\_array**(int lmax, double x, double result\_array[], double result\_deriv\_array[])

These functions compute arrays of Legendre polynomials  $P_l(x)$  and derivatives  $dP_l(x)/dx$  for  $l = 0, \dots, lmax$  and  $|x| \leq 1$ .

double **gsl\_sf\_legendre\_Q0**(double x)

int **gsl\_sf\_legendre\_Q0\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Legendre function  $Q_0(x)$  for  $x > -1$  and  $x \neq 1$ .

double **gsl\_sf\_legendre\_Q1**(double x)

int **gsl\_sf\_legendre\_Q1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Legendre function  $Q_1(x)$  for  $x > -1$  and  $x \neq 1$ .

double **gsl\_sf\_legendre\_Q1**(int l, double x)

int **gsl\_sf\_legendre\_Ql\_e**(int l, double x, *gsl\_sf\_result* \*result)

These routines compute the Legendre function  $Q_l(x)$  for  $x > -1$ ,  $x \neq 1$  and  $l \geq 0$ .

## 7.25.2 Associated Legendre Polynomials and Spherical Harmonics

The following functions compute the associated Legendre polynomials  $P_l^m(x)$  which are solutions of the differential equation

$$(1-x^2)\frac{d^2}{dx^2}P_l^m(x) - 2x\frac{d}{dx}P_l^m(x) + \left(l(l+1) - \frac{m^2}{1-x^2}\right)P_l^m(x) = 0$$

where the degree  $l$  and order  $m$  satisfy  $0 \leq l$  and  $0 \leq m \leq l$ . The functions  $P_l^m(x)$  grow combinatorially with  $l$  and can overflow for  $l$  larger than about 150. Alternatively, one may calculate normalized associated Legendre polynomials. There are a number of different normalization conventions, and these functions can be stably computed up to degree and order 2700. The following normalizations are provided:

- Schmidt semi-normalization

Schmidt semi-normalized associated Legendre polynomials are often used in the magnetics community and are defined as

$$S_l^0(x) = P_l^0(x)$$

$$S_l^m(x) = (-1)^m \sqrt{2 \frac{(l-m)!}{(l+m)!}} P_l^m(x), m > 0$$

The factor of  $(-1)^m$  is called the Condon-Shortley phase factor and can be excluded if desired by setting the parameter `csphase = 1` in the functions below.

- Spherical Harmonic Normalization

The associated Legendre polynomials suitable for calculating spherical harmonics are defined as

$$Y_l^m(x) = (-1)^m \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} P_l^m(x)$$

where again the phase factor  $(-1)^m$  can be included or excluded if desired.

- Full Normalization

The fully normalized associated Legendre polynomials are defined as

$$N_l^m(x) = (-1)^m \sqrt{\left(l + \frac{1}{2}\right) \frac{(l-m)!}{(l+m)!}} P_l^m(x)$$

and have the property

$$\int_{-1}^1 N_l^m(x)^2 dx = 1$$

The normalized associated Legendre routines below use a recurrence relation which is stable up to a degree and order of about 2700. Beyond this, the computed functions could suffer from underflow leading to incorrect results. Routines are provided to compute first and second derivatives  $dP_l^m(x)/dx$  and  $d^2P_l^m(x)/dx^2$  as well as their alternate versions  $dP_l^m(\cos\theta)/d\theta$  and  $d^2P_l^m(\cos\theta)/d\theta^2$ . While there is a simple scaling relationship between the two forms, the derivatives involving  $\theta$  are heavily used in spherical harmonic expansions and so these routines are also provided.

In the functions below, a parameter of type *gsl\_sf\_legendre\_t* specifies the type of normalization to use. The possible values are

type `gsl_sf_legendre_t`

Value	Description
<code>GSL_SF_LEGENDRE_NONE</code>	The unnormalized associated Legendre polynomials $P_l^m(x)$
<code>GSL_SF_LEGENDRE_SCHMIDT</code>	The Schmidt semi-normalized associated Legendre polynomials $S_l^m(x)$
<code>GSL_SF_LEGENDRE_SPHARM</code>	The spherical harmonic associated Legendre polynomials $Y_l^m(x)$
<code>GSL_SF_LEGENDRE_FULL</code>	The fully normalized associated Legendre polynomials $N_l^m(x)$

int `gsl_sf_legendre_array`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, double result\_array[])

int `gsl_sf_legendre_array_e`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, const double csphase, double result\_array[])

These functions calculate all normalized associated Legendre polynomials for  $0 \leq l \leq lmax$  and  $0 \leq m \leq l$  for  $|x| \leq 1$ . The `norm` parameter specifies which normalization is used. The normalized  $P_l^m(x)$  values are stored in `result_array`, whose minimum size can be obtained from calling `gsl_sf_legendre_array_n()`. The array index of  $P_l^m(x)$  is obtained from calling `gsl_sf_legendre_array_index(1, m)`. To include or exclude the Condon-Shortley phase factor of  $(-1)^m$ , set the parameter `csphase` to either  $-1$  or  $1$  respectively in the `_e` function. This factor is excluded by default.

int `gsl_sf_legendre_deriv_array`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, double result\_array[], double result\_deriv\_array[])

int `gsl_sf_legendre_deriv_array_e`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, const double csphase, double result\_array[], double result\_deriv\_array[])

These functions calculate all normalized associated Legendre functions and their first derivatives up to degree `lmax` for  $|x| < 1$ . The parameter `norm` specifies the normalization used. The normalized  $P_l^m(x)$  values and their derivatives  $dP_l^m(x)/dx$  are stored in `result_array` and `result_deriv_array` respectively. To include or exclude the Condon-Shortley phase factor of  $(-1)^m$ , set the parameter `csphase` to either  $-1$  or  $1$  respectively in the `_e` function. This factor is excluded by default.

int `gsl_sf_legendre_deriv_alt_array`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, double result\_array[], double result\_deriv\_array[])

int `gsl_sf_legendre_deriv_alt_array_e`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, const double csphase, double result\_array[], double result\_deriv\_array[])

These functions calculate all normalized associated Legendre functions and their (alternate) first derivatives up to degree `lmax` for  $|x| < 1$ . The normalized  $P_l^m(x)$  values and their derivatives  $dP_l^m(\cos \theta)/d\theta$  are stored in `result_array` and `result_deriv_array` respectively. To include or exclude the Condon-Shortley phase factor of  $(-1)^m$ , set the parameter `csphase` to either  $-1$  or  $1$  respectively in the `_e` function. This factor is excluded by default.

int `gsl_sf_legendre_deriv2_array`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, double result\_array[], double result\_deriv\_array[], double result\_deriv2\_array[])

int `gsl_sf_legendre_deriv2_array_e`(const `gsl_sf_legendre_t` norm, const size\_t lmax, const double x, const double csphase, double result\_array[], double result\_deriv\_array[], double result\_deriv2\_array[])

These functions calculate all normalized associated Legendre functions and their first and second derivatives up to degree `lmax` for  $|x| < 1$ . The parameter `norm` specifies the normalization used. The normalized  $P_l^m(x)$ , their first derivatives  $dP_l^m(x)/dx$ , and their second derivatives  $d^2P_l^m(x)/dx^2$  are stored in `result_array`, `result_deriv_array`, and `result_deriv2_array` respectively. To include or exclude the Condon-Shortley

phase factor of  $(-1)^m$ , set the parameter *csphase* to either  $-1$  or  $1$  respectively in the *\_e* function. This factor is excluded by default.

```
int gsl_sf_legendre_deriv2_alt_array(const gsl_sf_legendre_t norm, const size_t lmax, const double x,
                                     double result_array[], double result_deriv_array[], double
                                     result_deriv2_array[])
```

```
int gsl_sf_legendre_deriv2_alt_array_e(const gsl_sf_legendre_t norm, const size_t lmax, const double x,
                                       const double csphase, double result_array[], double
                                       result_deriv_array[], double result_deriv2_array[])
```

These functions calculate all normalized associated Legendre functions and their (alternate) first and second derivatives up to degree *lmax* for  $|x| < 1$ . The parameter *norm* specifies the normalization used. The normalized  $P_l^m(x)$ , their first derivatives  $dP_l^m(\cos \theta)/d\theta$ , and their second derivatives  $d^2P_l^m(\cos \theta)/d\theta^2$  are stored in *result\_array*, *result\_deriv\_array*, and *result\_deriv2\_array* respectively. To include or exclude the Condon-Shortley phase factor of  $(-1)^m$ , set the parameter *csphase* to either  $-1$  or  $1$  respectively in the *\_e* function. This factor is excluded by default.

```
size_t gsl_sf_legendre_nlm(const size_t lmax)
```

This function returns the total number of associated Legendre functions  $P_l^m(x)$  for a given *lmax*. The number is  $(lmax+1) * (lmax+2) / 2$ .

```
size_t gsl_sf_legendre_array_n(const size_t lmax)
```

This function returns the minimum array size for maximum degree *lmax* needed for the array versions of the associated Legendre functions. Size is calculated as the total number of  $P_l^m(x)$  functions (see *gsl\_sf\_legendre\_nlm()*), plus extra space for precomputing multiplicative factors used in the recurrence relations.

```
size_t gsl_sf_legendre_array_index(const size_t l, const size_t m)
```

This function returns the index into *result\_array*, *result\_deriv\_array*, or *result\_deriv2\_array* corresponding to  $P_l^m(x)$ ,  $P_l^m(x)$ , or  $P_l^m(x)$ . The index is given by  $l(l+1)/2 + m$ .

An inline version of this function is used if *HAVE\_INLINE* is defined.

```
double gsl_sf_legendre_Plm(int l, int m, double x)
```

```
int gsl_sf_legendre_Plm_e(int l, int m, double x, gsl_sf_result *result)
```

These routines compute the associated Legendre polynomial  $P_l^m(x)$  for  $m \geq 0$ ,  $l \geq m$ , and  $|x| \leq 1$ .

```
double gsl_sf_legendre_sphPlm(int l, int m, double x)
```

```
int gsl_sf_legendre_sphPlm_e(int l, int m, double x, gsl_sf_result *result)
```

These routines compute the normalized associated Legendre polynomial  $\sqrt{(2l+1)/(4\pi)}\sqrt{(l-m)!/(l+m)!}P_l^m(x)$  suitable for use in spherical harmonics. The parameters must satisfy  $m \geq 0$ ,  $l \geq m$ , and  $|x| \leq 1$ . These routines avoid the overflows that occur for the standard normalization of  $P_l^m(x)$ .

```
int gsl_sf_legendre_Plm_array(int lmax, int m, double x, double result_array[])
```

```
int gsl_sf_legendre_Plm_deriv_array(int lmax, int m, double x, double result_array[], double
                                     result_deriv_array[])
```

These functions are now deprecated and will be removed in a future release; see *gsl\_sf\_legendre\_array()* and *gsl\_sf\_legendre\_deriv\_array()*.

```
int gsl_sf_legendre_sphPlm_array(int lmax, int m, double x, double result_array[])
```

int **gsl\_sf\_legendre\_sphPlm\_deriv\_array**(int lmax, int m, double x, double result\_array[], double result\_deriv\_array[])

These functions are now deprecated and will be removed in a future release; see *gsl\_sf\_legendre\_array()* and *gsl\_sf\_legendre\_deriv\_array()*.

int **gsl\_sf\_legendre\_array\_size**(const int lmax, const int m)

This function is now deprecated and will be removed in a future release.

### 7.25.3 Conical Functions

The Conical Functions  $P_{-(1/2)+i\lambda}^{\mu}(x)$  and  $Q_{-(1/2)+i\lambda}^{\mu}$  are described in Abramowitz & Stegun, Section 8.12.

double **gsl\_sf\_conicalP\_half**(double lambda, double x)

int **gsl\_sf\_conicalP\_half\_e**(double lambda, double x, *gsl\_sf\_result* \*result)

These routines compute the irregular Spherical Conical Function  $P_{-1/2+i\lambda}^{1/2}(x)$  for  $x > -1$ .

double **gsl\_sf\_conicalP\_mhalf**(double lambda, double x)

int **gsl\_sf\_conicalP\_mhalf\_e**(double lambda, double x, *gsl\_sf\_result* \*result)

These routines compute the regular Spherical Conical Function  $P_{-1/2+i\lambda}^{-1/2}(x)$  for  $x > -1$ .

double **gsl\_sf\_conicalP\_0**(double lambda, double x)

int **gsl\_sf\_conicalP\_0\_e**(double lambda, double x, *gsl\_sf\_result* \*result)

These routines compute the conical function  $P_{-1/2+i\lambda}^0(x)$  for  $x > -1$ .

double **gsl\_sf\_conicalP\_1**(double lambda, double x)

int **gsl\_sf\_conicalP\_1\_e**(double lambda, double x, *gsl\_sf\_result* \*result)

These routines compute the conical function  $P_{-1/2+i\lambda}^1(x)$  for  $x > -1$ .

double **gsl\_sf\_conicalP\_sph\_reg**(int l, double lambda, double x)

int **gsl\_sf\_conicalP\_sph\_reg\_e**(int l, double lambda, double x, *gsl\_sf\_result* \*result)

These routines compute the Regular Spherical Conical Function  $P_{-1/2+i\lambda}^{-1/2-l}(x)$  for  $x > -1$  and  $l \geq -1$ .

double **gsl\_sf\_conicalP\_cyl\_reg**(int m, double lambda, double x)

int **gsl\_sf\_conicalP\_cyl\_reg\_e**(int m, double lambda, double x, *gsl\_sf\_result* \*result)

These routines compute the Regular Cylindrical Conical Function  $P_{-1/2+i\lambda}^{-m}(x)$  for  $x > -1$  and  $m \geq -1$ .

### 7.25.4 Radial Functions for Hyperbolic Space

The following spherical functions are specializations of Legendre functions which give the regular eigenfunctions of the Laplacian on a 3-dimensional hyperbolic space  $H^3$ . Of particular interest is the flat limit,  $\lambda \rightarrow \infty$ ,  $\eta \rightarrow 0$ ,  $\lambda\eta$  fixed.

double **gsl\_sf\_legendre\_H3d\_0**(double lambda, double eta)

int **gsl\_sf\_legendre\_H3d\_0\_e**(double lambda, double eta, *gsl\_sf\_result* \*result)

These routines compute the zeroth radial eigenfunction of the Laplacian on the 3-dimensional hyperbolic space,

$$L_0^{H3d}(\lambda, \eta) := \frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)}$$

for  $\eta \geq 0$ . In the flat limit this takes the form  $L_0^{H3d}(\lambda, \eta) = j_0(\lambda\eta)$ .

double **gsl\_sf\_legendre\_H3d\_1**(double lambda, double eta)

int **gsl\_sf\_legendre\_H3d\_1\_e**(double lambda, double eta, *gsl\_sf\_result* \*result)

These routines compute the first radial eigenfunction of the Laplacian on the 3-dimensional hyperbolic space,

$$L_1^{H3d}(\lambda, \eta) := \frac{1}{\sqrt{\lambda^2 + 1}} \left( \frac{\sin(\lambda\eta)}{\lambda \sinh(\eta)} \right) (\coth(\eta) - \lambda \cot(\lambda\eta))$$

for  $\eta \geq 0$  In the flat limit this takes the form  $L_1^{H3d}(\lambda, \eta) = j_1(\lambda\eta)$ .

double **gsl\_sf\_legendre\_H3d**(int l, double lambda, double eta)

int **gsl\_sf\_legendre\_H3d\_e**(int l, double lambda, double eta, *gsl\_sf\_result* \*result)

These routines compute the  $l$ -th radial eigenfunction of the Laplacian on the 3-dimensional hyperbolic space  $\eta \geq 0$  and  $l \geq 0$ . In the flat limit this takes the form  $L_l^{H3d}(\lambda, \eta) = j_l(\lambda\eta)$ .

int **gsl\_sf\_legendre\_H3d\_array**(int lmax, double lambda, double eta, double result\_array[])

This function computes an array of radial eigenfunctions  $L_l^{H3d}(\lambda, \eta)$  for  $0 \leq l \leq lmax$ .

## 7.26 Logarithm and Related Functions

Information on the properties of the Logarithm function can be found in Abramowitz & Stegun, Chapter 4. The functions described in this section are declared in the header file `gsl_sf_log.h`.

double **gsl\_sf\_log**(double x)

int **gsl\_sf\_log\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the logarithm of  $x$ ,  $\log(x)$ , for  $x > 0$ .

double **gsl\_sf\_log\_abs**(double x)

int **gsl\_sf\_log\_abs\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the logarithm of the magnitude of  $x$ ,  $\log(|x|)$ , for  $x \neq 0$ .

int **gsl\_sf\_complex\_log\_e**(double zr, double zi, *gsl\_sf\_result* \*lnr, *gsl\_sf\_result* \*theta)

This routine computes the complex logarithm of  $z = z_r + iz_i$ . The results are returned as *lnr*, *theta* such that  $\exp(\lnr + i\theta) = z_r + iz_i$ , where  $\theta$  lies in the range  $[-\pi, \pi]$ .

double **gsl\_sf\_log\_1plusx**(double x)

int **gsl\_sf\_log\_1plusx\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute  $\log(1+x)$  for  $x > -1$  using an algorithm that is accurate for small  $x$ .

double **gsl\_sf\_log\_1plusx\_mx**(double x)

int **gsl\_sf\_log\_1plusx\_mx\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute  $\log(1+x) - x$  for  $x > -1$  using an algorithm that is accurate for small  $x$ .

## 7.27 Mathieu Functions

The routines described in this section compute the angular and radial Mathieu functions, and their characteristic values. Mathieu functions are the solutions of the following two differential equations:

$$\frac{d^2 y}{dv^2} + (a - 2q \cos 2v)y = 0$$

$$\frac{d^2 f}{du^2} - (a - 2q \cosh 2u)f = 0$$

The angular Mathieu functions  $ce_r(x, q)$ ,  $se_r(x, q)$  are the even and odd periodic solutions of the first equation, which is known as Mathieu's equation. These exist only for the discrete sequence of characteristic values  $a = a_r(q)$  (even-periodic) and  $a = b_r(q)$  (odd-periodic).

The radial Mathieu functions  $Mc_r^{(j)}(z, q)$  and  $Ms_r^{(j)}(z, q)$  are the solutions of the second equation, which is referred to as Mathieu's modified equation. The radial Mathieu functions of the first, second, third and fourth kind are denoted by the parameter  $j$ , which takes the value 1, 2, 3 or 4.

For more information on the Mathieu functions, see Abramowitz and Stegun, Chapter 20. These functions are defined in the header file `gsl_sf_mathieu.h`.

### 7.27.1 Mathieu Function Workspace

The Mathieu functions can be computed for a single order or for multiple orders, using array-based routines. The array-based routines require a preallocated workspace.

type **gsl\_sf\_mathieu\_workspace**

Workspace required for array-based routines

`gsl_sf_mathieu_workspace *gsl_sf_mathieu_alloc(size_t n, double qmax)`

This function returns a workspace for the array versions of the Mathieu routines. The arguments `n` and `qmax` specify the maximum order and  $q$ -value of Mathieu functions which can be computed with this workspace.

void **gsl\_sf\_mathieu\_free**(`gsl_sf_mathieu_workspace *work`)

This function frees the workspace `work`.

### 7.27.2 Mathieu Function Characteristic Values

int **gsl\_sf\_mathieu\_a**(int `n`, double `q`)

int **gsl\_sf\_mathieu\_a\_e**(int `n`, double `q`, `gsl_sf_result *result`)

int **gsl\_sf\_mathieu\_b**(int `n`, double `q`)

int **gsl\_sf\_mathieu\_b\_e**(int `n`, double `q`, `gsl_sf_result *result`)

These routines compute the characteristic values  $a_n(q)$ ,  $b_n(q)$  of the Mathieu functions  $ce_n(q, x)$  and  $se_n(q, x)$ , respectively.

int **gsl\_sf\_mathieu\_a\_array**(int `order_min`, int `order_max`, double `q`, `gsl_sf_mathieu_workspace *work`, double `result_array[]`)

int **gsl\_sf\_mathieu\_b\_array**(int `order_min`, int `order_max`, double `q`, `gsl_sf_mathieu_workspace *work`, double `result_array[]`)

These routines compute a series of Mathieu characteristic values  $a_n(q)$ ,  $b_n(q)$  for  $n$  from `order_min` to `order_max` inclusive, storing the results in the array `result_array`.



### 7.27.3 Angular Mathieu Functions

int **gsl\_sf\_mathieu\_ce**(int n, double q, double x)

int **gsl\_sf\_mathieu\_ce\_e**(int n, double q, double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_mathieu\_se**(int n, double q, double x)

int **gsl\_sf\_mathieu\_se\_e**(int n, double q, double x, *gsl\_sf\_result* \*result)

These routines compute the angular Mathieu functions  $ce_n(q, x)$  and  $se_n(q, x)$ , respectively.

int **gsl\_sf\_mathieu\_ce\_array**(int nmin, int nmax, double q, double x, *gsl\_sf\_mathieu\_workspace* \*work, double result\_array[])

int **gsl\_sf\_mathieu\_se\_array**(int nmin, int nmax, double q, double x, *gsl\_sf\_mathieu\_workspace* \*work, double result\_array[])

These routines compute a series of the angular Mathieu functions  $ce_n(q, x)$  and  $se_n(q, x)$  of order  $n$  from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*.

### 7.27.4 Radial Mathieu Functions

int **gsl\_sf\_mathieu\_Mc**(int j, int n, double q, double x)

int **gsl\_sf\_mathieu\_Mc\_e**(int j, int n, double q, double x, *gsl\_sf\_result* \*result)

int **gsl\_sf\_mathieu\_Ms**(int j, int n, double q, double x)

int **gsl\_sf\_mathieu\_Ms\_e**(int j, int n, double q, double x, *gsl\_sf\_result* \*result)

These routines compute the radial  $j$ -th kind Mathieu functions  $Mc_n^{(j)}(q, x)$  and  $Ms_n^{(j)}(q, x)$  of order  $n$ .

The allowed values of  $j$  are 1 and 2. The functions for  $j = 3, 4$  can be computed as  $M_n^{(3)} = M_n^{(1)} + iM_n^{(2)}$  and  $M_n^{(4)} = M_n^{(1)} - iM_n^{(2)}$ , where  $M_n^{(j)} = Mc_n^{(j)}$  or  $Ms_n^{(j)}$ .

int **gsl\_sf\_mathieu\_Mc\_array**(int j, int nmin, int nmax, double q, double x, *gsl\_sf\_mathieu\_workspace* \*work, double result\_array[])

int **gsl\_sf\_mathieu\_Ms\_array**(int j, int nmin, int nmax, double q, double x, *gsl\_sf\_mathieu\_workspace* \*work, double result\_array[])

These routines compute a series of the radial Mathieu functions of kind  $j$ , with order from *nmin* to *nmax* inclusive, storing the results in the array *result\_array*.

## 7.28 Power Function

The following functions are equivalent to the function *gsl\_pow\_int()* with an error estimate. These functions are declared in the header file *gsl\_sf\_pow\_int.h*.

double **gsl\_sf\_pow\_int**(double x, int n)

int **gsl\_sf\_pow\_int\_e**(double x, int n, *gsl\_sf\_result* \*result)

These routines compute the power  $x^n$  for integer  $n$ . The power is computed using the minimum number of multiplications. For example,  $x^8$  is computed as  $((x^2)^2)^2$ , requiring only 3 multiplications. For reasons of efficiency, these functions do not check for overflow or underflow conditions. The following is a simple example:

```
#include <gsl/gsl_sf_pow_int.h>
/* compute 3.0**12 */
double y = gsl_sf_pow_int(3.0, 12);
```

## 7.29 Psi (Digamma) Function

The polygamma functions of order  $n$  are defined by

$$\psi^{(n)}(x) = \left(\frac{d}{dx}\right)^n \psi(x) = \left(\frac{d}{dx}\right)^{n+1} \log(\Gamma(x))$$

where  $\psi(x) = \Gamma'(x)/\Gamma(x)$  is known as the digamma function. These functions are declared in the header file `gsl_sf_psi.h`.

### 7.29.1 Digamma Function

double **gsl\_sf\_psi\_int**(int n)

int **gsl\_sf\_psi\_int\_e**(int n, *gsl\_sf\_result* \*result)

These routines compute the digamma function  $\psi(n)$  for positive integer  $n$ . The digamma function is also called the Psi function.

double **gsl\_sf\_psi**(double x)

int **gsl\_sf\_psi\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the digamma function  $\psi(x)$  for general  $x$ ,  $x \neq 0$ .

double **gsl\_sf\_psi\_1piy**(double y)

int **gsl\_sf\_psi\_1piy\_e**(double y, *gsl\_sf\_result* \*result)

These routines compute the real part of the digamma function on the line  $1 + iy$ ,  $\Re[\psi(1 + iy)]$ .

### 7.29.2 Trigamma Function

double **gsl\_sf\_psi\_1\_int**(int n)

int **gsl\_sf\_psi\_1\_int\_e**(int n, *gsl\_sf\_result* \*result)

These routines compute the Trigamma function  $\psi'(n)$  for positive integer  $n$ .

double **gsl\_sf\_psi\_1**(double x)

int **gsl\_sf\_psi\_1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the Trigamma function  $\psi'(x)$  for general  $x$ .

### 7.29.3 Polygamma Function

double **gsl\_sf\_psi\_n**(int n, double x)

int **gsl\_sf\_psi\_n\_e**(int n, double x, *gsl\_sf\_result* \*result)

These routines compute the polygamma function  $\psi^{(n)}(x)$  for  $n \geq 0$ ,  $x > 0$ .

## 7.30 Synchrotron Functions

The functions described in this section are declared in the header file `gsl_sf_synchrotron.h`.

double **gsl\_sf\_synchrotron\_1**(double x)

int **gsl\_sf\_synchrotron\_1\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the first synchrotron function  $x \int_x^\infty dt K_{5/3}(t)$  for  $x \geq 0$ .

double **gsl\_sf\_synchrotron\_2**(double x)

int **gsl\_sf\_synchrotron\_2\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the second synchrotron function  $x K_{2/3}(x)$  for  $x \geq 0$ .

## 7.31 Transport Functions

The transport functions  $J(n, x)$  are defined by the integral representations

$$J(n, x) = \int_0^x t^n e^t / (e^t - 1)^2 dt$$

They are declared in the header file `gsl_sf_transport.h`.

double **gsl\_sf\_transport\_2**(double x)

int **gsl\_sf\_transport\_2\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the transport function  $J(2, x)$ .

double **gsl\_sf\_transport\_3**(double x)

int **gsl\_sf\_transport\_3\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the transport function  $J(3, x)$ .

double **gsl\_sf\_transport\_4**(double x)

int **gsl\_sf\_transport\_4\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the transport function  $J(4, x)$ .

double **gsl\_sf\_transport\_5**(double x)

int **gsl\_sf\_transport\_5\_e**(double x, *gsl\_sf\_result* \*result)

These routines compute the transport function  $J(5, x)$ .

## 7.32 Trigonometric Functions

The library includes its own trigonometric functions in order to provide consistency across platforms and reliable error estimates. These functions are declared in the header file `gsl_sf_trig.h`.

### 7.32.1 Circular Trigonometric Functions

double `gsl_sf_sin`(double *x*)

int `gsl_sf_sin_e`(double *x*, *gsl\_sf\_result* \**result*)  
These routines compute the sine function  $\sin(x)$ .

double `gsl_sf_cos`(double *x*)

int `gsl_sf_cos_e`(double *x*, *gsl\_sf\_result* \**result*)  
These routines compute the cosine function  $\cos(x)$ .

double `gsl_sf_hypot`(double *x*, double *y*)

int `gsl_sf_hypot_e`(double *x*, double *y*, *gsl\_sf\_result* \**result*)  
These routines compute the hypotenuse function  $\sqrt{x^2 + y^2}$  avoiding overflow and underflow.

double `gsl_sf_sinc`(double *x*)

int `gsl_sf_sinc_e`(double *x*, *gsl\_sf\_result* \**result*)  
These routines compute  $\text{sinc}(x) = \sin(\pi x)/(\pi x)$  for any value of *x*.

### 7.32.2 Trigonometric Functions for Complex Arguments

int `gsl_sf_complex_sin_e`(double *zr*, double *zi*, *gsl\_sf\_result* \**szr*, *gsl\_sf\_result* \**sz\_i*)  
This function computes the complex sine,  $\sin(z_r + iz_i)$  storing the real and imaginary parts in *szr*, *sz\_i*.

int `gsl_sf_complex_cos_e`(double *zr*, double *zi*, *gsl\_sf\_result* \**czr*, *gsl\_sf\_result* \**czi*)  
This function computes the complex cosine,  $\cos(z_r + iz_i)$  storing the real and imaginary parts in *czr*, *czi*.

int `gsl_sf_complex_logsin_e`(double *zr*, double *zi*, *gsl\_sf\_result* \**lszr*, *gsl\_sf\_result* \**lsz\_i*)  
This function computes the logarithm of the complex sine,  $\log(\sin(z_r + iz_i))$  storing the real and imaginary parts in *lszr*, *lsz\_i*.

### 7.32.3 Hyperbolic Trigonometric Functions

double `gsl_sf_lnsinh`(double *x*)

int `gsl_sf_lnsinh_e`(double *x*, *gsl\_sf\_result* \**result*)  
These routines compute  $\log(\sinh(x))$  for  $x > 0$ .

double `gsl_sf_lncosh`(double *x*)

int `gsl_sf_lncosh_e`(double *x*, *gsl\_sf\_result* \**result*)  
These routines compute  $\log(\cosh(x))$  for any *x*.

### 7.32.4 Conversion Functions

int **gsl\_sf\_polar\_to\_rect**(double r, double theta, *gsl\_sf\_result* \*x, *gsl\_sf\_result* \*y)

This function converts the polar coordinates ( $r$ ,  $\theta$ ) to rectilinear coordinates ( $x$ ,  $y$ ),  $x = r \cos(\theta)$ ,  $y = r \sin(\theta)$ .

int **gsl\_sf\_rect\_to\_polar**(double x, double y, *gsl\_sf\_result* \*r, *gsl\_sf\_result* \*theta)

This function converts the rectilinear coordinates ( $x$ ,  $y$ ) to polar coordinates ( $r$ ,  $\theta$ ), such that  $x = r \cos(\theta)$ ,  $y = r \sin(\theta)$ . The argument  $\theta$  lies in the range  $[-\pi, \pi]$ .

### 7.32.5 Restriction Functions

double **gsl\_sf\_angle\_restrict\_symm**(double theta)

int **gsl\_sf\_angle\_restrict\_symm\_e**(double \*theta)

These routines force the angle  $\theta$  to lie in the range  $(-\pi, \pi]$ .

Note that the mathematical value of  $\pi$  is slightly greater than `M_PI`, so the machine numbers `M_PI` and `-M_PI` are included in the range.

double **gsl\_sf\_angle\_restrict\_pos**(double theta)

int **gsl\_sf\_angle\_restrict\_pos\_e**(double \*theta)

These routines force the angle  $\theta$  to lie in the range  $[0, 2\pi)$ .

Note that the mathematical value of  $2\pi$  is slightly greater than `2*M_PI`, so the machine number `2*M_PI` is included in the range.

### 7.32.6 Trigonometric Functions With Error Estimates

int **gsl\_sf\_sin\_err\_e**(double x, double dx, *gsl\_sf\_result* \*result)

This routine computes the sine of an angle  $x$  with an associated absolute error  $dx$ ,  $\sin(x \pm dx)$ . Note that this function is provided in the error-handling form only since its purpose is to compute the propagated error.

int **gsl\_sf\_cos\_err\_e**(double x, double dx, *gsl\_sf\_result* \*result)

This routine computes the cosine of an angle  $x$  with an associated absolute error  $dx$ ,  $\cos(x \pm dx)$ . Note that this function is provided in the error-handling form only since its purpose is to compute the propagated error.

## 7.33 Zeta Functions

The Riemann zeta function is defined in Abramowitz & Stegun, Section 23.2. The functions described in this section are declared in the header file `gsl_sf_zeta.h`.

### 7.33.1 Riemann Zeta Function

The Riemann zeta function is defined by the infinite sum

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$$

double **gsl\_sf\_zeta\_int**(int n)

int **gsl\_sf\_zeta\_int\_e**(int n, *gsl\_sf\_result* \*result)

These routines compute the Riemann zeta function  $\zeta(n)$  for integer  $n$ ,  $n \neq 1$ .

double **gsl\_sf\_zeta**(double s)

int **gsl\_sf\_zeta\_e**(double s, *gsl\_sf\_result* \*result)

These routines compute the Riemann zeta function  $\zeta(s)$  for arbitrary  $s$ ,  $s \neq 1$ .

### 7.33.2 Riemann Zeta Function Minus One

For large positive argument, the Riemann zeta function approaches one. In this region the fractional part is interesting, and therefore we need a function to evaluate it explicitly.

double **gsl\_sf\_zetam1\_int**(int n)

int **gsl\_sf\_zetam1\_int\_e**(int n, *gsl\_sf\_result* \*result)

These routines compute  $\zeta(n) - 1$  for integer  $n$ ,  $n \neq 1$ .

double **gsl\_sf\_zetam1**(double s)

int **gsl\_sf\_zetam1\_e**(double s, *gsl\_sf\_result* \*result)

These routines compute  $\zeta(s) - 1$  for arbitrary  $s$ ,  $s \neq 1$ .

### 7.33.3 Hurwitz Zeta Function

The Hurwitz zeta function is defined by

$$\zeta(s, q) = \sum_{k=0}^{\infty} (k + q)^{-s}$$

double **gsl\_sf\_hzeta**(double s, double q)

int **gsl\_sf\_hzeta\_e**(double s, double q, *gsl\_sf\_result* \*result)

These routines compute the Hurwitz zeta function  $\zeta(s, q)$  for  $s > 1$ ,  $q > 0$ .

### 7.33.4 Eta Function

The eta function is defined by

$$\eta(s) = (1 - 2^{1-s})\zeta(s)$$

double **gsl\_sf\_eta\_int**(int n)

int **gsl\_sf\_eta\_int\_e**(int n, *gsl\_sf\_result* \*result)

These routines compute the eta function  $\eta(n)$  for integer  $n$ .

double **gsl\_sf\_eta**(double s)

int **gsl\_sf\_eta\_e**(double s, *gsl\_sf\_result* \*result)

These routines compute the eta function  $\eta(s)$  for arbitrary  $s$ .

## 7.34 Examples

The following example demonstrates the use of the error handling form of the special functions, in this case to compute the Bessel function  $J_0(5.0)$ ,

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
    double x = 5.0;
    gsl_sf_result result;

    double expected = -0.17759677131433830434739701;

    int status = gsl_sf_bessel_J0_e (x, &result);

    printf ("status = %s\n", gsl_strerror(status));
    printf ("J0(5.0) = %.18f\n"
           "      +/- %.18f\n",
           result.val, result.err);
    printf ("exact   = %.18f\n", expected);
    return status;
}
```

Here are the results of running the program,

```
status = success
J0(5.0) = -0.177596771314338264
      +/- 0.0000000000000000193
exact   = -0.177596771314338292
```

The next program computes the same quantity using the natural form of the function. In this case the error term `result.err` and return status are not accessible.

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
  double x = 5.0;
  double expected = -0.17759677131433830434739701;

  double y = gsl_sf_bessel_J0 (x);

  printf ("J0(5.0) = %.18f\n", y);
  printf ("exact   = %.18f\n", expected);
  return 0;
}
```

The results of the function are the same,

```
J0(5.0) = -0.177596771314338264
exact   = -0.177596771314338292
```

## 7.35 References and Further Reading

The library follows the conventions of the following book where possible,

- Handbook of Mathematical Functions, edited by Abramowitz & Stegun, Dover, ISBN 0486612724.

The following papers contain information on the algorithms used to compute the special functions,

- Allan J. MacLeod, MISCFUN: A software package to compute uncommon special functions. ACM Trans. Math. Soft., vol.: 22, 1996, 288–301
- Bunck, B. F., A fast algorithm for evaluation of normalized Hermite functions, BIT Numer. Math, 49: 281-295, 2009.
- G.N. Watson, A Treatise on the Theory of Bessel Functions, 2nd Edition (Cambridge University Press, 1944).
- G. Nemeth, Mathematical Approximations of Special Functions, Nova Science Publishers, ISBN 1-56072-052-2
- B.C. Carlson, Special Functions of Applied Mathematics (1977)
- N. M. Temme, Special Functions: An Introduction to the Classical Functions of Mathematical Physics (1996), ISBN 978-0471113133.
- W.J. Thompson, Atlas for Computing Mathematical Functions, John Wiley & Sons, New York (1997).
- Y.Y. Luke, Algorithms for the Computation of Mathematical Functions, Academic Press, New York (1977).
- S. A. Holmes and W. E. Featherstone, A unified approach to the Clenshaw summation and the recursive computation of very high degree and order normalised associated Legendre functions, Journal of Geodesy, 76, pg. 279-299, 2002.



## VECTORS AND MATRICES

The functions described in this chapter provide a simple vector and matrix interface to ordinary C arrays. The memory management of these arrays is implemented using a single underlying type, known as a block. By writing your functions in terms of vectors and matrices you can pass a single structure containing both data and dimensions as an argument without needing additional function parameters. The structures are compatible with the vector and matrix formats used by BLAS routines.

### 8.1 Data types

All the functions are available for each of the standard data-types. The versions for double have the prefix `gsl_block`, `gsl_vector` and `gsl_matrix`. Similarly the versions for single-precision float arrays have the prefix `gsl_block_float`, `gsl_vector_float` and `gsl_matrix_float`. The full list of available types is given below,

Prefix	Type
<code>gsl_block</code>	double
<code>gsl_block_float</code>	float
<code>gsl_block_long_double</code>	long double
<code>gsl_block_int</code>	int
<code>gsl_block_uint</code>	unsigned int
<code>gsl_block_long</code>	long
<code>gsl_block_ulong</code>	unsigned long
<code>gsl_block_short</code>	short
<code>gsl_block_ushort</code>	unsigned short
<code>gsl_block_char</code>	char
<code>gsl_block_uchar</code>	unsigned char
<code>gsl_block_complex</code>	complex double
<code>gsl_block_complex_float</code>	complex float
<code>gsl_block_complex_long_double</code>	complex long double

Corresponding types exist for the `gsl_vector` and `gsl_matrix` functions.

## 8.2 Blocks

For consistency all memory is allocated through a `gsl_block` structure. The structure contains two components, the size of an area of memory and a pointer to the memory. The `gsl_block` structure looks like this,

type `gsl_block`

```
typedef struct
{
    size_t size;
    double * data;
} gsl_block;
```

Vectors and matrices are made by *slicing* an underlying block. A slice is a set of elements formed from an initial offset and a combination of indices and step-sizes. In the case of a matrix the step-size for the column index represents the row-length. The step-size for a vector is known as the *stride*.

The functions for allocating and deallocating blocks are defined in `gsl_block.h`.

### 8.2.1 Block allocation

The functions for allocating memory to a block follow the style of `malloc` and `free`. In addition they also perform their own error checking. If there is insufficient memory available to allocate a block then the functions call the GSL error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every `alloc`.

`gsl_block *gsl_block_alloc(size_t n)`

This function allocates memory for a block of `n` double-precision elements, returning a pointer to the block struct. The block is not initialized and so the values of its elements are undefined. Use the function `gsl_block_calloc()` if you want to ensure that all the elements are initialized to zero.

Zero-sized requests are valid and return a non-null result. A null pointer is returned if insufficient memory is available to create the block.

`gsl_block *gsl_block_calloc(size_t n)`

This function allocates memory for a block and initializes all the elements of the block to zero.

void `gsl_block_free(gsl_block *b)`

This function frees the memory used by a block `b` previously allocated with `gsl_block_alloc()` or `gsl_block_calloc()`.

### 8.2.2 Reading and writing blocks

The library provides functions for reading and writing blocks to a file as binary data or formatted text.

int `gsl_block_fwrite(FILE *stream, const gsl_block *b)`

This function writes the elements of the block `b` to the stream `stream` in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int `gsl_block_fread(FILE *stream, gsl_block *b)`

This function reads into the block `b` from the open stream `stream` in binary format. The block `b` must be preallocated with the correct length since the function uses the size of `b` to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl\_block\_fprintf**(FILE \*stream, const *gsl\_block* \*b, const char \*format)

This function writes the elements of the block *b* line-by-line to the stream *stream* using the format specifier *format*, which should be one of the %g, %e or %f formats for floating point numbers and %d for integers. The function returns 0 for success and GSL\_EFAILED if there was a problem writing to the file.

int **gsl\_block\_fscanf**(FILE \*stream, *gsl\_block* \*b)

This function reads formatted data from the stream *stream* into the block *b*. The block *b* must be preallocated with the correct length since the function uses the size of *b* to determine how many numbers to read. The function returns 0 for success and GSL\_EFAILED if there was a problem reading from the file.

### 8.2.3 Example programs for blocks

The following program shows how to allocate a block,

```
#include <stdio.h>
#include <gsl/gsl_block.h>

int
main (void)
{
    gsl_block * b = gsl_block_alloc (100);

    printf ("length of block = %zu\n", b->size);
    printf ("block data address = %p\n", b->data);

    gsl_block_free (b);
    return 0;
}
```

Here is the output from the program,

```
length of block = 100
block data address = 0x804b0d8
```

## 8.3 Vectors

Vectors are defined by a *gsl\_vector* structure which describes a slice of a block. Different vectors can be created which point to the same block. A vector slice is a set of equally-spaced elements of an area of memory.

The *gsl\_vector* structure contains five components, the *size*, the *stride*, a pointer to the memory where the elements are stored, *data*, a pointer to the block owned by the vector, *block*, if any, and an ownership flag, *owner*. The structure is very simple and looks like this,

type **gsl\_vector**

```
typedef struct
{
    size_t size;
    size_t stride;
    double * data;
    gsl_block * block;
```

(continues on next page)

```

    int owner;
} gsl_vector;

```

The `size` is simply the number of vector elements. The range of valid indices runs from 0 to `size-1`. The `stride` is the step-size from one element to the next in physical memory, measured in units of the appropriate datatype. The pointer `data` gives the location of the first element of the vector in memory. The pointer `block` stores the location of the memory block in which the vector elements are located (if any). If the vector owns this block then the `owner` field is set to one and the block will be deallocated when the vector is freed. If the vector points to a block owned by another object then the `owner` field is zero and any underlying block will not be deallocated with the vector.

The functions for allocating and accessing vectors are defined in `gsl_vector.h`.

### 8.3.1 Vector allocation

The functions for allocating memory to a vector follow the style of `malloc` and `free`. In addition they also perform their own error checking. If there is insufficient memory available to allocate a vector then the functions call the GSL error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every `alloc`.

*gsl\_vector* \***gsl\_vector\_alloc**(size\_t n)

This function creates a vector of length `n`, returning a pointer to a newly initialized vector struct. A new block is allocated for the elements of the vector, and stored in the `block` component of the vector struct. The block is “owned” by the vector, and will be deallocated when the vector is deallocated. Zero-sized requests are valid and return a non-null result.

*gsl\_vector* \***gsl\_vector\_calloc**(size\_t n)

This function allocates memory for a vector of length `n` and initializes all the elements of the vector to zero.

void **gsl\_vector\_free**(*gsl\_vector* \*v)

This function frees a previously allocated vector `v`. If the vector was created using `gsl_vector_alloc()` then the block underlying the vector will also be deallocated. If the vector has been created from another object then the memory is still owned by that object and will not be deallocated.

### 8.3.2 Accessing vector elements

Unlike Fortran compilers, C compilers do not usually provide support for range checking of vectors and matrices.<sup>1</sup> The functions `gsl_vector_get()` and `gsl_vector_set()` can perform portable range checking for you and report an error if you attempt to access elements outside the allowed range.

The functions for accessing the elements of a vector or matrix are defined in `gsl_vector.h` and declared `extern inline` to eliminate function-call overhead. You must compile your program with the preprocessor macro `HAVE_INLINE` defined to use these functions.

#### **GSL\_RANGE\_CHECK\_OFF**

If necessary you can turn off range checking completely without modifying any source files by recompiling your program with the preprocessor definition `GSL_RANGE_CHECK_OFF`. Provided your compiler supports inline functions the effect of turning off range checking is to replace calls to `gsl_vector_get(v,i)` by `v->data[i*v->stride]` and calls to `gsl_vector_set(v,i,x)` by `v->data[i*v->stride]=x`. Thus there should be no performance penalty for using the range checking functions when range checking is turned off.

#### **GSL\_C99\_INLINE**

If you use a C99 compiler which requires inline functions in header files to be declared `inline` instead of `extern`

<sup>1</sup> Range checking is available in the GNU C Compiler bounds-checking extension, but it is not part of the default installation of GCC. Memory accesses can also be checked with Valgrind or the `gcc -fmadflap` memory protection option.

inline, define the macro `GSL_C99_INLINE` (see *Inline functions*). With GCC this is selected automatically when compiling in C99 mode (`-std=c99`).

#### int `gsl_check_range`

If inline functions are not used, calls to the functions `gsl_vector_get()` and `gsl_vector_set()` will link to the compiled versions of these functions in the library itself. The range checking in these functions is controlled by the global integer variable `gsl_check_range`. It is enabled by default—to disable it, set `gsl_check_range` to zero. Due to function-call overhead, there is less benefit in disabling range checking here than for inline functions.

double `gsl_vector_get`(const *gsl\_vector* \*v, const size\_t i)

This function returns the *i*-th element of a vector *v*. If *i* lies outside the allowed range of 0 to `size - 1` then the error handler is invoked and 0 is returned. An inline version of this function is used when `HAVE_INLINE` is defined.

void `gsl_vector_set`(*gsl\_vector* \*v, const size\_t i, double x)

This function sets the value of the *i*-th element of a vector *v* to *x*. If *i* lies outside the allowed range of 0 to `size - 1` then the error handler is invoked. An inline version of this function is used when `HAVE_INLINE` is defined.

double \*`gsl_vector_ptr`(*gsl\_vector* \*v, size\_t i)

const double \*`gsl_vector_const_ptr`(const *gsl\_vector* \*v, size\_t i)

These functions return a pointer to the *i*-th element of a vector *v*. If *i* lies outside the allowed range of 0 to `size - 1` then the error handler is invoked and a null pointer is returned. Inline versions of these functions are used when `HAVE_INLINE` is defined.

### 8.3.3 Initializing vector elements

void `gsl_vector_set_all`(*gsl\_vector* \*v, double x)

This function sets all the elements of the vector *v* to the value *x*.

void `gsl_vector_set_zero`(*gsl\_vector* \*v)

This function sets all the elements of the vector *v* to zero.

int `gsl_vector_set_basis`(*gsl\_vector* \*v, size\_t i)

This function makes a basis vector by setting all the elements of the vector *v* to zero except for the *i*-th element which is set to one.

### 8.3.4 Reading and writing vectors

The library provides functions for reading and writing vectors to a file as binary data or formatted text.

int `gsl_vector_fwrite`(FILE \*stream, const *gsl\_vector* \*v)

This function writes the elements of the vector *v* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int `gsl_vector_fread`(FILE \*stream, *gsl\_vector* \*v)

This function reads into the vector *v* from the open stream *stream* in binary format. The vector *v* must be preallocated with the correct length since the function uses the size of *v* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int `gsl_vector_fprintf`(FILE \*stream, const *gsl\_vector* \*v, const char \*format)

This function writes the elements of the vector *v* line-by-line to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats for floating point numbers and `%d` for integers. The

function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file.

int **`gsl_vector_fscanf`**(FILE \*stream, *`gsl_vector`* \*v)

This function reads formatted data from the stream *stream* into the vector *v*. The vector *v* must be preallocated with the correct length since the function uses the size of *v* to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

### 8.3.5 Vector views

In addition to creating vectors from slices of blocks it is also possible to slice vectors and create vector views. For example, a subvector of another vector can be described with a view, or two views can be made which provide access to the even and odd elements of a vector.

type **`gsl_vector_view`**

type **`gsl_vector_const_view`**

A vector view is a temporary object, stored on the stack, which can be used to operate on a subset of vector elements. Vector views can be defined for both constant and non-constant vectors, using separate types that preserve constness. A vector view has the type *`gsl_vector_view`* and a constant vector view has the type *`gsl_vector_const_view`*. In both cases the elements of the view can be accessed as a *`gsl_vector`* using the `vector` component of the view object. A pointer to a vector of type `gsl_vector *` or `const gsl_vector *` can be obtained by taking the address of this component with the `&` operator.

When using this pointer it is important to ensure that the view itself remains in scope—the simplest way to do so is by always writing the pointer as `&view.vector`, and never storing this value in another variable.

*`gsl_vector_view`* **`gsl_vector_subvector`**(*`gsl_vector`* \*v, size\_t offset, size\_t n)

*`gsl_vector_const_view`* **`gsl_vector_const_subvector`**(const *`gsl_vector`* \*v, size\_t offset, size\_t n)

These functions return a vector view of a subvector of another vector *v*. The start of the new vector is offset by *offset* elements from the start of the original vector. The new vector has *n* elements. Mathematically, the *i*-th element of the new vector *v'* is given by:

$$v'(i) = v->data[(offset + i)*v->stride]$$

where the index *i* runs from 0 to *n* - 1.

The `data` pointer of the returned vector struct is set to null if the combined parameters (*offset*, *n*) overrun the end of the original vector.

The new vector is only a view of the block underlying the original vector, *v*. The block containing the elements of *v* is not owned by the new vector. When the view goes out of scope the original vector *v* and its block will continue to exist. The original memory can only be deallocated by freeing the original vector. Of course, the original vector should not be deallocated while the view is still in use.

The function `gsl_vector_const_subvector()` is equivalent to `gsl_vector_subvector()` but can be used for vectors which are declared `const`.

*`gsl_vector_view`* **`gsl_vector_subvector_with_stride`**(*`gsl_vector`* \*v, size\_t offset, size\_t stride, size\_t n)

*`gsl_vector_const_view`* **`gsl_vector_const_subvector_with_stride`**(const *`gsl_vector`* \*v, size\_t offset, size\_t stride, size\_t n)

These functions return a vector view of a subvector of another vector *v* with an additional stride argument. The subvector is formed in the same way as for `gsl_vector_subvector()` but the new vector has *n* elements with a step-size of *stride* from one element to the next in the original vector. Mathematically, the *i*-th element of the new vector *v'* is given by:

```
v'(i) = v->data[(offset + i*stride)*v->stride]
```

where the index  $i$  runs from 0 to  $n - 1$ .

Note that subvector views give direct access to the underlying elements of the original vector. For example, the following code will zero the even elements of the vector  $v$  of length  $n$ , while leaving the odd elements untouched:

```
gsl_vector_view v_even = gsl_vector_subvector_with_stride (v, 0, 2, n/2);
gsl_vector_set_zero (&v_even.vector);
```

A vector view can be passed to any subroutine which takes a vector argument just as a directly allocated vector would be, using `&view.vector`. For example, the following code computes the norm of the odd elements of  $v$  using the BLAS routine `dnorm2`:

```
gsl_vector_view v_odd = gsl_vector_subvector_with_stride (v, 1, 2, n/2);
double r = gsl_blas_dnorm2 (&v_odd.vector);
```

The function `gsl_vector_const_subvector_with_stride()` is equivalent to `gsl_vector_subvector_with_stride()` but can be used for vectors which are declared `const`.

*gsl\_vector\_view* **gsl\_vector\_complex\_real**(gsl\_vector\_complex \*v)

*gsl\_vector\_const\_view* **gsl\_vector\_complex\_const\_real**(const gsl\_vector\_complex \*v)

These functions return a vector view of the real parts of the complex vector  $v$ .

The function `gsl_vector_complex_const_real()` is equivalent to `gsl_vector_complex_real()` but can be used for vectors which are declared `const`.

*gsl\_vector\_view* **gsl\_vector\_complex\_imag**(gsl\_vector\_complex \*v)

*gsl\_vector\_const\_view* **gsl\_vector\_complex\_const\_imag**(const gsl\_vector\_complex \*v)

These functions return a vector view of the imaginary parts of the complex vector  $v$ .

The function `gsl_vector_complex_const_imag()` is equivalent to `gsl_vector_complex_imag()` but can be used for vectors which are declared `const`.

*gsl\_vector\_view* **gsl\_vector\_view\_array**(double \*base, size\_t n)

*gsl\_vector\_const\_view* **gsl\_vector\_const\_view\_array**(const double \*base, size\_t n)

These functions return a vector view of an array. The start of the new vector is given by *base* and has  $n$  elements. Mathematically, the  $i$ -th element of the new vector  $v'$  is given by:

```
v'(i) = base[i]
```

where the index  $i$  runs from 0 to  $n - 1$ .

The array containing the elements of  $v$  is not owned by the new vector view. When the view goes out of scope the original array will continue to exist. The original memory can only be deallocated by freeing the original pointer *base*. Of course, the original array should not be deallocated while the view is still in use.

The function `gsl_vector_const_view_array()` is equivalent to `gsl_vector_view_array()` but can be used for arrays which are declared `const`.

*gsl\_vector\_view* **gsl\_vector\_view\_array\_with\_stride**(double \*base, size\_t stride, size\_t n)

*gsl\_vector\_const\_view* **gsl\_vector\_const\_view\_array\_with\_stride**(const double \*base, size\_t stride, size\_t n)

These functions return a vector view of an array *base* with an additional stride argument. The subvector is formed in the same way as for *gsl\_vector\_view\_array()* but the new vector has *n* elements with a step-size of *stride* from one element to the next in the original array. Mathematically, the *i*-th element of the new vector *v'* is given by:

$$v'(i) = base[i*stride]$$

where the index *i* runs from 0 to *n* - 1.

Note that the view gives direct access to the underlying elements of the original array. A vector view can be passed to any subroutine which takes a vector argument just as a directly allocated vector would be, using `&view.vector`.

The function *gsl\_vector\_const\_view\_array\_with\_stride()* is equivalent to *gsl\_vector\_view\_array\_with\_stride()* but can be used for arrays which are declared `const`.

### 8.3.6 Copying vectors

Common operations on vectors such as addition and multiplication are available in the BLAS part of the library (see *BLAS Support*). However, it is useful to have a small number of utility functions which do not require the full BLAS code. The following functions fall into this category.

int **gsl\_vector\_memcpy**(*gsl\_vector* \*dest, const *gsl\_vector* \*src)

This function copies the elements of the vector *src* into the vector *dest*. The two vectors must have the same length.

int **gsl\_vector\_complex\_conj\_memcpy**(*gsl\_vector\_complex* \*dest, const *gsl\_vector\_complex* \*src)

This function copies the complex conjugate of the vector *src* into the vector *dest*. The two vectors must have the same length.

int **gsl\_vector\_swap**(*gsl\_vector* \*v, *gsl\_vector* \*w)

This function exchanges the elements of the vectors *v* and *w* by copying. The two vectors must have the same length.

### 8.3.7 Exchanging elements

The following functions can be used to exchange, or permute, the elements of a vector.

int **gsl\_vector\_swap\_elements**(*gsl\_vector* \*v, size\_t i, size\_t j)

This function exchanges the *i*-th and *j*-th elements of the vector *v* in-place.

int **gsl\_vector\_reverse**(*gsl\_vector* \*v)

This function reverses the order of the elements of the vector *v*.



### 8.3.8 Vector operations

int **gsl\_vector\_add**(*gsl\_vector* \*a, const *gsl\_vector* \*b)

This function adds the elements of vector *b* to the elements of vector *a*. The result  $a_i \leftarrow a_i + b_i$  is stored in *a* and *b* remains unchanged. The two vectors must have the same length.

int **gsl\_vector\_sub**(*gsl\_vector* \*a, const *gsl\_vector* \*b)

This function subtracts the elements of vector *b* from the elements of vector *a*. The result  $a_i \leftarrow a_i - b_i$  is stored in *a* and *b* remains unchanged. The two vectors must have the same length.

int **gsl\_vector\_mul**(*gsl\_vector* \*a, const *gsl\_vector* \*b)

This function multiplies the elements of vector *a* by the elements of vector *b*. The result  $a_i \leftarrow a_i * b_i$  is stored in *a* and *b* remains unchanged. The two vectors must have the same length.

int **gsl\_vector\_div**(*gsl\_vector* \*a, const *gsl\_vector* \*b)

This function divides the elements of vector *a* by the elements of vector *b*. The result  $a_i \leftarrow a_i / b_i$  is stored in *a* and *b* remains unchanged. The two vectors must have the same length.

int **gsl\_vector\_scale**(*gsl\_vector* \*a, const double x)

This function multiplies the elements of vector *a* by the constant factor *x*. The result  $a_i \leftarrow x a_i$  is stored in *a*.

int **gsl\_vector\_add\_constant**(*gsl\_vector* \*a, const double x)

This function adds the constant value *x* to the elements of the vector *a*. The result  $a_i \leftarrow a_i + x$  is stored in *a*.

double **gsl\_vector\_sum**(const *gsl\_vector* \*a)

This function returns the sum of the elements of *a*, defined as  $\sum_{i=1}^n a_i$

int **gsl\_vector\_axpby**(const double alpha, const *gsl\_vector* \*x, const double beta, *gsl\_vector* \*y)

This function performs the operation  $y \leftarrow \alpha x + \beta y$ . The vectors *x* and *y* must have the same length.

### 8.3.9 Finding maximum and minimum elements of vectors

The following operations are only defined for real vectors.

double **gsl\_vector\_max**(const *gsl\_vector* \*v)

This function returns the maximum value in the vector *v*.

double **gsl\_vector\_min**(const *gsl\_vector* \*v)

This function returns the minimum value in the vector *v*.

void **gsl\_vector\_minmax**(const *gsl\_vector* \*v, double \*min\_out, double \*max\_out)

This function returns the minimum and maximum values in the vector *v*, storing them in *min\_out* and *max\_out*.

size\_t **gsl\_vector\_max\_index**(const *gsl\_vector* \*v)

This function returns the index of the maximum value in the vector *v*. When there are several equal maximum elements then the lowest index is returned.

size\_t **gsl\_vector\_min\_index**(const *gsl\_vector* \*v)

This function returns the index of the minimum value in the vector *v*. When there are several equal minimum elements then the lowest index is returned.

void **gsl\_vector\_minmax\_index**(const *gsl\_vector* \*v, size\_t \*imin, size\_t \*imax)

This function returns the indices of the minimum and maximum values in the vector *v*, storing them in *imin* and *imax*. When there are several equal minimum or maximum elements then the lowest indices are returned.

### 8.3.10 Vector properties

The following functions are defined for real and complex vectors. For complex vectors both the real and imaginary parts must satisfy the conditions.

```
int gsl_vector_isnull(const gsl_vector *v)
```

```
int gsl_vector_ispos(const gsl_vector *v)
```

```
int gsl_vector_isneg(const gsl_vector *v)
```

```
int gsl_vector_isnonneg(const gsl_vector *v)
```

These functions return 1 if all the elements of the vector *v* are zero, strictly positive, strictly negative, or non-negative respectively, and 0 otherwise.

```
int gsl_vector_equal(const gsl_vector *u, const gsl_vector *v)
```

This function returns 1 if the vectors *u* and *v* are equal (by comparison of element values) and 0 otherwise.

### 8.3.11 Example programs for vectors

This program shows how to allocate, initialize and read from a vector using the functions *gsl\_vector\_alloc()*, *gsl\_vector\_set()* and *gsl\_vector\_get()*.

```
#include <stdio.h>
#include <gsl/gsl_vector.h>

int
main (void)
{
    int i;
    gsl_vector * v = gsl_vector_alloc (3);

    for (i = 0; i < 3; i++)
    {
        gsl_vector_set (v, i, 1.23 + i);
    }

    for (i = 0; i < 100; i++) /* OUT OF RANGE ERROR */
    {
        printf ("v_%d = %g\n", i, gsl_vector_get (v, i));
    }

    gsl_vector_free (v);
    return 0;
}
```

Here is the output from the program. The final loop attempts to read outside the range of the vector *v*, and the error is trapped by the range-checking code in *gsl\_vector\_get()*.

```
$ ./a.out
v_0 = 1.23
v_1 = 2.23
v_2 = 3.23
```

(continues on next page)

(continued from previous page)

```
gsl: vector_source.c:12: ERROR: index out of range
Default GSL error handler invoked.
Aborted (core dumped)
```

The next program shows how to write a vector to a file.

```
#include <stdio.h>
#include <gsl/gsl_vector.h>

int
main (void)
{
  int i;
  gsl_vector * v = gsl_vector_alloc (100);

  for (i = 0; i < 100; i++)
    {
      gsl_vector_set (v, i, 1.23 + i);
    }

  {
    FILE * f = fopen ("test.dat", "w");
    gsl_vector_fprintf (f, v, "%.5g");
    fclose (f);
  }

  gsl_vector_free (v);
  return 0;
}
```

After running this program the file `test.dat` should contain the elements of `v`, written using the format specifier `%.5g`. The vector could then be read back in using the function `gsl_vector_fscanf (f, v)` as follows:

```
#include <stdio.h>
#include <gsl/gsl_vector.h>

int
main (void)
{
  int i;
  gsl_vector * v = gsl_vector_alloc (10);

  {
    FILE * f = fopen ("test.dat", "r");
    gsl_vector_fscanf (f, v);
    fclose (f);
  }

  for (i = 0; i < 10; i++)
    {
      printf ("%g\n", gsl_vector_get(v, i));
    }
}
```

(continues on next page)

(continued from previous page)

```

gsl_vector_free (v);
return 0;
}

```

## 8.4 Matrices

Matrices are defined by a `gsl_matrix` structure which describes a generalized slice of a block. Like a vector it represents a set of elements in an area of memory, but uses two indices instead of one.

### type `gsl_matrix`

The `gsl_matrix` structure contains six components, the two dimensions of the matrix, a physical dimension, a pointer to the memory where the elements of the matrix are stored, `data`, a pointer to the block owned by the matrix `block`, if any, and an ownership flag, `owner`. The physical dimension determines the memory layout and can differ from the matrix dimension to allow the use of submatrices. The `gsl_matrix` structure is very simple and looks like this:

```

typedef struct
{
    size_t size1;
    size_t size2;
    size_t tda;
    double * data;
    gsl_block * block;
    int owner;
} gsl_matrix;

```

Matrices are stored in row-major order, meaning that each row of elements forms a contiguous block in memory. This is the standard “C-language ordering” of two-dimensional arrays. Note that Fortran stores arrays in column-major order. The number of rows is `size1`. The range of valid row indices runs from 0 to `size1 - 1`. Similarly `size2` is the number of columns. The range of valid column indices runs from 0 to `size2 - 1`. The physical row dimension `tda`, or *trailing dimension*, specifies the size of a row of the matrix as laid out in memory.

For example, in the following matrix `size1` is 3, `size2` is 4, and `tda` is 8. The physical memory layout of the matrix begins in the top left hand-corner and proceeds from left to right along each row in turn.

```

00 01 02 03 XX XX XX XX
10 11 12 13 XX XX XX XX
20 21 22 23 XX XX XX XX

```

Each unused memory location is represented by “XX”. The pointer `data` gives the location of the first element of the matrix in memory. The pointer `block` stores the location of the memory block in which the elements of the matrix are located (if any). If the matrix owns this block then the `owner` field is set to one and the block will be deallocated when the matrix is freed. If the matrix is only a slice of a block owned by another object then the `owner` field is zero and any underlying block will not be freed.

The functions for allocating and accessing matrices are defined in `gsl_matrix.h`.

### 8.4.1 Matrix allocation

The functions for allocating memory to a matrix follow the style of `malloc` and `free`. They also perform their own error checking. If there is insufficient memory available to allocate a matrix then the functions call the GSL error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every `alloc`.

`gsl_matrix *gsl_matrix_alloc(size_t n1, size_t n2)`

This function creates a matrix of size  $n1$  rows by  $n2$  columns, returning a pointer to a newly initialized matrix struct. A new block is allocated for the elements of the matrix, and stored in the `block` component of the matrix struct. The block is “owned” by the matrix, and will be deallocated when the matrix is deallocated. Requesting zero for  $n1$  or  $n2$  is valid and returns a non-null result.

`gsl_matrix *gsl_matrix_calloc(size_t n1, size_t n2)`

This function allocates memory for a matrix of size  $n1$  rows by  $n2$  columns and initializes all the elements of the matrix to zero.

`void gsl_matrix_free(gsl_matrix *m)`

This function frees a previously allocated matrix  $m$ . If the matrix was created using `gsl_matrix_alloc()` then the block underlying the matrix will also be deallocated. If the matrix has been created from another object then the memory is still owned by that object and will not be deallocated.

### 8.4.2 Accessing matrix elements

The functions for accessing the elements of a matrix use the same range checking system as vectors. You can turn off range checking by recompiling your program with the preprocessor definition `GSL_RANGE_CHECK_OFF`.

The elements of the matrix are stored in “C-order”, where the second index moves continuously through memory. More precisely, the element accessed by the function `gsl_matrix_get(m, i, j)` and `gsl_matrix_set(m, i, j, x)` is:

```
m->data[i * m->tda + j]
```

where `tda` is the physical row-length of the matrix.

`double gsl_matrix_get(const gsl_matrix *m, const size_t i, const size_t j)`

This function returns the  $(i, j)$ -th element of a matrix  $m$ . If  $i$  or  $j$  lie outside the allowed range of 0 to  $n1 - 1$  and 0 to  $n2 - 1$  then the error handler is invoked and 0 is returned. An inline version of this function is used when `HAVE_INLINE` is defined.

`void gsl_matrix_set(gsl_matrix *m, const size_t i, const size_t j, double x)`

This function sets the value of the  $(i, j)$ -th element of a matrix  $m$  to  $x$ . If  $i$  or  $j$  lies outside the allowed range of 0 to  $n1 - 1$  and 0 to  $n2 - 1$  then the error handler is invoked. An inline version of this function is used when `HAVE_INLINE` is defined.

`double *gsl_matrix_ptr(gsl_matrix *m, size_t i, size_t j)`

`const double *gsl_matrix_const_ptr(const gsl_matrix *m, size_t i, size_t j)`

These functions return a pointer to the  $(i, j)$ -th element of a matrix  $m$ . If  $i$  or  $j$  lie outside the allowed range of 0 to  $n1 - 1$  and 0 to  $n2 - 1$  then the error handler is invoked and a null pointer is returned. Inline versions of these functions are used when `HAVE_INLINE` is defined.

### 8.4.3 Initializing matrix elements

void **gsl\_matrix\_set\_all**(*gsl\_matrix* \*m, double x)

This function sets all the elements of the matrix *m* to the value *x*.

void **gsl\_matrix\_set\_zero**(*gsl\_matrix* \*m)

This function sets all the elements of the matrix *m* to zero.

void **gsl\_matrix\_set\_identity**(*gsl\_matrix* \*m)

This function sets the elements of the matrix *m* to the corresponding elements of the identity matrix,  $m(i, j) = \delta(i, j)$ , i.e. a unit diagonal with all off-diagonal elements zero. This applies to both square and rectangular matrices.

### 8.4.4 Reading and writing matrices

The library provides functions for reading and writing matrices to a file as binary data or formatted text.

int **gsl\_matrix\_fwrite**(FILE \*stream, const *gsl\_matrix* \*m)

This function writes the elements of the matrix *m* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl\_matrix\_fread**(FILE \*stream, *gsl\_matrix* \*m)

This function reads into the matrix *m* from the open stream *stream* in binary format. The matrix *m* must be preallocated with the correct dimensions since the function uses the size of *m* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl\_matrix\_fprintf**(FILE \*stream, const *gsl\_matrix* \*m, const char \*format)

This function writes the elements of the matrix *m* line-by-line to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats for floating point numbers and `%d` for integers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file.

int **gsl\_matrix\_fscanf**(FILE \*stream, *gsl\_matrix* \*m)

This function reads formatted data from the stream *stream* into the matrix *m*. The matrix *m* must be preallocated with the correct dimensions since the function uses the size of *m* to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

### 8.4.5 Matrix views

type **gsl\_matrix\_view**

type **gsl\_matrix\_const\_view**

A matrix view is a temporary object, stored on the stack, which can be used to operate on a subset of matrix elements. Matrix views can be defined for both constant and non-constant matrices using separate types that preserve constness. A matrix view has the type *gsl\_matrix\_view* and a constant matrix view has the type *gsl\_matrix\_const\_view*. In both cases the elements of the view can be accessed using the `matrix` component of the view object. A pointer `gsl_matrix *` or `const gsl_matrix *` can be obtained by taking the address of the `matrix` component with the `&` operator. In addition to matrix views it is also possible to create vector views of a matrix, such as row or column views.

*gsl\_matrix\_view* **gsl\_matrix\_submatrix**(*gsl\_matrix* \*m, size\_t k1, size\_t k2, size\_t n1, size\_t n2)

*gsl\_matrix\_const\_view* **gsl\_matrix\_const\_submatrix**(const *gsl\_matrix* \*m, size\_t k1, size\_t k2, size\_t n1, size\_t n2)

These functions return a matrix view of a submatrix of the matrix *m*. The upper-left element of the submatrix is the element (*k1*, *k2*) of the original matrix. The submatrix has *n1* rows and *n2* columns. The physical number of columns in memory given by *tda* is unchanged. Mathematically, the (*i*, *j*)-th element of the new matrix is given by:

$$m'(i, j) = m->data[(k1*m->tda + k2) + i*m->tda + j]$$

where the index *i* runs from 0 to *n1* - 1 and the index *j* runs from 0 to *n2* - 1.

The *data* pointer of the returned matrix struct is set to null if the combined parameters (*i*, *j*, *n1*, *n2*, *tda*) overrun the ends of the original matrix.

The new matrix view is only a view of the block underlying the existing matrix, *m*. The block containing the elements of *m* is not owned by the new matrix view. When the view goes out of scope the original matrix *m* and its block will continue to exist. The original memory can only be deallocated by freeing the original matrix. Of course, the original matrix should not be deallocated while the view is still in use.

The function *gsl\_matrix\_const\_submatrix()* is equivalent to *gsl\_matrix\_submatrix()* but can be used for matrices which are declared *const*.

*gsl\_matrix\_view* **gsl\_matrix\_view\_array**(double \*base, size\_t n1, size\_t n2)

*gsl\_matrix\_const\_view* **gsl\_matrix\_const\_view\_array**(const double \*base, size\_t n1, size\_t n2)

These functions return a matrix view of the array *base*. The matrix has *n1* rows and *n2* columns. The physical number of columns in memory is also given by *n2*. Mathematically, the (*i*, *j*)-th element of the new matrix is given by:

$$m'(i, j) = base[i*n2 + j]$$

where the index *i* runs from 0 to *n1* - 1 and the index *j* runs from 0 to *n2* - 1.

The new matrix is only a view of the array *base*. When the view goes out of scope the original array *base* will continue to exist. The original memory can only be deallocated by freeing the original array. Of course, the original array should not be deallocated while the view is still in use.

The function *gsl\_matrix\_const\_view\_array()* is equivalent to *gsl\_matrix\_view\_array()* but can be used for matrices which are declared *const*.

*gsl\_matrix\_view* **gsl\_matrix\_view\_array\_with\_tda**(double \*base, size\_t n1, size\_t n2, size\_t tda)

*gsl\_matrix\_const\_view* **gsl\_matrix\_const\_view\_array\_with\_tda**(const double \*base, size\_t n1, size\_t n2, size\_t tda)

These functions return a matrix view of the array *base* with a physical number of columns *tda* which may differ from the corresponding dimension of the matrix. The matrix has *n1* rows and *n2* columns, and the physical number of columns in memory is given by *tda*. Mathematically, the (*i*, *j*)-th element of the new matrix is given by:

$$m'(i, j) = base[i*tda + j]$$

where the index *i* runs from 0 to *n1* - 1 and the index *j* runs from 0 to *n2* - 1.

The new matrix is only a view of the array *base*. When the view goes out of scope the original array *base* will continue to exist. The original memory can only be deallocated by freeing the original array. Of course, the original array should not be deallocated while the view is still in use.

The function *gsl\_matrix\_const\_view\_array\_with\_tda()* is equivalent to *gsl\_matrix\_view\_array\_with\_tda()* but can be used for matrices which are declared *const*.

*gsl\_matrix\_view* **gsl\_matrix\_view\_vector**(*gsl\_vector* \*v, size\_t n1, size\_t n2)

*gsl\_matrix\_const\_view* **gsl\_matrix\_const\_view\_vector**(const *gsl\_vector* \*v, size\_t n1, size\_t n2)

These functions return a matrix view of the vector *v*. The matrix has *n1* rows and *n2* columns. The vector must have unit stride. The physical number of columns in memory is also given by *n2*. Mathematically, the  $(i, j)$ -th element of the new matrix is given by:

$$m'(i, j) = v->data[i*n2 + j]$$

where the index *i* runs from 0 to *n1* - 1 and the index *j* runs from 0 to *n2* - 1.

The new matrix is only a view of the vector *v*. When the view goes out of scope the original vector *v* will continue to exist. The original memory can only be deallocated by freeing the original vector. Of course, the original vector should not be deallocated while the view is still in use.

The function *gsl\_matrix\_const\_view\_vector*() is equivalent to *gsl\_matrix\_view\_vector*() but can be used for matrices which are declared `const`.

*gsl\_matrix\_view* **gsl\_matrix\_view\_vector\_with\_tda**(*gsl\_vector* \*v, size\_t n1, size\_t n2, size\_t tda)

*gsl\_matrix\_const\_view* **gsl\_matrix\_const\_view\_vector\_with\_tda**(const *gsl\_vector* \*v, size\_t n1, size\_t n2, size\_t tda)

These functions return a matrix view of the vector *v* with a physical number of columns *tda* which may differ from the corresponding matrix dimension. The vector must have unit stride. The matrix has *n1* rows and *n2* columns, and the physical number of columns in memory is given by *tda*. Mathematically, the  $(i, j)$ -th element of the new matrix is given by:

$$m'(i, j) = v->data[i*tda + j]$$

where the index *i* runs from 0 to *n1* - 1 and the index *j* runs from 0 to *n2* - 1.

The new matrix is only a view of the vector *v*. When the view goes out of scope the original vector *v* will continue to exist. The original memory can only be deallocated by freeing the original vector. Of course, the original vector should not be deallocated while the view is still in use.

The function *gsl\_matrix\_const\_view\_vector\_with\_tda*() is equivalent to *gsl\_matrix\_view\_vector\_with\_tda*() but can be used for matrices which are declared `const`.

## 8.4.6 Creating row and column views

In general there are two ways to access an object, by reference or by copying. The functions described in this section create vector views which allow access to a row or column of a matrix by reference. Modifying elements of the view is equivalent to modifying the matrix, since both the vector view and the matrix point to the same memory block.

*gsl\_vector\_view* **gsl\_matrix\_row**(*gsl\_matrix* \*m, size\_t i)

*gsl\_vector\_const\_view* **gsl\_matrix\_const\_row**(const *gsl\_matrix* \*m, size\_t i)

These functions return a vector view of the *i*-th row of the matrix *m*. The data pointer of the new vector is set to null if *i* is out of range.

The function *gsl\_matrix\_const\_row*() is equivalent to *gsl\_matrix\_row*() but can be used for matrices which are declared `const`.

*gsl\_vector\_view* **gsl\_matrix\_column**(*gsl\_matrix* \*m, size\_t j)



*gsl\_vector\_const\_view* **gsl\_matrix\_const\_column**(const *gsl\_matrix* \*m, size\_t j)

These functions return a vector view of the  $j$ -th column of the matrix  $m$ . The data pointer of the new vector is set to null if  $j$  is out of range.

The function *gsl\_matrix\_const\_column()* is equivalent to *gsl\_matrix\_column()* but can be used for matrices which are declared `const`.

*gsl\_vector\_view* **gsl\_matrix\_subrow**(*gsl\_matrix* \*m, size\_t i, size\_t offset, size\_t n)

*gsl\_vector\_const\_view* **gsl\_matrix\_const\_subrow**(const *gsl\_matrix* \*m, size\_t i, size\_t offset, size\_t n)

These functions return a vector view of the  $i$ -th row of the matrix  $m$  beginning at *offset* elements past the first column and containing  $n$  elements. The data pointer of the new vector is set to null if  $i$ , *offset*, or  $n$  are out of range.

The function *gsl\_matrix\_const\_subrow()* is equivalent to *gsl\_matrix\_subrow()* but can be used for matrices which are declared `const`.

*gsl\_vector\_view* **gsl\_matrix\_subcolumn**(*gsl\_matrix* \*m, size\_t j, size\_t offset, size\_t n)

*gsl\_vector\_const\_view* **gsl\_matrix\_const\_subcolumn**(const *gsl\_matrix* \*m, size\_t j, size\_t offset, size\_t n)

These functions return a vector view of the  $j$ -th column of the matrix  $m$  beginning at *offset* elements past the first row and containing  $n$  elements. The data pointer of the new vector is set to null if  $j$ , *offset*, or  $n$  are out of range.

The function *gsl\_matrix\_const\_subcolumn()* is equivalent to *gsl\_matrix\_subcolumn()* but can be used for matrices which are declared `const`.

*gsl\_vector\_view* **gsl\_matrix\_diagonal**(*gsl\_matrix* \*m)

*gsl\_vector\_const\_view* **gsl\_matrix\_const\_diagonal**(const *gsl\_matrix* \*m)

These functions return a vector view of the diagonal of the matrix  $m$ . The matrix  $m$  is not required to be square. For a rectangular matrix the length of the diagonal is the same as the smaller dimension of the matrix.

The function *gsl\_matrix\_const\_diagonal()* is equivalent to *gsl\_matrix\_diagonal()* but can be used for matrices which are declared `const`.

*gsl\_vector\_view* **gsl\_matrix\_subdiagonal**(*gsl\_matrix* \*m, size\_t k)

*gsl\_vector\_const\_view* **gsl\_matrix\_const\_subdiagonal**(const *gsl\_matrix* \*m, size\_t k)

These functions return a vector view of the  $k$ -th subdiagonal of the matrix  $m$ . The matrix  $m$  is not required to be square. The diagonal of the matrix corresponds to  $k = 0$ .

The function *gsl\_matrix\_const\_subdiagonal()* is equivalent to *gsl\_matrix\_subdiagonal()* but can be used for matrices which are declared `const`.

*gsl\_vector\_view* **gsl\_matrix\_superdiagonal**(*gsl\_matrix* \*m, size\_t k)

*gsl\_vector\_const\_view* **gsl\_matrix\_const\_superdiagonal**(const *gsl\_matrix* \*m, size\_t k)

These functions return a vector view of the  $k$ -th superdiagonal of the matrix  $m$ . The matrix  $m$  is not required to be square. The diagonal of the matrix corresponds to  $k = 0$ .

The function *gsl\_matrix\_const\_superdiagonal()* is equivalent to *gsl\_matrix\_superdiagonal()* but can be used for matrices which are declared `const`.

### 8.4.7 Copying matrices

int **gsl\_matrix\_memcpy**(*gsl\_matrix* \*dest, const *gsl\_matrix* \*src)

This function copies the elements of the matrix *src* into the matrix *dest*. The two matrices must have the same size.

int **gsl\_matrix\_swap**(*gsl\_matrix* \*m1, *gsl\_matrix* \*m2)

This function exchanges the elements of the matrices *m1* and *m2* by copying. The two matrices must have the same size.

### 8.4.8 Copying rows and columns

The functions described in this section copy a row or column of a matrix into a vector. This allows the elements of the vector and the matrix to be modified independently. Note that if the matrix and the vector point to overlapping regions of memory then the result will be undefined. The same effect can be achieved with more generality using *gsl\_vector\_memcpy()* with vector views of rows and columns.

int **gsl\_matrix\_get\_row**(*gsl\_vector* \*v, const *gsl\_matrix* \*m, size\_t i)

This function copies the elements of the *i*-th row of the matrix *m* into the vector *v*. The length of the vector must be the same as the length of the row.

int **gsl\_matrix\_get\_col**(*gsl\_vector* \*v, const *gsl\_matrix* \*m, size\_t j)

This function copies the elements of the *j*-th column of the matrix *m* into the vector *v*. The length of the vector must be the same as the length of the column.

int **gsl\_matrix\_set\_row**(*gsl\_matrix* \*m, size\_t i, const *gsl\_vector* \*v)

This function copies the elements of the vector *v* into the *i*-th row of the matrix *m*. The length of the vector must be the same as the length of the row.

int **gsl\_matrix\_set\_col**(*gsl\_matrix* \*m, size\_t j, const *gsl\_vector* \*v)

This function copies the elements of the vector *v* into the *j*-th column of the matrix *m*. The length of the vector must be the same as the length of the column.

### 8.4.9 Exchanging rows and columns

The following functions can be used to exchange the rows and columns of a matrix.

int **gsl\_matrix\_swap\_rows**(*gsl\_matrix* \*m, size\_t i, size\_t j)

This function exchanges the *i*-th and *j*-th rows of the matrix *m* in-place.

int **gsl\_matrix\_swap\_columns**(*gsl\_matrix* \*m, size\_t i, size\_t j)

This function exchanges the *i*-th and *j*-th columns of the matrix *m* in-place.

int **gsl\_matrix\_swap\_rowcol**(*gsl\_matrix* \*m, size\_t i, size\_t j)

This function exchanges the *i*-th row and *j*-th column of the matrix *m* in-place. The matrix must be square for this operation to be possible.

int **gsl\_matrix\_transpose\_memcpy**(*gsl\_matrix* \*dest, const *gsl\_matrix* \*src)

This function makes the matrix *dest* the transpose of the matrix *src* by copying the elements of *src* into *dest*. This function works for all matrices provided that the dimensions of the matrix *dest* match the transposed dimensions of the matrix *src*.

int **gsl\_matrix\_transpose**(*gsl\_matrix* \*m)

This function replaces the matrix *m* by its transpose by copying the elements of the matrix in-place. The matrix must be square for this operation to be possible.

int **gsl\_matrix\_complex\_conjtrans\_memcpy**(gsl\_matrix\_complex \*dest, const gsl\_matrix\_complex \*src)

This function makes the matrix *dest* the conjugate transpose of the matrix *src* by copying the complex conjugate elements of *src* into *dest*. This function works for all complex matrices provided that the dimensions of the matrix *dest* match the transposed dimensions of the matrix *src*.

### 8.4.10 Matrix operations

The following operations are defined for real and complex matrices.

int **gsl\_matrix\_add**(gsl\_matrix \*a, const gsl\_matrix \*b)

This function adds the elements of matrix *b* to the elements of matrix *a*. The result  $a(i, j) \leftarrow a(i, j) + b(i, j)$  is stored in *a* and *b* remains unchanged. The two matrices must have the same dimensions.

int **gsl\_matrix\_sub**(gsl\_matrix \*a, const gsl\_matrix \*b)

This function subtracts the elements of matrix *b* from the elements of matrix *a*. The result  $a(i, j) \leftarrow a(i, j) - b(i, j)$  is stored in *a* and *b* remains unchanged. The two matrices must have the same dimensions.

int **gsl\_matrix\_mul\_elements**(gsl\_matrix \*a, const gsl\_matrix \*b)

This function multiplies the elements of matrix *a* by the elements of matrix *b*. The result  $a(i, j) \leftarrow a(i, j) * b(i, j)$  is stored in *a* and *b* remains unchanged. The two matrices must have the same dimensions.

int **gsl\_matrix\_div\_elements**(gsl\_matrix \*a, const gsl\_matrix \*b)

This function divides the elements of matrix *a* by the elements of matrix *b*. The result  $a(i, j) \leftarrow a(i, j) / b(i, j)$  is stored in *a* and *b* remains unchanged. The two matrices must have the same dimensions.

int **gsl\_matrix\_scale**(gsl\_matrix \*a, const double x)

This function multiplies the elements of matrix *a* by the constant factor *x*. The result  $a(i, j) \leftarrow xa(i, j)$  is stored in *a*.

int **gsl\_matrix\_scale\_columns**(gsl\_matrix \*A, const gsl\_vector \*x)

This function scales the columns of the *M*-by-*N* matrix *A* by the elements of the vector *x*, of length *N*. The *j*-th column of *A* is multiplied by  $x_j$ . This is equivalent to forming

$$A \rightarrow AX$$

where  $X = \text{diag}(x)$ .

int **gsl\_matrix\_scale\_rows**(gsl\_matrix \*A, const gsl\_vector \*x)

This function scales the rows of the *M*-by-*N* matrix *A* by the elements of the vector *x*, of length *M*. The *i*-th row of *A* is multiplied by  $x_i$ . This is equivalent to forming

$$A \rightarrow XA$$

where  $X = \text{diag}(x)$ .

int **gsl\_matrix\_add\_constant**(gsl\_matrix \*a, const double x)

This function adds the constant value *x* to the elements of the matrix *a*. The result  $a(i, j) \leftarrow a(i, j) + x$  is stored in *a*.

### 8.4.11 Finding maximum and minimum elements of matrices

The following operations are only defined for real matrices.

double **gsl\_matrix\_max**(const *gsl\_matrix* \*m)

This function returns the maximum value in the matrix *m*.

double **gsl\_matrix\_min**(const *gsl\_matrix* \*m)

This function returns the minimum value in the matrix *m*.

void **gsl\_matrix\_minmax**(const *gsl\_matrix* \*m, double \*min\_out, double \*max\_out)

This function returns the minimum and maximum values in the matrix *m*, storing them in *min\_out* and *max\_out*.

void **gsl\_matrix\_max\_index**(const *gsl\_matrix* \*m, size\_t \*imax, size\_t \*jmax)

This function returns the indices of the maximum value in the matrix *m*, storing them in *imax* and *jmax*. When there are several equal maximum elements then the first element found is returned, searching in row-major order.

void **gsl\_matrix\_min\_index**(const *gsl\_matrix* \*m, size\_t \*imin, size\_t \*jmin)

This function returns the indices of the minimum value in the matrix *m*, storing them in *imin* and *jmin*. When there are several equal minimum elements then the first element found is returned, searching in row-major order.

void **gsl\_matrix\_minmax\_index**(const *gsl\_matrix* \*m, size\_t \*imin, size\_t \*jmin, size\_t \*imax, size\_t \*jmax)

This function returns the indices of the minimum and maximum values in the matrix *m*, storing them in (*imin*, *jmin*) and (*imax*, *jmax*). When there are several equal minimum or maximum elements then the first elements found are returned, searching in row-major order.

### 8.4.12 Matrix properties

The following functions are defined for real and complex matrices. For complex matrices both the real and imaginary parts must satisfy the conditions.

int **gsl\_matrix\_isnull**(const *gsl\_matrix* \*m)

int **gsl\_matrix\_ispos**(const *gsl\_matrix* \*m)

int **gsl\_matrix\_isneg**(const *gsl\_matrix* \*m)

int **gsl\_matrix\_isnonneg**(const *gsl\_matrix* \*m)

These functions return 1 if all the elements of the matrix *m* are zero, strictly positive, strictly negative, or non-negative respectively, and 0 otherwise. To test whether a matrix is positive-definite, use the *Cholesky decomposition*.

int **gsl\_matrix\_equal**(const *gsl\_matrix* \*a, const *gsl\_matrix* \*b)

This function returns 1 if the matrices *a* and *b* are equal (by comparison of element values) and 0 otherwise.

double **gsl\_matrix\_norm1**(const *gsl\_matrix* \*A)

This function returns the 1-norm of the *m*-by-*n* matrix *A*, defined as the maximum column sum,

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |A_{ij}|$$

### 8.4.13 Example programs for matrices

The program below shows how to allocate, initialize and read from a matrix using the functions `gsl_matrix_alloc()`, `gsl_matrix_set()` and `gsl_matrix_get()`.

```
#include <stdio.h>
#include <gsl/gsl_matrix.h>

int
main (void)
{
  int i, j;
  gsl_matrix * m = gsl_matrix_alloc (10, 3);

  for (i = 0; i < 10; i++)
    for (j = 0; j < 3; j++)
      gsl_matrix_set (m, i, j, 0.23 + 100*i + j);

  for (i = 0; i < 100; i++) /* OUT OF RANGE ERROR */
    for (j = 0; j < 3; j++)
      printf ("m(%d,%d) = %g\n", i, j,
             gsl_matrix_get (m, i, j));

  gsl_matrix_free (m);

  return 0;
}
```

Here is the output from the program. The final loop attempts to read outside the range of the matrix `m`, and the error is trapped by the range-checking code in `gsl_matrix_get()`.

```
$ ./a.out
m(0,0) = 0.23
m(0,1) = 1.23
m(0,2) = 2.23
m(1,0) = 100.23
m(1,1) = 101.23
m(1,2) = 102.23
...
m(9,2) = 902.23
gsl: matrix_source.c:13: ERROR: first index out of range
Default GSL error handler invoked.
Aborted (core dumped)
```

The next program shows how to write a matrix to a file.

```
#include <stdio.h>
#include <gsl/gsl_matrix.h>

int
main (void)
{
  int i, j, k = 0;
  gsl_matrix * m = gsl_matrix_alloc (100, 100);
```

(continues on next page)

(continued from previous page)

```

gsl_matrix * a = gsl_matrix_alloc (100, 100);

for (i = 0; i < 100; i++)
  for (j = 0; j < 100; j++)
    gsl_matrix_set (m, i, j, 0.23 + i + j);

{
  FILE * f = fopen ("test.dat", "wb");
  gsl_matrix_fwrite (f, m);
  fclose (f);
}

{
  FILE * f = fopen ("test.dat", "rb");
  gsl_matrix_fread (f, a);
  fclose (f);
}

for (i = 0; i < 100; i++)
  for (j = 0; j < 100; j++)
    {
      double mij = gsl_matrix_get (m, i, j);
      double aij = gsl_matrix_get (a, i, j);
      if (mij != aij) k++;
    }

gsl_matrix_free (m);
gsl_matrix_free (a);

printf ("differences = %d (should be zero)\n", k);
return (k > 0);
}

```

After running this program the file `test.dat` should contain the elements of `m`, written in binary format. The matrix which is read back in using the function `gsl_matrix_fread()` should be exactly equal to the original matrix.

The following program demonstrates the use of vector views. The program computes the column norms of a matrix.

```

#include <math.h>
#include <stdio.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>

int
main (void)
{
  size_t i, j;

  gsl_matrix *m = gsl_matrix_alloc (10, 10);

  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      gsl_matrix_set (m, i, j, sin (i) + cos (j));

```

(continues on next page)

(continued from previous page)

```

for (j = 0; j < 10; j++)
{
    gsl_vector_view column = gsl_matrix_column (m, j);
    double d;

    d = gsl_blas_dnorm2 (&column.vector);

    printf ("matrix column %zu, norm = %g\n", j, d);
}

gsl_matrix_free (m);

return 0;
}

```

Here is the output of the program,

```

matrix column 0, norm = 4.31461
matrix column 1, norm = 3.1205
matrix column 2, norm = 2.19316
matrix column 3, norm = 3.26114
matrix column 4, norm = 2.53416
matrix column 5, norm = 2.57281
matrix column 6, norm = 4.20469
matrix column 7, norm = 3.65202
matrix column 8, norm = 2.08524
matrix column 9, norm = 3.07313

```

The results can be confirmed using GNU octave:

```

$ octave
GNU Octave, version 2.0.16.92
octave> m = sin(0:9)' * ones(1,10)
           + ones(10,1) * cos(0:9);
octave> sqrt(sum(m.^2))
ans =
  4.3146  3.1205  2.1932  3.2611  2.5342  2.5728
  4.2047  3.6520  2.0852  3.0731

```

## 8.4.14 References and Further Reading

The block, vector and matrix objects in GSL follow the `valarray` model of C++. A description of this model can be found in the following reference,

- B. Stroustrup, *The C++ Programming Language* (3rd Ed), Section 22.4 Vector Arithmetic. Addison-Wesley 1997, ISBN 0-201-88954-4.





## PERMUTATIONS

This chapter describes functions for creating and manipulating permutations. A permutation  $p$  is represented by an array of  $n$  integers in the range 0 to  $n - 1$ , where each value  $p_i$  occurs once and only once. The application of a permutation  $p$  to a vector  $v$  yields a new vector  $v'$  where  $v'_i = v_{p_i}$ . For example, the array  $(0, 1, 3, 2)$  represents a permutation which exchanges the last two elements of a four element vector. The corresponding identity permutation is  $(0, 1, 2, 3)$ .

Note that the permutations produced by the linear algebra routines correspond to the exchange of matrix columns, and so should be considered as applying to row-vectors in the form  $v' = vP$  rather than column-vectors, when permuting the elements of a vector.

The functions described in this chapter are defined in the header file `gsl_permutation.h`.

### 9.1 The Permutation struct

type `gsl_permutation`

A permutation is defined by a structure containing two components, the size of the permutation and a pointer to the permutation array. The elements of the permutation array are all of type `size_t`. The `gsl_permutation` structure looks like this:

```
typedef struct
{
    size_t size;
    size_t * data;
} gsl_permutation;
```

### 9.2 Permutation allocation

`gsl_permutation *gsl_permutation_alloc(size_t n)`

This function allocates memory for a new permutation of size  $n$ . The permutation is not initialized and its elements are undefined. Use the function `gsl_permutation_calloc()` if you want to create a permutation which is initialized to the identity. A null pointer is returned if insufficient memory is available to create the permutation.

`gsl_permutation *gsl_permutation_calloc(size_t n)`

This function allocates memory for a new permutation of size  $n$  and initializes it to the identity. A null pointer is returned if insufficient memory is available to create the permutation.

void `gsl_permutation_init(gsl_permutation *p)`

This function initializes the permutation  $p$  to the identity, i.e.  $(0, 1, 2, \dots, n - 1)$ .

void **gsl\_permutation\_free**(*gsl\_permutation* \*p)

This function frees all the memory used by the permutation *p*.

int **gsl\_permutation\_memcpy**(*gsl\_permutation* \*dest, const *gsl\_permutation* \*src)

This function copies the elements of the permutation *src* into the permutation *dest*. The two permutations must have the same size.

## 9.3 Accessing permutation elements

The following functions can be used to access and manipulate permutations.

size\_t **gsl\_permutation\_get**(const *gsl\_permutation* \*p, const size\_t i)

This function returns the value of the *i*-th element of the permutation *p*. If *i* lies outside the allowed range of 0 to  $n - 1$  then the error handler is invoked and 0 is returned. An inline version of this function is used when `HAVE_INLINE` is defined.

int **gsl\_permutation\_swap**(*gsl\_permutation* \*p, const size\_t i, const size\_t j)

This function exchanges the *i*-th and *j*-th elements of the permutation *p*.

## 9.4 Permutation properties

size\_t **gsl\_permutation\_size**(const *gsl\_permutation* \*p)

This function returns the size of the permutation *p*.

size\_t \***gsl\_permutation\_data**(const *gsl\_permutation* \*p)

This function returns a pointer to the array of elements in the permutation *p*.

int **gsl\_permutation\_valid**(const *gsl\_permutation* \*p)

This function checks that the permutation *p* is valid. The *n* elements should contain each of the numbers 0 to  $n - 1$  once and only once.

## 9.5 Permutation functions

void **gsl\_permutation\_reverse**(*gsl\_permutation* \*p)

This function reverses the elements of the permutation *p*.

int **gsl\_permutation\_inverse**(*gsl\_permutation* \*inv, const *gsl\_permutation* \*p)

This function computes the inverse of the permutation *p*, storing the result in *inv*.

int **gsl\_permutation\_next**(*gsl\_permutation* \*p)

This function advances the permutation *p* to the next permutation in lexicographic order and returns `GSL_SUCCESS`. If no further permutations are available it returns `GSL_FAILURE` and leaves *p* unmodified. Starting with the identity permutation and repeatedly applying this function will iterate through all possible permutations of a given order.

int **gsl\_permutation\_prev**(*gsl\_permutation* \*p)

This function steps backwards from the permutation *p* to the previous permutation in lexicographic order, returning `GSL_SUCCESS`. If no previous permutation is available it returns `GSL_FAILURE` and leaves *p* unmodified.

## 9.6 Applying Permutations

The following functions are defined in the header files `gsl_permute.h` and `gsl_permute_vector.h`.

int **gsl\_permute**(const size\_t \*p, double \*data, size\_t stride, size\_t n)

This function applies the permutation  $p$  to the array  $data$  of size  $n$  with stride  $stride$ .

int **gsl\_permute\_inverse**(const size\_t \*p, double \*data, size\_t stride, size\_t n)

This function applies the inverse of the permutation  $p$  to the array  $data$  of size  $n$  with stride  $stride$ .

int **gsl\_permute\_vector**(const *gsl\_permutation* \*p, *gsl\_vector* \*v)

This function applies the permutation  $p$  to the elements of the vector  $v$ , considered as a row-vector acted on by a permutation matrix from the right,  $v' = vP$ . The  $j$ -th column of the permutation matrix  $P$  is given by the  $p_j$ -th column of the identity matrix. The permutation  $p$  and the vector  $v$  must have the same length.

int **gsl\_permute\_vector\_inverse**(const *gsl\_permutation* \*p, *gsl\_vector* \*v)

This function applies the inverse of the permutation  $p$  to the elements of the vector  $v$ , considered as a row-vector acted on by an inverse permutation matrix from the right,  $v' = vP^T$ . Note that for permutation matrices the inverse is the same as the transpose. The  $j$ -th column of the permutation matrix  $P$  is given by the  $p_j$ -th column of the identity matrix. The permutation  $p$  and the vector  $v$  must have the same length.

int **gsl\_permute\_matrix**(const *gsl\_permutation* \*p, *gsl\_matrix* \*A)

This function applies the permutation  $p$  to the matrix  $A$  from the right,  $A' = AP$ . The  $j$ -th column of the permutation matrix  $P$  is given by the  $p_j$ -th column of the identity matrix. This effectively permutes the columns of  $A$  according to the permutation  $p$ , and so the number of columns of  $A$  must equal the size of the permutation  $p$ .

int **gsl\_permutation\_mul**(*gsl\_permutation* \*p, const *gsl\_permutation* \*pa, const *gsl\_permutation* \*pb)

This function combines the two permutations  $pa$  and  $pb$  into a single permutation  $p$ , where  $p = pa * pb$ . The permutation  $p$  is equivalent to applying  $pb$  first and then  $pa$ .

## 9.7 Reading and writing permutations

The library provides functions for reading and writing permutations to a file as binary data or formatted text.

int **gsl\_permutation\_fwrite**(FILE \*stream, const *gsl\_permutation* \*p)

This function writes the elements of the permutation  $p$  to the stream  $stream$  in binary format. The function returns `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl\_permutation\_fread**(FILE \*stream, *gsl\_permutation* \*p)

This function reads into the permutation  $p$  from the open stream  $stream$  in binary format. The permutation  $p$  must be preallocated with the correct length since the function uses the size of  $p$  to determine how many bytes to read. The function returns `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl\_permutation\_fprintf**(FILE \*stream, const *gsl\_permutation* \*p, const char \*format)

This function writes the elements of the permutation  $p$  line-by-line to the stream  $stream$  using the format specifier  $format$ , which should be suitable for a type of `size_t`. In ISO C99 the type modifier `z` represents `size_t`, so `"%zu\n"` is a suitable format<sup>1</sup>. The function returns `GSL_EFAILED` if there was a problem writing to the file.

int **gsl\_permutation\_fscanf**(FILE \*stream, *gsl\_permutation* \*p)

This function reads formatted data from the stream  $stream$  into the permutation  $p$ . The permutation  $p$  must be preallocated with the correct length since the function uses the size of  $p$  to determine how many numbers to read. The function returns `GSL_EFAILED` if there was a problem reading from the file.

<sup>1</sup> In versions of the GNU C library prior to the ISO C99 standard, the type modifier `Z` was used instead.

## 9.8 Permutations in cyclic form

A permutation can be represented in both *linear* and *cyclic* notations. The functions described in this section convert between the two forms. The linear notation is an index mapping, and has already been described above. The cyclic notation expresses a permutation as a series of circular rearrangements of groups of elements, or *cycles*.

For example, under the cycle (1 2 3), 1 is replaced by 2, 2 is replaced by 3 and 3 is replaced by 1 in a circular fashion. Cycles of different sets of elements can be combined independently, for example (1 2 3) (4 5) combines the cycle (1 2 3) with the cycle (4 5), which is an exchange of elements 4 and 5. A cycle of length one represents an element which is unchanged by the permutation and is referred to as a *singleton*.

It can be shown that every permutation can be decomposed into combinations of cycles. The decomposition is not unique, but can always be rearranged into a standard *canonical form* by a reordering of elements. The library uses the canonical form defined in Knuth's *Art of Computer Programming* (Vol 1, 3rd Ed, 1997) Section 1.3.3, p.178.

The procedure for obtaining the canonical form given by Knuth is,

1. Write all singleton cycles explicitly
2. Within each cycle, put the smallest number first
3. Order the cycles in decreasing order of the first number in the cycle.

For example, the linear representation (2 4 3 0 1) is represented as (1 4) (0 2 3) in canonical form. The permutation corresponds to an exchange of elements 1 and 4, and rotation of elements 0, 2 and 3.

The important property of the canonical form is that it can be reconstructed from the contents of each cycle without the brackets. In addition, by removing the brackets it can be considered as a linear representation of a different permutation. In the example given above the permutation (2 4 3 0 1) would become (1 4 0 2 3). This mapping has many applications in the theory of permutations.

int **gsl\_permutation\_linear\_to\_canonical**(*gsl\_permutation* \*q, const *gsl\_permutation* \*p)

This function computes the canonical form of the permutation *p* and stores it in the output argument *q*.

int **gsl\_permutation\_canonical\_to\_linear**(*gsl\_permutation* \*p, const *gsl\_permutation* \*q)

This function converts a permutation *q* in canonical form back into linear form storing it in the output argument *p*.

size\_t **gsl\_permutation\_inversions**(const *gsl\_permutation* \*p)

This function counts the number of inversions in the permutation *p*. An inversion is any pair of elements that are not in order. For example, the permutation 2031 has three inversions, corresponding to the pairs (2,0) (2,1) and (3,1). The identity permutation has no inversions.

size\_t **gsl\_permutation\_linear\_cycles**(const *gsl\_permutation* \*p)

This function counts the number of cycles in the permutation *p*, given in linear form.

size\_t **gsl\_permutation\_canonical\_cycles**(const *gsl\_permutation* \*q)

This function counts the number of cycles in the permutation *q*, given in canonical form.

## 9.9 Examples

The example program below creates a random permutation (by shuffling the elements of the identity) and finds its inverse.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

(continues on next page)

(continued from previous page)

```

#include <gsl/gsl_permutation.h>

int
main (void)
{
  const size_t N = 10;
  const gsl_rng_type * T;
  gsl_rng * r;

  gsl_permutation * p = gsl_permutation_alloc (N);
  gsl_permutation * q = gsl_permutation_alloc (N);

  gsl_rng_env_setup();
  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  printf ("initial permutation:");
  gsl_permutation_init (p);
  gsl_permutation_fprintf (stdout, p, " %u");
  printf ("\n");

  printf (" random permutation:");
  gsl_ran_shuffle (r, p->data, N, sizeof(size_t));
  gsl_permutation_fprintf (stdout, p, " %u");
  printf ("\n");

  printf ("inverse permutation:");
  gsl_permutation_inverse (q, p);
  gsl_permutation_fprintf (stdout, q, " %u");
  printf ("\n");

  gsl_permutation_free (p);
  gsl_permutation_free (q);
  gsl_rng_free (r);

  return 0;
}

```

Here is the output from the program:

```

$ ./a.out
initial permutation: 0 1 2 3 4 5 6 7 8 9
 random permutation: 1 3 5 2 7 6 0 4 9 8
inverse permutation: 6 0 3 1 7 2 5 4 9 8

```

The random permutation  $p[i]$  and its inverse  $q[i]$  are related through the identity  $p[q[i]] = i$ , which can be verified from the output.

The next example program steps forwards through all possible third order permutations, starting from the identity,

```

#include <stdio.h>
#include <gsl/gsl_permutation.h>

```

(continues on next page)

(continued from previous page)

```
int
main (void)
{
  gsl_permutation * p = gsl_permutation_alloc (3);

  gsl_permutation_init (p);

  do
  {
    gsl_permutation_fprintf (stdout, p, " %u");
    printf ("\n");
  }
  while (gsl_permutation_next(p) == GSL_SUCCESS);

  gsl_permutation_free (p);

  return 0;
}
```

Here is the output from the program:

```
$ ./a.out
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

The permutations are generated in lexicographic order. To reverse the sequence, begin with the final permutation (which is the reverse of the identity) and replace `gsl_permutation_next()` with `gsl_permutation_prev()`.

## 9.10 References and Further Reading

The subject of permutations is covered extensively in the following,

- Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching* (Vol 3, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850.

For the definition of the *canonical form* see,

- Donald E. Knuth, *The Art of Computer Programming: Fundamental Algorithms* (Vol 1, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850. Section 1.3.3, An Unusual Correspondence, p.178–179.

## COMBINATIONS

This chapter describes functions for creating and manipulating combinations. A combination  $c$  is represented by an array of  $k$  integers in the range 0 to  $n - 1$ , where each value  $c_i$  occurs at most once. The combination  $c$  corresponds to indices of  $k$  elements chosen from an  $n$  element vector. Combinations are useful for iterating over all  $k$ -element subsets of a set.

The functions described in this chapter are defined in the header file `gsl_combination.h`.

### 10.1 The Combination struct

type **gsl\_combination**

A combination is defined by a structure containing three components, the values of  $n$  and  $k$ , and a pointer to the combination array. The elements of the combination array are all of type `size_t`, and are stored in increasing order. The `gsl_combination` structure looks like this:

```
typedef struct
{
    size_t n;
    size_t k;
    size_t *data;
} gsl_combination;
```

### 10.2 Combination allocation

`gsl_combination *gsl_combination_alloc(size_t n, size_t k)`

This function allocates memory for a new combination with parameters  $n$ ,  $k$ . The combination is not initialized and its elements are undefined. Use the function `gsl_combination_calloc()` if you want to create a combination which is initialized to the lexicographically first combination. A null pointer is returned if insufficient memory is available to create the combination.

`gsl_combination *gsl_combination_calloc(size_t n, size_t k)`

This function allocates memory for a new combination with parameters  $n$ ,  $k$  and initializes it to the lexicographically first combination. A null pointer is returned if insufficient memory is available to create the combination.

void **gsl\_combination\_init\_first**(`gsl_combination *c`)

This function initializes the combination  $c$  to the lexicographically first combination, i.e.  $(0, 1, 2, \dots, k - 1)$ .

void **gsl\_combination\_init\_last**(`gsl_combination *c`)

This function initializes the combination  $c$  to the lexicographically last combination, i.e.  $(n - k, n - k + 1, \dots, n - 1)$ .

void **gsl\_combination\_free**(*gsl\_combination* \*c)

This function frees all the memory used by the combination *c*.

int **gsl\_combination\_memcpy**(*gsl\_combination* \*dest, const *gsl\_combination* \*src)

This function copies the elements of the combination *src* into the combination *dest*. The two combinations must have the same size.

## 10.3 Accessing combination elements

The following function can be used to access the elements of a combination.

size\_t **gsl\_combination\_get**(const *gsl\_combination* \*c, const size\_t i)

This function returns the value of the *i*-th element of the combination *c*. If *i* lies outside the allowed range of 0 to  $k - 1$  then the error handler is invoked and 0 is returned. An inline version of this function is used when `HAVE_INLINE` is defined.

## 10.4 Combination properties

size\_t **gsl\_combination\_n**(const *gsl\_combination* \*c)

This function returns the range (*n*) of the combination *c*.

size\_t **gsl\_combination\_k**(const *gsl\_combination* \*c)

This function returns the number of elements (*k*) in the combination *c*.

size\_t \***gsl\_combination\_data**(const *gsl\_combination* \*c)

This function returns a pointer to the array of elements in the combination *c*.

int **gsl\_combination\_valid**(*gsl\_combination* \*c)

This function checks that the combination *c* is valid. The *k* elements should lie in the range 0 to  $n - 1$ , with each value occurring once at most and in increasing order.

## 10.5 Combination functions

int **gsl\_combination\_next**(*gsl\_combination* \*c)

This function advances the combination *c* to the next combination in lexicographic order and returns `GSL_SUCCESS`. If no further combinations are available it returns `GSL_FAILURE` and leaves *c* unmodified. Starting with the first combination and repeatedly applying this function will iterate through all possible combinations of a given order.

int **gsl\_combination\_prev**(*gsl\_combination* \*c)

This function steps backwards from the combination *c* to the previous combination in lexicographic order, returning `GSL_SUCCESS`. If no previous combination is available it returns `GSL_FAILURE` and leaves *c* unmodified.



## 10.6 Reading and writing combinations

The library provides functions for reading and writing combinations to a file as binary data or formatted text.

int `gsl_combination_fwrite`(FILE \*stream, const *gsl\_combination* \*c)

This function writes the elements of the combination *c* to the stream *stream* in binary format. The function returns `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int `gsl_combination_fread`(FILE \*stream, *gsl\_combination* \*c)

This function reads elements from the open stream *stream* into the combination *c* in binary format. The combination *c* must be preallocated with correct values of *n* and *k* since the function uses the size of *c* to determine how many bytes to read. The function returns `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int `gsl_combination_fprintf`(FILE \*stream, const *gsl\_combination* \*c, const char \*format)

This function writes the elements of the combination *c* line-by-line to the stream *stream* using the format specifier *format*, which should be suitable for a type of `size_t`. In ISO C99 the type modifier *z* represents `size_t`, so `"%zu\n"` is a suitable format<sup>1</sup>. The function returns `GSL_EFAILED` if there was a problem writing to the file.

int `gsl_combination_fscanf`(FILE \*stream, *gsl\_combination* \*c)

This function reads formatted data from the stream *stream* into the combination *c*. The combination *c* must be preallocated with correct values of *n* and *k* since the function uses the size of *c* to determine how many numbers to read. The function returns `GSL_EFAILED` if there was a problem reading from the file.

## 10.7 Examples

The example program below prints all subsets of the set 0, 1, 2, 3 ordered by size. Subsets of the same size are ordered lexicographically.

```
#include <stdio.h>
#include <gsl/gsl_combination.h>

int
main (void)
{
    gsl_combination * c;
    size_t i;

    printf ("All subsets of {0,1,2,3} by size:\n") ;
    for (i = 0; i <= 4; i++)
    {
        c = gsl_combination_calloc (4, i);
        do
        {
            printf ("{");
            gsl_combination_fprintf (stdout, c, " %u");
            printf (" }\n");
        }
        while (gsl_combination_next (c) == GSL_SUCCESS);
    }
}
```

(continues on next page)

<sup>1</sup> In versions of the GNU C library prior to the ISO C99 standard, the type modifier *Z* was used instead.

(continued from previous page)

```
    gsl_combination_free (c);
}

return 0;
}
```

Here is the output from the program,

```
All subsets of {0,1,2,3} by size:
{ }
{ 0 }
{ 1 }
{ 2 }
{ 3 }
{ 0 1 }
{ 0 2 }
{ 0 3 }
{ 1 2 }
{ 1 3 }
{ 2 3 }
{ 0 1 2 }
{ 0 1 3 }
{ 0 2 3 }
{ 1 2 3 }
{ 0 1 2 3 }
```

All 16 subsets are generated, and the subsets of each size are sorted lexicographically.

## 10.8 References and Further Reading

Further information on combinations can be found in,

- Donald L. Kreher, Douglas R. Stinson, Combinatorial Algorithms: Generation, Enumeration and Search, 1998, CRC Press LLC, ISBN 084933988X

## MULTISETS

This chapter describes functions for creating and manipulating multisets. A multiset  $c$  is represented by an array of  $k$  integers in the range 0 to  $n - 1$ , where each value  $c_i$  may occur more than once. The multiset  $c$  corresponds to indices of  $k$  elements chosen from an  $n$  element vector with replacement. In mathematical terms,  $n$  is the cardinality of the multiset while  $k$  is the maximum multiplicity of any value. Multisets are useful, for example, when iterating over the indices of a  $k$ -th order symmetric tensor in  $n$ -space.

The functions described in this chapter are defined in the header file `gsl_multiset.h`.

### 11.1 The Multiset struct

type `gsl_multiset`

A multiset is defined by a structure containing three components, the values of  $n$  and  $k$ , and a pointer to the multiset array. The elements of the multiset array are all of type `size_t`, and are stored in increasing order. The `gsl_multiset` structure looks like this:

```
typedef struct
{
    size_t n;
    size_t k;
    size_t *data;
} gsl_multiset;
```

### 11.2 Multiset allocation

`gsl_multiset *gsl_multiset_alloc(size_t n, size_t k)`

This function allocates memory for a new multiset with parameters  $n$ ,  $k$ . The multiset is not initialized and its elements are undefined. Use the function `gsl_multiset_calloc()` if you want to create a multiset which is initialized to the lexicographically first multiset element. A null pointer is returned if insufficient memory is available to create the multiset.

`gsl_multiset *gsl_multiset_calloc(size_t n, size_t k)`

This function allocates memory for a new multiset with parameters  $n$ ,  $k$  and initializes it to the lexicographically first multiset element. A null pointer is returned if insufficient memory is available to create the multiset.

void `gsl_multiset_init_first(gsl_multiset *c)`

This function initializes the multiset  $c$  to the lexicographically first multiset element, i.e. 0 repeated  $k$  times.

void `gsl_multiset_init_last(gsl_multiset *c)`

This function initializes the multiset  $c$  to the lexicographically last multiset element, i.e.  $n - 1$  repeated  $k$  times.

void **gsl\_multiset\_free**(*gsl\_multiset* \*c)

This function frees all the memory used by the multiset *c*.

int **gsl\_multiset\_memcpy**(*gsl\_multiset* \*dest, const *gsl\_multiset* \*src)

This function copies the elements of the multiset *src* into the multiset *dest*. The two multisets must have the same size.

## 11.3 Accessing multiset elements

The following function can be used to access the elements of a multiset.

size\_t **gsl\_multiset\_get**(const *gsl\_multiset* \*c, const size\_t i)

This function returns the value of the *i*-th element of the multiset *c*. If *i* lies outside the allowed range of 0 to  $k - 1$  then the error handler is invoked and 0 is returned. An inline version of this function is used when `HAVE_INLINE` is defined.

## 11.4 Multiset properties

size\_t **gsl\_multiset\_n**(const *gsl\_multiset* \*c)

This function returns the range (*n*) of the multiset *c*.

size\_t **gsl\_multiset\_k**(const *gsl\_multiset* \*c)

This function returns the number of elements (*k*) in the multiset *c*.

size\_t \***gsl\_multiset\_data**(const *gsl\_multiset* \*c)

This function returns a pointer to the array of elements in the multiset *c*.

int **gsl\_multiset\_valid**(*gsl\_multiset* \*c)

This function checks that the multiset *c* is valid. The *k* elements should lie in the range 0 to  $n - 1$ , with each value occurring in nondecreasing order.

## 11.5 Multiset functions

int **gsl\_multiset\_next**(*gsl\_multiset* \*c)

This function advances the multiset *c* to the next multiset element in lexicographic order and returns `GSL_SUCCESS`. If no further multisets elements are available it returns `GSL_FAILURE` and leaves *c* unmodified. Starting with the first multiset and repeatedly applying this function will iterate through all possible multisets of a given order.

int **gsl\_multiset\_prev**(*gsl\_multiset* \*c)

This function steps backwards from the multiset *c* to the previous multiset element in lexicographic order, returning `GSL_SUCCESS`. If no previous multiset is available it returns `GSL_FAILURE` and leaves *c* unmodified.

## 11.6 Reading and writing multisets

The library provides functions for reading and writing multisets to a file as binary data or formatted text.

int `gsl_multiset_fwrite`(FILE \*stream, const *gsl\_multiset* \*c)

This function writes the elements of the multiset *c* to the stream *stream* in binary format. The function returns `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int `gsl_multiset_fread`(FILE \*stream, *gsl\_multiset* \*c)

This function reads elements from the open stream *stream* into the multiset *c* in binary format. The multiset *c* must be preallocated with correct values of *n* and *k* since the function uses the size of *c* to determine how many bytes to read. The function returns `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int `gsl_multiset_fprintf`(FILE \*stream, const *gsl\_multiset* \*c, const char \*format)

This function writes the elements of the multiset *c* line-by-line to the stream *stream* using the format specifier *format*, which should be suitable for a type of `size_t`. In ISO C99 the type modifier `z` represents `size_t`, so `"%zu\n"` is a suitable format<sup>1</sup>. The function returns `GSL_EFAILED` if there was a problem writing to the file.

int `gsl_multiset_fscanf`(FILE \*stream, *gsl\_multiset* \*c)

This function reads formatted data from the stream *stream* into the multiset *c*. The multiset *c* must be preallocated with correct values of *n* and *k* since the function uses the size of *c* to determine how many numbers to read. The function returns `GSL_EFAILED` if there was a problem reading from the file.

## 11.7 Examples

The example program below prints all multisets elements containing the values 0, 1, 2, 3 ordered by size. Multiset elements of the same size are ordered lexicographically.

```
#include <stdio.h>
#include <gsl/gsl_multiset.h>

int
main (void)
{
    gsl_multiset * c;
    size_t i;

    printf ("All multisets of {0,1,2,3} by size:\n") ;
    for (i = 0; i <= 4; i++)
    {
        c = gsl_multiset_calloc (4, i);
        do
        {
            printf ("{" );
            gsl_multiset_fprintf (stdout, c, " %u");
            printf (" }\n");
        }
        while (gsl_multiset_next (c) == GSL_SUCCESS);
        gsl_multiset_free (c);
    }
}
```

(continues on next page)

<sup>1</sup> In versions of the GNU C library prior to the ISO C99 standard, the type modifier `Z` was used instead.

(continued from previous page)

```
}  
  
return 0;  
}
```

Here is the output from the program,

All multisets of {0,1,2,3} by size:

```
{ }  
{ 0 }  
{ 1 }  
{ 2 }  
{ 3 }  
{ 0 0 }  
{ 0 1 }  
{ 0 2 }  
{ 0 3 }  
{ 1 1 }  
{ 1 2 }  
{ 1 3 }  
{ 2 2 }  
{ 2 3 }  
{ 3 3 }  
{ 0 0 0 }  
{ 0 0 1 }  
{ 0 0 2 }  
{ 0 0 3 }  
{ 0 1 1 }  
{ 0 1 2 }  
{ 0 1 3 }  
{ 0 2 2 }  
{ 0 2 3 }  
{ 0 3 3 }  
{ 1 1 1 }  
{ 1 1 2 }  
{ 1 1 3 }  
{ 1 2 2 }  
{ 1 2 3 }  
{ 1 3 3 }  
{ 2 2 2 }  
{ 2 2 3 }  
{ 2 3 3 }  
{ 3 3 3 }  
{ 0 0 0 0 }  
{ 0 0 0 1 }  
{ 0 0 0 2 }  
{ 0 0 0 3 }  
{ 0 0 1 1 }  
{ 0 0 1 2 }  
{ 0 0 1 3 }  
{ 0 0 2 2 }  
{ 0 0 2 3 }
```

(continues on next page)

(continued from previous page)

```
{ 0 0 3 3 }
{ 0 1 1 1 }
{ 0 1 1 2 }
{ 0 1 1 3 }
{ 0 1 2 2 }
{ 0 1 2 3 }
{ 0 1 3 3 }
{ 0 2 2 2 }
{ 0 2 2 3 }
{ 0 2 3 3 }
{ 0 3 3 3 }
{ 1 1 1 1 }
{ 1 1 1 2 }
{ 1 1 1 3 }
{ 1 1 2 2 }
{ 1 1 2 3 }
{ 1 1 3 3 }
{ 1 2 2 2 }
{ 1 2 2 3 }
{ 1 2 3 3 }
{ 1 3 3 3 }
{ 2 2 2 2 }
{ 2 2 2 3 }
{ 2 2 3 3 }
{ 2 3 3 3 }
{ 3 3 3 3 }
```

All 70 multisets are generated and sorted lexicographically.





## SORTING

This chapter describes functions for sorting data, both directly and indirectly (using an index). All the functions use the *heapsort* algorithm. Heapsort is an  $O(N \log N)$  algorithm which operates in-place and does not require any additional storage. It also provides consistent performance, the running time for its worst-case (ordered data) being not significantly longer than the average and best cases. Note that the heapsort algorithm does not preserve the relative ordering of equal elements—it is an *unstable* sort. However the resulting order of equal elements will be consistent across different platforms when using these functions.

### 12.1 Sorting objects

The following function provides a simple alternative to the standard library function `qsort()`. It is intended for systems lacking `qsort()`, not as a replacement for it. The function `qsort()` should be used whenever possible, as it will be faster and can provide stable ordering of equal elements. Documentation for `qsort()` is available in the GNU C Library Reference Manual.

The functions described in this section are defined in the header file `gsl_heapsort.h`.

void **gsl\_heapsort**(void \*array, size\_t count, size\_t size, *gsl\_comparison\_fn\_t* compare)

This function sorts the *count* elements of the array *array*, each of size *size*, into ascending order using the comparison function *compare*. The type of the comparison function is defined by

type **gsl\_comparison\_fn\_t**

```
int (*gsl_comparison_fn_t) (const void * a, const void * b)
```

A comparison function should return a negative integer if the first argument is less than the second argument, 0 if the two arguments are equal and a positive integer if the first argument is greater than the second argument.

For example, the following function can be used to sort doubles into ascending numerical order.

```
int  
compare_doubles (const double * a, const double * b)  
{  
    if (*a > *b)  
        return 1;  
    else if (*a < *b)  
        return -1;  
    else  
        return 0;  
}
```

The appropriate function call to perform the sort is:

```
gsl_heapsort (array, count, sizeof(double), compare_doubles);
```

Note that unlike `qsort()` the heapsort algorithm cannot be made into a stable sort by pointer arithmetic. The trick of comparing pointers for equal elements in the comparison function does not work for the heapsort algorithm. The heapsort algorithm performs an internal rearrangement of the data which destroys its initial ordering.

int **gsl\_heapsort\_index**(size\_t \*p, const void \*array, size\_t count, size\_t size, gsl\_comparison\_fn\_t compare)

This function indirectly sorts the *count* elements of the array *array*, each of size *size*, into ascending order using the comparison function *compare*. The resulting permutation is stored in *p*, an array of length *n*. The elements of *p* give the index of the array element which would have been stored in that position if the array had been sorted in place. The first element of *p* gives the index of the least element in *array*, and the last element of *p* gives the index of the greatest element in *array*. The array itself is not changed.

## 12.2 Sorting vectors

The following functions will sort the elements of an array or vector, either directly or indirectly. They are defined for all real and integer types using the normal suffix rules. For example, the float versions of the array functions are `gsl_sort_float()` and `gsl_sort_float_index()`. The corresponding vector functions are `gsl_sort_vector_float()` and `gsl_sort_vector_float_index()`. The prototypes are available in the header files `gsl_sort_float.h` `gsl_sort_vector_float.h`. The complete set of prototypes can be included using the header files `gsl_sort.h` and `gsl_sort_vector.h`.

There are no functions for sorting complex arrays or vectors, since the ordering of complex numbers is not uniquely defined. To sort a complex vector by magnitude compute a real vector containing the magnitudes of the complex elements, and sort this vector indirectly. The resulting index gives the appropriate ordering of the original complex vector.

void **gsl\_sort**(double \*data, const size\_t stride, size\_t n)

This function sorts the *n* elements of the array *data* with stride *stride* into ascending numerical order.

void **gsl\_sort2**(double \*data1, const size\_t stride1, double \*data2, const size\_t stride2, size\_t n)

This function sorts the *n* elements of the array *data1* with stride *stride1* into ascending numerical order, while making the same rearrangement of the array *data2* with stride *stride2*, also of size *n*.

void **gsl\_sort\_vector**(gsl\_vector \*v)

This function sorts the elements of the vector *v* into ascending numerical order.

void **gsl\_sort\_vector2**(gsl\_vector \*v1, gsl\_vector \*v2)

This function sorts the elements of the vector *v1* into ascending numerical order, while making the same rearrangement of the vector *v2*.

void **gsl\_sort\_index**(size\_t \*p, const double \*data, size\_t stride, size\_t n)

This function indirectly sorts the *n* elements of the array *data* with stride *stride* into ascending order, storing the resulting permutation in *p*. The array *p* must be allocated with a sufficient length to store the *n* elements of the permutation. The elements of *p* give the index of the array element which would have been stored in that position if the array had been sorted in place. The array *data* is not changed.

int **gsl\_sort\_vector\_index**(gsl\_permutation \*p, const gsl\_vector \*v)

This function indirectly sorts the elements of the vector *v* into ascending order, storing the resulting permutation in *p*. The elements of *p* give the index of the vector element which would have been stored in that position if the vector had been sorted in place. The first element of *p* gives the index of the least element in *v*, and the last element of *p* gives the index of the greatest element in *v*. The vector *v* is not changed.

## 12.3 Selecting the $k$ smallest or largest elements

The functions described in this section select the  $k$  smallest or largest elements of a data set of size  $N$ . The routines use an  $O(kN)$  direct insertion algorithm which is suited to subsets that are small compared with the total size of the dataset. For example, the routines are useful for selecting the 10 largest values from one million data points, but not for selecting the largest 100,000 values. If the subset is a significant part of the total dataset it may be faster to sort all the elements of the dataset directly with an  $O(N \log N)$  algorithm and obtain the smallest or largest values that way.

int **gsl\_sort\_smallest**(double \*dest, size\_t k, const double \*src, size\_t stride, size\_t n)

This function copies the  $k$  smallest elements of the array *src*, of size  $n$  and stride *stride*, in ascending numerical order into the array *dest*. The size  $k$  of the subset must be less than or equal to  $n$ . The data *src* is not modified by this operation.

int **gsl\_sort\_largest**(double \*dest, size\_t k, const double \*src, size\_t stride, size\_t n)

This function copies the  $k$  largest elements of the array *src*, of size  $n$  and stride *stride*, in descending numerical order into the array *dest*.  $k$  must be less than or equal to  $n$ . The data *src* is not modified by this operation.

int **gsl\_sort\_vector\_smallest**(double \*dest, size\_t k, const *gsl\_vector* \*v)

int **gsl\_sort\_vector\_largest**(double \*dest, size\_t k, const *gsl\_vector* \*v)

These functions copy the  $k$  smallest or largest elements of the vector *v* into the array *dest*.  $k$  must be less than or equal to the length of the vector *v*.

The following functions find the indices of the  $k$  smallest or largest elements of a dataset.

int **gsl\_sort\_smallest\_index**(size\_t \*p, size\_t k, const double \*src, size\_t stride, size\_t n)

This function stores the indices of the  $k$  smallest elements of the array *src*, of size  $n$  and stride *stride*, in the array *p*. The indices are chosen so that the corresponding data is in ascending numerical order.  $k$  must be less than or equal to  $n$ . The data *src* is not modified by this operation.

int **gsl\_sort\_largest\_index**(size\_t \*p, size\_t k, const double \*src, size\_t stride, size\_t n)

This function stores the indices of the  $k$  largest elements of the array *src*, of size  $n$  and stride *stride*, in the array *p*. The indices are chosen so that the corresponding data is in descending numerical order.  $k$  must be less than or equal to  $n$ . The data *src* is not modified by this operation.

int **gsl\_sort\_vector\_smallest\_index**(size\_t \*p, size\_t k, const *gsl\_vector* \*v)

int **gsl\_sort\_vector\_largest\_index**(size\_t \*p, size\_t k, const *gsl\_vector* \*v)

These functions store the indices of the  $k$  smallest or largest elements of the vector *v* in the array *p*.  $k$  must be less than or equal to the length of the vector *v*.

## 12.4 Computing the rank

The *rank* of an element is its order in the sorted data. The rank is the inverse of the index permutation,  $p$ . It can be computed using the following algorithm:

```
for (i = 0; i < p->size; i++)
{
    size_t pi = p->data[i];
    rank->data[pi] = i;
}
```

This can be computed directly from the function `gsl_permutation_inverse(rank, p)`.

The following function will print the rank of each element of the vector *v*:

```

void
print_rank (gsl_vector * v)
{
    size_t i;
    size_t n = v->size;
    gsl_permutation * perm = gsl_permutation_alloc(n);
    gsl_permutation * rank = gsl_permutation_alloc(n);

    gsl_sort_vector_index (perm, v);
    gsl_permutation_inverse (rank, perm);

    for (i = 0; i < n; i++)
    {
        double vi = gsl_vector_get(v, i);
        printf ("element = %d, value = %g, rank = %d\n",
                i, vi, rank->data[i]);
    }

    gsl_permutation_free (perm);
    gsl_permutation_free (rank);
}

```

## 12.5 Examples

The following example shows how to use the permutation  $p$  to print the elements of the vector  $v$  in ascending order:

```

gsl_sort_vector_index (p, v);

for (i = 0; i < v->size; i++)
{
    double vpi = gsl_vector_get (v, p->data[i]);
    printf ("order = %d, value = %g\n", i, vpi);
}

```

The next example uses the function `gsl_sort_smallest()` to select the 5 smallest numbers from 100000 uniform random variates stored in an array,

```

#include <gsl/gsl_rng.h>
#include <gsl/gsl_sort_double.h>

int
main (void)
{
    const gsl_rng_type * T;
    gsl_rng * r;

    size_t i, k = 5, N = 100000;

    double * x = malloc (N * sizeof(double));
    double * small = malloc (k * sizeof(double));
}

```

(continues on next page)

(continued from previous page)

```
gsl_rng_env_setup();

T = gsl_rng_default;
r = gsl_rng_alloc (T);

for (i = 0; i < N; i++)
  {
    x[i] = gsl_rng_uniform(r);
  }

gsl_sort_smallest (small, k, x, 1, N);

printf ("%zu smallest values from %zu\n", k, N);

for (i = 0; i < k; i++)
  {
    printf ("%zu: %.18f\n", i, small[i]);
  }

free (x);
free (small);
gsl_rng_free (r);
return 0;
}
```

The output lists the 5 smallest values, in ascending order,

```
5 smallest values from 100000
0: 0.000003489200025797
1: 0.000008199829608202
2: 0.000008953968062997
3: 0.000010712770745158
4: 0.000033531803637743
```

## 12.6 References and Further Reading

The subject of sorting is covered extensively in the following,

- Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching* (Vol 3, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896850.

The Heapsort algorithm is described in the following book,

- Robert Sedgewick, *Algorithms in C*, Addison-Wesley, ISBN 0201514257.



## BLAS SUPPORT

The Basic Linear Algebra Subprograms (BLAS) define a set of fundamental operations on vectors and matrices which can be used to create optimized higher-level linear algebra functionality.

The library provides a low-level layer which corresponds directly to the C-language BLAS standard, referred to here as “CBLAS”, and a higher-level interface for operations on GSL vectors and matrices. Users who are interested in simple operations on GSL vector and matrix objects should use the high-level layer described in this chapter. The functions are declared in the file `gsl_blas.h` and should satisfy the needs of most users.

Note that GSL matrices are implemented using dense-storage so the interface only includes the corresponding dense-storage BLAS functions. The full BLAS functionality for band-format and packed-format matrices is available through the low-level CBLAS interface. Similarly, GSL vectors are restricted to positive strides, whereas the low-level CBLAS interface supports negative strides as specified in the BLAS standard<sup>1</sup>.

The interface for the `gsl_cblas` layer is specified in the file `gsl_cblas.h`. This interface corresponds to the BLAS Technical Forum’s standard for the C interface to legacy BLAS implementations. Users who have access to other conforming CBLAS implementations can use these in place of the version provided by the library. Note that users who have only a Fortran BLAS library can use a CBLAS conformant wrapper to convert it into a CBLAS library. A reference CBLAS wrapper for legacy Fortran implementations exists as part of the CBLAS standard and can be obtained from Netlib. The complete set of CBLAS functions is listed in an *appendix*.

There are three levels of BLAS operations,

<b>Level 1</b>	Vector operations, e.g. $y = \alpha x + y$
<b>Level 2</b>	Matrix-vector operations, e.g. $y = \alpha Ax + \beta y$
<b>Level 3</b>	Matrix-matrix operations, e.g. $C = \alpha AB + C$

Each routine has a name which specifies the operation, the type of matrices involved and their precisions. Some of the most common operations and their names are given below,

<b>DOT</b>	scalar product, $x^T y$
<b>AXPY</b>	vector sum, $\alpha x + y$
<b>MV</b>	matrix-vector product, $Ax$
<b>SV</b>	matrix-vector solve, $inv(A)x$
<b>MM</b>	matrix-matrix product, $AB$
<b>SM</b>	matrix-matrix solve, $inv(A)B$

The types of matrices are,

---

<sup>1</sup> In the low-level CBLAS interface, a negative stride accesses the vector elements in reverse order, i.e. the  $i$ -th element is given by  $(N-i)*|incx|$  for  $incx < 0$ .

<b>GE</b>	general
<b>GB</b>	general band
<b>SY</b>	symmetric
<b>SB</b>	symmetric band
<b>SP</b>	symmetric packed
<b>HE</b>	hermitian
<b>HB</b>	hermitian band
<b>HP</b>	hermitian packed
<b>TR</b>	triangular
<b>TB</b>	triangular band
<b>TP</b>	triangular packed

Each operation is defined for four precisions,

<b>S</b>	single real
<b>D</b>	double real
<b>C</b>	single complex
<b>Z</b>	double complex

Thus, for example, the name SGEMM stands for “single-precision general matrix-matrix multiply” and ZGEMM stands for “double-precision complex matrix-matrix multiply”.

Note that the vector and matrix arguments to BLAS functions must not be aliased, as the results are undefined when the underlying arrays overlap (*Aliasing of arrays*).

## 13.1 GSL BLAS Interface

GSL provides dense vector and matrix objects, based on the relevant built-in types. The library provides an interface to the BLAS operations which apply to these objects. The interface to this functionality is given in the file `gsl_blas.h`.

### 13.1.1 Level 1

int **gsl\_blas\_sdsdot**(float alpha, const gsl\_vector\_float \*x, const gsl\_vector\_float \*y, float \*result)

This function computes the sum  $\alpha + x^T y$  for the vectors  $x$  and  $y$ , returning the result in *result*.

int **gsl\_blas\_sdot**(const gsl\_vector\_float \*x, const gsl\_vector\_float \*y, float \*result)

int **gsl\_blas\_dsdot**(const gsl\_vector\_float \*x, const gsl\_vector\_float \*y, double \*result)

int **gsl\_blas\_ddot**(const *gsl\_vector* \*x, const *gsl\_vector* \*y, double \*result)

These functions compute the scalar product  $x^T y$  for the vectors  $x$  and  $y$ , returning the result in *result*.

int **gsl\_blas\_cdotu**(const gsl\_vector\_complex\_float \*x, const gsl\_vector\_complex\_float \*y, gsl\_complex\_float \*dotu)

int **gsl\_blas\_zdotu**(const gsl\_vector\_complex \*x, const gsl\_vector\_complex \*y, gsl\_complex \*dotu)

These functions compute the complex scalar product  $x^T y$  for the vectors  $x$  and  $y$ , returning the result in *dotu*

int **gsl\_blas\_cdotc**(const gsl\_vector\_complex\_float \*x, const gsl\_vector\_complex\_float \*y, gsl\_complex\_float \*dotc)



int **gsl\_blas\_zdotc**(const gsl\_vector\_complex \*x, const gsl\_vector\_complex \*y, gsl\_complex \*dotc)  
 These functions compute the complex conjugate scalar product  $x^H y$  for the vectors  $x$  and  $y$ , returning the result in *dotc*

float **gsl\_blas\_snorm2**(const gsl\_vector\_float \*x)

double **gsl\_blas\_dnorm2**(const *gsl\_vector* \*x)

These functions compute the Euclidean norm  $\|x\|_2 = \sqrt{\sum x_i^2}$  of the vector  $x$ .

float **gsl\_blas\_scnorm2**(const gsl\_vector\_complex\_float \*x)

double **gsl\_blas\_dznrm2**(const gsl\_vector\_complex \*x)

These functions compute the Euclidean norm of the complex vector  $x$ ,

$$\|x\|_2 = \sqrt{\sum (\Re(x_i)^2 + \Im(x_i)^2)}.$$

float **gsl\_blas\_sasum**(const gsl\_vector\_float \*x)

double **gsl\_blas\_dasum**(const *gsl\_vector* \*x)

These functions compute the absolute sum  $\sum |x_i|$  of the elements of the vector  $x$ .

float **gsl\_blas\_scasum**(const gsl\_vector\_complex\_float \*x)

double **gsl\_blas\_dzasum**(const gsl\_vector\_complex \*x)

These functions compute the sum of the magnitudes of the real and imaginary parts of the complex vector  $x$ ,  $\sum (|\Re(x_i)| + |\Im(x_i)|)$ .

CBLAS\_INDEX\_t **gsl\_blas\_isamax**(const gsl\_vector\_float \*x)

CBLAS\_INDEX\_t **gsl\_blas\_idamax**(const *gsl\_vector* \*x)

CBLAS\_INDEX\_t **gsl\_blas\_icamax**(const gsl\_vector\_complex\_float \*x)

CBLAS\_INDEX\_t **gsl\_blas\_izamax**(const gsl\_vector\_complex \*x)

These functions return the index of the largest element of the vector  $x$ . The largest element is determined by its absolute magnitude for real vectors and by the sum of the magnitudes of the real and imaginary parts  $|\Re(x_i)| + |\Im(x_i)|$  for complex vectors. If the largest value occurs several times then the index of the first occurrence is returned.

int **gsl\_blas\_sswap**(gsl\_vector\_float \*x, gsl\_vector\_float \*y)

int **gsl\_blas\_dswap**(*gsl\_vector* \*x, *gsl\_vector* \*y)

int **gsl\_blas\_cswap**(gsl\_vector\_complex\_float \*x, gsl\_vector\_complex\_float \*y)

int **gsl\_blas\_zswap**(gsl\_vector\_complex \*x, gsl\_vector\_complex \*y)

These functions exchange the elements of the vectors  $x$  and  $y$ .

int **gsl\_blas\_scopy**(const gsl\_vector\_float \*x, gsl\_vector\_float \*y)

int **gsl\_blas\_dcopy**(const *gsl\_vector* \*x, *gsl\_vector* \*y)

int **gsl\_blas\_ccopy**(const gsl\_vector\_complex\_float \*x, gsl\_vector\_complex\_float \*y)

int **gsl\_blas\_zcopy**(const gsl\_vector\_complex \*x, gsl\_vector\_complex \*y)

These functions copy the elements of the vector  $x$  into the vector  $y$ .

int **gsl\_blas\_saxpy**(float alpha, const gsl\_vector\_float \*x, gsl\_vector\_float \*y)

int **gsl\_blas\_daxpy**(double alpha, const *gsl\_vector* \*x, *gsl\_vector* \*y)

int **gsl\_blas\_caxpy**(const gsl\_complex\_float alpha, const gsl\_vector\_complex\_float \*x, gsl\_vector\_complex\_float \*y)

int **gsl\_blas\_zaxpy**(const gsl\_complex alpha, const gsl\_vector\_complex \*x, gsl\_vector\_complex \*y)

These functions compute the sum  $y = \alpha x + y$  for the vectors  $x$  and  $y$ .

void **gsl\_blas\_sscal**(float alpha, gsl\_vector\_float \*x)

void **gsl\_blas\_dscal**(double alpha, *gsl\_vector* \*x)

void **gsl\_blas\_cscal**(const gsl\_complex\_float alpha, gsl\_vector\_complex\_float \*x)

void **gsl\_blas\_zscal**(const gsl\_complex alpha, gsl\_vector\_complex \*x)

void **gsl\_blas\_csscal**(float alpha, gsl\_vector\_complex\_float \*x)

void **gsl\_blas\_zdscal**(double alpha, gsl\_vector\_complex \*x)

These functions rescale the vector  $x$  by the multiplicative factor *alpha*.

int **gsl\_blas\_srotg**(float a[], float b[], float c[], float s[])

int **gsl\_blas\_drotg**(double a[], double b[], double c[], double s[])

These functions compute a Givens rotation  $(c, s)$  which zeroes the vector  $(a, b)$ ,

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r' \\ 0 \end{pmatrix}$$

The variables  $a$  and  $b$  are overwritten by the routine.

int **gsl\_blas\_srot**(gsl\_vector\_float \*x, gsl\_vector\_float \*y, float c, float s)

int **gsl\_blas\_drot**(*gsl\_vector* \*x, *gsl\_vector* \*y, const double c, const double s)

These functions apply a Givens rotation  $(x', y') = (cx + sy, -sx + cy)$  to the vectors  $x, y$ .

int **gsl\_blas\_srotmg**(float d1[], float d2[], float b1[], float b2, float P[])

int **gsl\_blas\_drotmg**(double d1[], double d2[], double b1[], double b2, double P[])

These functions compute a modified Givens transformation. The modified Givens transformation is defined in the original Level-1 BLAS specification, given in the references.

int **gsl\_blas\_srotm**(gsl\_vector\_float \*x, gsl\_vector\_float \*y, const float P[])

int **gsl\_blas\_drotm**(*gsl\_vector* \*x, *gsl\_vector* \*y, const double P[])

These functions apply a modified Givens transformation.

### 13.1.2 Level 2

int **gsl\_blas\_sgemv**(CBLAS\_TRANSPOSE\_t *TransA*, float *alpha*, const gsl\_matrix\_float \**A*, const gsl\_vector\_float \**x*, float *beta*, gsl\_vector\_float \**y*)

int **gsl\_blas\_dgemv**(CBLAS\_TRANSPOSE\_t *TransA*, double *alpha*, const *gsl\_matrix* \**A*, const *gsl\_vector* \**x*, double *beta*, *gsl\_vector* \**y*)

int **gsl\_blas\_cgemv**(CBLAS\_TRANSPOSE\_t *TransA*, const gsl\_complex\_float *alpha*, const gsl\_matrix\_complex\_float \**A*, const gsl\_vector\_complex\_float \**x*, const gsl\_complex\_float *beta*, gsl\_vector\_complex\_float \**y*)

int **gsl\_blas\_zgemv**(CBLAS\_TRANSPOSE\_t *TransA*, const gsl\_complex *alpha*, const gsl\_matrix\_complex \**A*, const gsl\_vector\_complex \**x*, const gsl\_complex *beta*, gsl\_vector\_complex \**y*)

These functions compute the matrix-vector product and sum  $y = \alpha op(A)x + \beta y$ , where  $op(A) = A, A^T, A^H$  for *TransA* = CblasNoTrans, CblasTrans, CblasConjTrans.

int **gsl\_blas\_strmv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const gsl\_matrix\_float \**A*, gsl\_vector\_float \**x*)

int **gsl\_blas\_dtrmv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const *gsl\_matrix* \**A*, *gsl\_vector* \**x*)

int **gsl\_blas\_ctrmv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const gsl\_matrix\_complex\_float \**A*, gsl\_vector\_complex\_float \**x*)

int **gsl\_blas\_ztrmv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const gsl\_matrix\_complex \**A*, gsl\_vector\_complex \**x*)

These functions compute the matrix-vector product  $x = op(A)x$  for the triangular matrix *A*, where  $op(A) = A, A^T, A^H$  for *TransA* = CblasNoTrans, CblasTrans, CblasConjTrans. When *Uplo* is CblasUpper then the upper triangle of *A* is used, and when *Uplo* is CblasLower then the lower triangle of *A* is used. If *Diag* is CblasNonUnit then the diagonal of the matrix is used, but if *Diag* is CblasUnit then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

int **gsl\_blas\_strsv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const gsl\_matrix\_float \**A*, gsl\_vector\_float \**x*)

int **gsl\_blas\_dtrsv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const *gsl\_matrix* \**A*, *gsl\_vector* \**x*)

int **gsl\_blas\_ctrsv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const gsl\_matrix\_complex\_float \**A*, gsl\_vector\_complex\_float \**x*)

int **gsl\_blas\_ztrsv**(CBLAS\_UPLO\_t *Uplo*, CBLAS\_TRANSPOSE\_t *TransA*, CBLAS\_DIAG\_t *Diag*, const gsl\_matrix\_complex \**A*, gsl\_vector\_complex \**x*)

These functions compute  $inv(op(A))x$  for *x*, where  $op(A) = A, A^T, A^H$  for *TransA* = CblasNoTrans, CblasTrans, CblasConjTrans. When *Uplo* is CblasUpper then the upper triangle of *A* is used, and when *Uplo* is CblasLower then the lower triangle of *A* is used. If *Diag* is CblasNonUnit then the diagonal of the matrix is used, but if *Diag* is CblasUnit then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

int **gsl\_blas\_ssymv**(CBLAS\_UPLO\_t *Uplo*, float *alpha*, const gsl\_matrix\_float \**A*, const gsl\_vector\_float \**x*, float *beta*, gsl\_vector\_float \**y*)

int **gsl\_blas\_dsymv**(CBLAS\_UPLO\_t Uplo, double alpha, const *gsl\_matrix* \*A, const *gsl\_vector* \*x, double beta, *gsl\_vector* \*y)

These functions compute the matrix-vector product and sum  $y = \alpha Ax + \beta y$  for the symmetric matrix  $A$ . Since the matrix  $A$  is symmetric only its upper half or lower half need to be stored. When *Uplo* is CblasUpper then the upper triangle and diagonal of  $A$  are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of  $A$  are used.

int **gsl\_blas\_chemv**(CBLAS\_UPLO\_t Uplo, const gsl\_complex\_float alpha, const gsl\_matrix\_complex\_float \*A, const gsl\_vector\_complex\_float \*x, const gsl\_complex\_float beta, gsl\_vector\_complex\_float \*y)

int **gsl\_blas\_zhemv**(CBLAS\_UPLO\_t Uplo, const gsl\_complex alpha, const gsl\_matrix\_complex \*A, const gsl\_vector\_complex \*x, const gsl\_complex beta, gsl\_vector\_complex \*y)

These functions compute the matrix-vector product and sum  $y = \alpha Ax + \beta y$  for the hermitian matrix  $A$ . Since the matrix  $A$  is hermitian only its upper half or lower half need to be stored. When *Uplo* is CblasUpper then the upper triangle and diagonal of  $A$  are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of  $A$  are used. The imaginary elements of the diagonal are automatically assumed to be zero and are not referenced.

int **gsl\_blas\_sger**(float alpha, const gsl\_vector\_float \*x, const gsl\_vector\_float \*y, gsl\_matrix\_float \*A)

int **gsl\_blas\_dger**(double alpha, const *gsl\_vector* \*x, const *gsl\_vector* \*y, *gsl\_matrix* \*A)

int **gsl\_blas\_cgeru**(const gsl\_complex\_float alpha, const gsl\_vector\_complex\_float \*x, const gsl\_vector\_complex\_float \*y, gsl\_matrix\_complex\_float \*A)

int **gsl\_blas\_zgeru**(const gsl\_complex alpha, const gsl\_vector\_complex \*x, const gsl\_vector\_complex \*y, gsl\_matrix\_complex \*A)

These functions compute the rank-1 update  $A = \alpha xy^T + A$  of the matrix  $A$ .

int **gsl\_blas\_cgerc**(const gsl\_complex\_float alpha, const gsl\_vector\_complex\_float \*x, const gsl\_vector\_complex\_float \*y, gsl\_matrix\_complex\_float \*A)

int **gsl\_blas\_zgerc**(const gsl\_complex alpha, const gsl\_vector\_complex \*x, const gsl\_vector\_complex \*y, gsl\_matrix\_complex \*A)

These functions compute the conjugate rank-1 update  $A = \alpha xy^H + A$  of the matrix  $A$ .

int **gsl\_blas\_ssyr**(CBLAS\_UPLO\_t Uplo, float alpha, const gsl\_vector\_float \*x, gsl\_matrix\_float \*A)

int **gsl\_blas\_dsyr**(CBLAS\_UPLO\_t Uplo, double alpha, const *gsl\_vector* \*x, *gsl\_matrix* \*A)

These functions compute the symmetric rank-1 update  $A = \alpha xx^T + A$  of the symmetric matrix  $A$ . Since the matrix  $A$  is symmetric only its upper half or lower half need to be stored. When *Uplo* is CblasUpper then the upper triangle and diagonal of  $A$  are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of  $A$  are used.

int **gsl\_blas\_cher**(CBLAS\_UPLO\_t Uplo, float alpha, const gsl\_vector\_complex\_float \*x, gsl\_matrix\_complex\_float \*A)

int **gsl\_blas\_zher**(CBLAS\_UPLO\_t Uplo, double alpha, const gsl\_vector\_complex \*x, gsl\_matrix\_complex \*A)

These functions compute the hermitian rank-1 update  $A = \alpha xx^H + A$  of the hermitian matrix  $A$ . Since the matrix  $A$  is hermitian only its upper half or lower half need to be stored. When *Uplo* is CblasUpper then the upper triangle and diagonal of  $A$  are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of  $A$  are used. The imaginary elements of the diagonal are automatically set to zero.

int **gsl\_blas\_ssyrr2**(CBLAS\_UPLO\_t Uplo, float alpha, const gsl\_vector\_float \*x, const gsl\_vector\_float \*y, gsl\_matrix\_float \*A)

int **gsl\_blas\_dsyr2**(CBLAS\_UPLO\_t Uplo, double alpha, const *gsl\_vector* \*x, const *gsl\_vector* \*y, *gsl\_matrix* \*A)

These functions compute the symmetric rank-2 update  $A = \alpha xy^T + \alpha yx^T + A$  of the symmetric matrix  $A$ . Since the matrix  $A$  is symmetric only its upper half or lower half need to be stored. When *Uplo* is CblasUpper then the upper triangle and diagonal of  $A$  are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of  $A$  are used.

int **gsl\_blas\_cher2**(CBLAS\_UPLO\_t Uplo, const gsl\_complex\_float alpha, const gsl\_vector\_complex\_float \*x, const gsl\_vector\_complex\_float \*y, gsl\_matrix\_complex\_float \*A)

int **gsl\_blas\_zher2**(CBLAS\_UPLO\_t Uplo, const gsl\_complex alpha, const gsl\_vector\_complex \*x, const gsl\_vector\_complex \*y, gsl\_matrix\_complex \*A)

These functions compute the hermitian rank-2 update  $A = \alpha xy^H + \alpha^* yx^H + A$  of the hermitian matrix  $A$ . Since the matrix  $A$  is hermitian only its upper half or lower half need to be stored. When *Uplo* is CblasUpper then the upper triangle and diagonal of  $A$  are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of  $A$  are used. The imaginary elements of the diagonal are automatically set to zero.

### 13.1.3 Level 3

int **gsl\_blas\_sgemm**(CBLAS\_TRANSPOSE\_t TransA, CBLAS\_TRANSPOSE\_t TransB, float alpha, const gsl\_matrix\_float \*A, const gsl\_matrix\_float \*B, float beta, gsl\_matrix\_float \*C)

int **gsl\_blas\_dgemm**(CBLAS\_TRANSPOSE\_t TransA, CBLAS\_TRANSPOSE\_t TransB, double alpha, const *gsl\_matrix* \*A, const *gsl\_matrix* \*B, double beta, *gsl\_matrix* \*C)

int **gsl\_blas\_cgemm**(CBLAS\_TRANSPOSE\_t TransA, CBLAS\_TRANSPOSE\_t TransB, const gsl\_complex\_float alpha, const gsl\_matrix\_complex\_float \*A, const gsl\_matrix\_complex\_float \*B, const gsl\_complex\_float beta, gsl\_matrix\_complex\_float \*C)

int **gsl\_blas\_zgemm**(CBLAS\_TRANSPOSE\_t TransA, CBLAS\_TRANSPOSE\_t TransB, const gsl\_complex alpha, const gsl\_matrix\_complex \*A, const gsl\_matrix\_complex \*B, const gsl\_complex beta, gsl\_matrix\_complex \*C)

These functions compute the matrix-matrix product and sum  $C = \alpha op(A)op(B) + \beta C$  where  $op(A) = A, A^T, A^H$  for *TransA* = CblasNoTrans, CblasTrans, CblasConjTrans and similarly for the parameter *TransB*.

int **gsl\_blas\_ssymm**(CBLAS\_SIDE\_t Side, CBLAS\_UPLO\_t Uplo, float alpha, const gsl\_matrix\_float \*A, const gsl\_matrix\_float \*B, float beta, gsl\_matrix\_float \*C)

int **gsl\_blas\_dsymm**(CBLAS\_SIDE\_t Side, CBLAS\_UPLO\_t Uplo, double alpha, const *gsl\_matrix* \*A, const *gsl\_matrix* \*B, double beta, *gsl\_matrix* \*C)

int **gsl\_blas\_csymm**(CBLAS\_SIDE\_t Side, CBLAS\_UPLO\_t Uplo, const gsl\_complex\_float alpha, const gsl\_matrix\_complex\_float \*A, const gsl\_matrix\_complex\_float \*B, const gsl\_complex\_float beta, gsl\_matrix\_complex\_float \*C)

int **gsl\_blas\_zsymm**(CBLAS\_SIDE\_t Side, CBLAS\_UPLO\_t Uplo, const gsl\_complex alpha, const gsl\_matrix\_complex \*A, const gsl\_matrix\_complex \*B, const gsl\_complex beta, gsl\_matrix\_complex \*C)

These functions compute the matrix-matrix product and sum  $C = \alpha AB + \beta C$  for *Side* is CblasLeft and

$C = \alpha BA + \beta C$  for *Side* is CblasRight, where the matrix *A* is symmetric. When *Uplo* is CblasUpper then the upper triangle and diagonal of *A* are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of *A* are used.

```
int gsl_blas_chemm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const
                  gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, const gsl_complex_float
                  beta, gsl_matrix_complex_float *C)
```

```
int gsl_blas_zhemm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex alpha, const
                  gsl_matrix_complex *A, const gsl_matrix_complex *B, const gsl_complex beta,
                  gsl_matrix_complex *C)
```

These functions compute the matrix-matrix product and sum  $C = \alpha AB + \beta C$  for *Side* is CblasLeft and  $C = \alpha BA + \beta C$  for *Side* is CblasRight, where the matrix *A* is hermitian. When *Uplo* is CblasUpper then the upper triangle and diagonal of *A* are used, and when *Uplo* is CblasLower then the lower triangle and diagonal of *A* are used. The imaginary elements of the diagonal are automatically set to zero.

```
int gsl_blas_strmm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, float alpha, const gsl_matrix_float *A, gsl_matrix_float *B)
```

```
int gsl_blas_dtrmm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, double alpha, const gsl_matrix *A, gsl_matrix *B)
```

```
int gsl_blas_ctrmm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, const gsl_complex_float alpha, const gsl_matrix_complex_float *A,
                  gsl_matrix_complex_float *B)
```

```
int gsl_blas_ztrmm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, const gsl_complex alpha, const gsl_matrix_complex *A,
                  gsl_matrix_complex *B)
```

These functions compute the matrix-matrix product  $B = \alpha op(A)B$  for *Side* is CblasLeft and  $B = \alpha Bop(A)$  for *Side* is CblasRight. The matrix *A* is triangular and  $op(A) = A, A^T, A^H$  for *TransA* = CblasNoTrans, CblasTrans, CblasConjTrans. When *Uplo* is CblasUpper then the upper triangle of *A* is used, and when *Uplo* is CblasLower then the lower triangle of *A* is used. If *Diag* is CblasNonUnit then the diagonal of *A* is used, but if *Diag* is CblasUnit then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

```
int gsl_blas_strsm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, float alpha, const gsl_matrix_float *A, gsl_matrix_float *B)
```

```
int gsl_blas_dtrsm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, double alpha, const gsl_matrix *A, gsl_matrix *B)
```

```
int gsl_blas_ctrsm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, const gsl_complex_float alpha, const gsl_matrix_complex_float *A,
                  gsl_matrix_complex_float *B)
```

```
int gsl_blas_ztrsm(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA,
                  CBLAS_DIAG_t Diag, const gsl_complex alpha, const gsl_matrix_complex *A,
                  gsl_matrix_complex *B)
```

These functions compute the inverse-matrix matrix product  $B = \alpha op(inv(A))B$  for *Side* is CblasLeft and  $B = \alpha Bop(inv(A))$  for *Side* is CblasRight. The matrix *A* is triangular and  $op(A) = A, A^T, A^H$  for *TransA* = CblasNoTrans, CblasTrans, CblasConjTrans. When *Uplo* is CblasUpper then the upper triangle of *A* is used, and when *Uplo* is CblasLower then the lower triangle of *A* is used. If *Diag* is CblasNonUnit then the



diagonal of  $A$  is used, but if *Diag* is `CblasUnit` then the diagonal elements of the matrix  $A$  are taken as unity and are not referenced.

```
int gsl_blas_ssyrrk(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_float
                  *A, float beta, gsl_matrix_float *C)
```

```
int gsl_blas_dsyrk(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix *A,
                  double beta, gsl_matrix *C)
```

```
int gsl_blas_csyrk(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const
                  gsl_matrix_complex_float *A, const gsl_complex_float beta, gsl_matrix_complex_float *C)
```

```
int gsl_blas_zsyrk(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const
                  gsl_matrix_complex *A, const gsl_complex beta, gsl_matrix_complex *C)
```

These functions compute a rank- $k$  update of the symmetric matrix  $C$ ,  $C = \alpha AA^T + \beta C$  when *Trans* is `CblasNoTrans` and  $C = \alpha A^T A + \beta C$  when *Trans* is `CblasTrans`. Since the matrix  $C$  is symmetric only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of  $C$  are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of  $C$  are used.

```
int gsl_blas_cherk(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const
                  gsl_matrix_complex_float *A, float beta, gsl_matrix_complex_float *C)
```

```
int gsl_blas_zherk(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const
                  gsl_matrix_complex *A, double beta, gsl_matrix_complex *C)
```

These functions compute a rank- $k$  update of the hermitian matrix  $C$ ,  $C = \alpha AA^H + \beta C$  when *Trans* is `CblasNoTrans` and  $C = \alpha A^H A + \beta C$  when *Trans* is `CblasConjTrans`. Since the matrix  $C$  is hermitian only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of  $C$  are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of  $C$  are used. The imaginary elements of the diagonal are automatically set to zero.

```
int gsl_blas_ssyrr2k(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_float
                   *A, const gsl_matrix_float *B, float beta, gsl_matrix_float *C)
```

```
int gsl_blas_dsyr2k(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix *A,
                   const gsl_matrix *B, double beta, gsl_matrix *C)
```

```
int gsl_blas_csyr2k(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha,
                   const gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, const
                   gsl_complex_float beta, gsl_matrix_complex_float *C)
```

```
int gsl_blas_zsyr2k(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const
                   gsl_matrix_complex *A, const gsl_matrix_complex *B, const gsl_complex beta,
                   gsl_matrix_complex *C)
```

These functions compute a rank- $2k$  update of the symmetric matrix  $C$ ,  $C = \alpha AB^T + \alpha BA^T + \beta C$  when *Trans* is `CblasNoTrans` and  $C = \alpha A^T B + \alpha B^T A + \beta C$  when *Trans* is `CblasTrans`. Since the matrix  $C$  is symmetric only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of  $C$  are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of  $C$  are used.

```
int gsl_blas_cher2k(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha,
                   const gsl_matrix_complex_float *A, const gsl_matrix_complex_float *B, float beta,
                   gsl_matrix_complex_float *C)
```

```
int gsl_blas_zher2k(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const
                  gsl_matrix_complex *A, const gsl_matrix_complex *B, double beta, gsl_matrix_complex
                  *C)
```

These functions compute a rank-2k update of the hermitian matrix  $C$ ,  $C = \alpha AB^H + \alpha^* BA^H + \beta C$  when *Trans* is `CblasNoTrans` and  $C = \alpha A^H B + \alpha^* B^H A + \beta C$  when *Trans* is `CblasConjTrans`. Since the matrix  $C$  is hermitian only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of  $C$  are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of  $C$  are used. The imaginary elements of the diagonal are automatically set to zero.

## 13.2 Examples

The following program computes the product of two matrices using the Level-3 BLAS function DGEMM,

$$\begin{pmatrix} 0.11 & 0.12 & 0.13 \\ 0.21 & 0.22 & 0.23 \end{pmatrix} \begin{pmatrix} 1011 & 1012 \\ 1021 & 1022 \\ 1031 & 1031 \end{pmatrix} = \begin{pmatrix} 367.76 & 368.12 \\ 674.06 & 674.72 \end{pmatrix}$$

The matrices are stored in row major order, according to the C convention for arrays.

```
#include <stdio.h>
#include <gsl/gsl_blas.h>

int
main (void)
{
    double a[] = { 0.11, 0.12, 0.13,
                  0.21, 0.22, 0.23 };

    double b[] = { 1011, 1012,
                  1021, 1022,
                  1031, 1032 };

    double c[] = { 0.00, 0.00,
                  0.00, 0.00 };

    gsl_matrix_view A = gsl_matrix_view_array(a, 2, 3);
    gsl_matrix_view B = gsl_matrix_view_array(b, 3, 2);
    gsl_matrix_view C = gsl_matrix_view_array(c, 2, 2);

    /* Compute C = A B */

    gsl_blas_dgemm (CblasNoTrans, CblasNoTrans,
                   1.0, &A.matrix, &B.matrix,
                   0.0, &C.matrix);

    printf (" [ %g, %g\n", c[0], c[1]);
    printf ("  %g, %g ]\n", c[2], c[3]);

    return 0;
}
```

Here is the output from the program,



[ 367.76, 368.12 674.06, 674.72 ]
--------------------------------------

## 13.3 References and Further Reading

Information on the BLAS standards, including both the legacy and updated interface standards, is available online from the BLAS Homepage and BLAS Technical Forum web-site.

- BLAS Homepage, <http://www.netlib.org/blas/>
- BLAS Technical Forum, <http://www.netlib.org/blas/blast-forum/>

The following papers contain the specifications for Level 1, Level 2 and Level 3 BLAS.

- C. Lawson, R. Hanson, D. Kincaid, F. Krogh, “Basic Linear Algebra Subprograms for Fortran Usage”, ACM Transactions on Mathematical Software, Vol.: 5 (1979), Pages 308–325.
- J.J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson, “An Extended Set of Fortran Basic Linear Algebra Subprograms”, ACM Transactions on Mathematical Software, Vol.: 14, No.: 1 (1988), Pages 1–32.
- J.J. Dongarra, I. Duff, J. DuCroz, S. Hammarling, “A Set of Level 3 Basic Linear Algebra Subprograms”, ACM Transactions on Mathematical Software, Vol.: 16 (1990), Pages 1–28.

Postscript versions of the latter two papers are available from <http://www.netlib.org/blas/>. A CBLAS wrapper for Fortran BLAS libraries is available from the same location.



## LINEAR ALGEBRA

This chapter describes functions for solving linear systems. The library provides linear algebra operations which operate directly on the *gsl\_vector* and *gsl\_matrix* objects. These routines use the standard algorithms from Golub & Van Loan's *Matrix Computations* with Level-1 and Level-2 BLAS calls for efficiency.

The functions described in this chapter are declared in the header file *gsl\_linalg.h*.

### 14.1 LU Decomposition

A general  $M$ -by- $N$  matrix  $A$  has an  $LU$  decomposition

$$PA = LU$$

where  $P$  is an  $M$ -by- $M$  permutation matrix,  $L$  is  $M$ -by- $\min(M, N)$  and  $U$  is  $\min(M, N)$ -by- $N$ . For square matrices,  $L$  is a lower unit triangular matrix and  $U$  is upper triangular. For  $M > N$ ,  $L$  is a unit lower trapezoidal matrix of size  $M$ -by- $N$ . For  $M < N$ ,  $U$  is upper trapezoidal of size  $M$ -by- $N$ . For square matrices this decomposition can be used to convert the linear system  $Ax = b$  into a pair of triangular systems ( $Ly = Pb$ ,  $Ux = y$ ), which can be solved by forward and back-substitution. Note that the  $LU$  decomposition is valid for singular matrices.

```
int gsl_linalg_LU_decomp(gsl_matrix *A, gsl_permutation *p, int *signum)
```

```
int gsl_linalg_complex_LU_decomp(gsl_matrix_complex *A, gsl_permutation *p, int *signum)
```

These functions factorize the matrix  $A$  into the  $LU$  decomposition  $PA = LU$ . On output the diagonal and upper triangular (or trapezoidal) part of the input matrix  $A$  contain the matrix  $U$ . The lower triangular (or trapezoidal) part of the input matrix (excluding the diagonal) contains  $L$ . The diagonal elements of  $L$  are unity, and are not stored.

The permutation matrix  $P$  is encoded in the permutation  $p$  on output. The  $j$ -th column of the matrix  $P$  is given by the  $k$ -th column of the identity matrix, where  $k = p_j$  the  $j$ -th element of the permutation vector. The sign of the permutation is given by *signum*. It has the value  $(-1)^n$ , where  $n$  is the number of interchanges in the permutation.

The algorithm used in the decomposition is Gaussian Elimination with partial pivoting (Golub & Van Loan, *Matrix Computations*, Algorithm 3.4.1), combined with a recursive algorithm based on Level 3 BLAS (Peise and Bientinesi, 2016).

```
int gsl_linalg_LU_solve(const gsl_matrix *LU, const gsl_permutation *p, const gsl_vector *b, gsl_vector *x)
```

```
int gsl_linalg_complex_LU_solve(const gsl_matrix_complex *LU, const gsl_permutation *p, const  
                                gsl_vector_complex *b, gsl_vector_complex *x)
```

These functions solve the square system  $Ax = b$  using the  $LU$  decomposition of  $A$  into  $(LU, p)$  given by *gsl\_linalg\_LU\_decomp()* or *gsl\_linalg\_complex\_LU\_decomp()* as input.

int **gsl\_linalg\_LU\_svx**(const *gsl\_matrix* \*LU, const *gsl\_permutation* \*p, *gsl\_vector* \*x)

int **gsl\_linalg\_complex\_LU\_svx**(const *gsl\_matrix\_complex* \*LU, const *gsl\_permutation* \*p, *gsl\_vector\_complex* \*x)

These functions solve the square system  $Ax = b$  in-place using the precomputed  $LU$  decomposition of  $A$  into  $(LU, p)$ . On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

int **gsl\_linalg\_LU\_refine**(const *gsl\_matrix* \*A, const *gsl\_matrix* \*LU, const *gsl\_permutation* \*p, const *gsl\_vector* \*b, *gsl\_vector* \*x, *gsl\_vector* \*work)

int **gsl\_linalg\_complex\_LU\_refine**(const *gsl\_matrix\_complex* \*A, const *gsl\_matrix\_complex* \*LU, const *gsl\_permutation* \*p, const *gsl\_vector\_complex* \*b, *gsl\_vector\_complex* \*x, *gsl\_vector\_complex* \*work)

These functions apply an iterative improvement to  $x$ , the solution of  $Ax = b$ , from the precomputed  $LU$  decomposition of  $A$  into  $(LU, p)$ . Additional workspace of length  $N$  is required in *work*.

int **gsl\_linalg\_LU\_invert**(const *gsl\_matrix* \*LU, const *gsl\_permutation* \*p, *gsl\_matrix* \*inverse)

int **gsl\_linalg\_complex\_LU\_invert**(const *gsl\_matrix\_complex* \*LU, const *gsl\_permutation* \*p, *gsl\_matrix\_complex* \*inverse)

These functions compute the inverse of a matrix  $A$  from its  $LU$  decomposition  $(LU, p)$ , storing the result in the matrix *inverse*. The inverse is computed by computing the inverses  $U^{-1}$ ,  $L^{-1}$  and finally forming the product  $A^{-1} = U^{-1}L^{-1}P$ . Each step is based on Level 3 BLAS calls.

It is preferable to avoid direct use of the inverse whenever possible, as the linear solver functions can obtain the same result more efficiently and reliably (consult any introductory textbook on numerical linear algebra for details).

int **gsl\_linalg\_LU\_invx**(*gsl\_matrix* \*LU, const *gsl\_permutation* \*p)

int **gsl\_linalg\_complex\_LU\_invx**(*gsl\_matrix\_complex* \*LU, const *gsl\_permutation* \*p)

These functions compute the inverse of a matrix  $A$  from its  $LU$  decomposition  $(LU, p)$ , storing the result in-place in the matrix *LU*. The inverse is computed by computing the inverses  $U^{-1}$ ,  $L^{-1}$  and finally forming the product  $A^{-1} = U^{-1}L^{-1}P$ . Each step is based on Level 3 BLAS calls.

It is preferable to avoid direct use of the inverse whenever possible, as the linear solver functions can obtain the same result more efficiently and reliably (consult any introductory textbook on numerical linear algebra for details).

double **gsl\_linalg\_LU\_det**(*gsl\_matrix* \*LU, int signum)

*gsl\_complex* **gsl\_linalg\_complex\_LU\_det**(*gsl\_matrix\_complex* \*LU, int signum)

These functions compute the determinant of a matrix  $A$  from its  $LU$  decomposition,  $LU$ . The determinant is computed as the product of the diagonal elements of  $U$  and the sign of the row permutation *signum*.

double **gsl\_linalg\_LU\_lndet**(*gsl\_matrix* \*LU)

double **gsl\_linalg\_complex\_LU\_lndet**(*gsl\_matrix\_complex* \*LU)

These functions compute the logarithm of the absolute value of the determinant of a matrix  $A$ ,  $\ln |\det(A)|$ , from its  $LU$  decomposition,  $LU$ . This function may be useful if the direct computation of the determinant would overflow or underflow.

int **gsl\_linalg\_LU\_sgndet**(*gsl\_matrix* \*LU, int signum)

*gsl\_complex* **gsl\_linalg\_complex\_LU\_sgndet**(*gsl\_matrix\_complex* \*LU, int signum)

These functions compute the sign or phase factor of the determinant of a matrix  $A$ ,  $\det(A)/|\det(A)|$ , from its  $LU$  decomposition,  $LU$ .

## 14.2 QR Decomposition

A general rectangular  $M$ -by- $N$  matrix  $A$  has a  $QR$  decomposition into the product of a unitary  $M$ -by- $M$  square matrix  $Q$  (where  $Q^\dagger Q = I$ ) and an  $M$ -by- $N$  right-triangular matrix  $R$ ,

$$A = QR$$

This decomposition can be used to convert the square linear system  $Ax = b$  into the triangular system  $Rx = Q^\dagger b$ , which can be solved by back-substitution. Another use of the  $QR$  decomposition is to compute an orthonormal basis for a set of vectors. The first  $N$  columns of  $Q$  form an orthonormal basis for the range of  $A$ ,  $\text{ran}(A)$ , when  $A$  has full column rank.

When  $M > N$ , the bottom  $M - N$  rows of  $R$  are zero, and so  $A$  can be naturally partitioned as

$$A = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} = Q_1 R_1$$

where  $R_1$  is  $N$ -by- $N$  upper triangular,  $Q_1$  is  $M$ -by- $N$ , and  $Q_2$  is  $M$ -by- $(M - N)$ .  $Q_1 R_1$  is sometimes called the *thin* or *reduced*  $QR$  decomposition. The solution of the least squares problem  $\min_x \|b - Ax\|^2$  when  $A$  has full rank is:

$$x = R_1^{-1} c_1$$

where  $c_1$  is the first  $N$  elements of  $Q^\dagger b$ . If  $A$  is rank deficient, see [QR Decomposition with Column Pivoting](#) and [Complete Orthogonal Decomposition](#).

GSL offers two interfaces for the  $QR$  decomposition. The first proceeds by zeroing out columns below the diagonal of  $A$ , one column at a time using Householder transforms. In this method, the factor  $Q$  is represented as a product of Householder reflectors:

$$Q = H_n \cdots H_2 H_1$$

where each  $H_i = I - \tau_i v_i v_i^\dagger$  for a scalar  $\tau_i$  and column vector  $v_i$ . In this method, functions which compute the full matrix  $Q$  or apply  $Q^\dagger$  to a right hand side vector operate by applying the Householder matrices one at a time using Level 2 BLAS.

The second interface is based on a Level 3 BLAS block recursive algorithm developed by Elmroth and Gustavson. In this case,  $Q$  is written in block form as

$$Q = I - VTV^\dagger$$

where  $V$  is an  $M$ -by- $N$  matrix of the column vectors  $v_i$  and  $T$  is an  $N$ -by- $N$  upper triangular matrix, whose diagonal elements are the  $\tau_i$ . Computing the full  $T$ , while requiring more flops than the Level 2 approach, offers the advantage that all standard operations can take advantage of cache efficient Level 3 BLAS operations, and so this method often performs faster than the Level 2 approach. The functions for the recursive block algorithm have a `_r` suffix, and it is recommended to use this interface for performance critical applications.

```
int gsl_linalg_QR_decomp_r(gsl_matrix *A, gsl_matrix *T)
```

```
int gsl_linalg_complex_QR_decomp_r(gsl_matrix_complex *A, gsl_matrix_complex *T)
```

These functions factor the  $M$ -by- $N$  matrix  $A$  into the  $QR$  decomposition  $A = QR$  using the recursive Level 3 BLAS algorithm of Elmroth and Gustavson. On output the diagonal and upper triangular part of  $A$  contain the matrix  $R$ . The  $N$ -by- $N$  matrix  $T$  stores the upper triangular factor appearing in  $Q$ . The matrix  $Q$  is given by  $Q = I - VTV^\dagger$ , where the elements below the diagonal of  $A$  contain the columns of  $V$  on output.

This algorithm requires  $M \geq N$  and performs best for “tall-skinny” matrices, i.e.  $M \gg N$ .

```
int gsl_linalg_QR_solve_r(const gsl_matrix *QR, const gsl_matrix *T, const gsl_vector *b, gsl_vector *x)
```

```
int gsl_linalg_complex_QR_solve_r(const gsl_matrix_complex *QR, const gsl_matrix_complex *T, const
                                gsl_vector_complex *b, gsl_vector_complex *x)
```

These functions solve the square system  $Ax = b$  using the  $QR$  decomposition of  $A$  held in  $(QR, T)$ . The least-squares solution for rectangular systems can be found using `gsl\_linalg\_QR\_lssolve\_r\(\)` or `gsl\_linalg\_complex\_QR\_lssolve\_r\(\)`.

```
int gsl_linalg_QR_lssolve_r(const gsl_matrix *QR, const gsl_matrix *T, const gsl_vector *b, gsl_vector *x,
                           gsl_vector *work)
```

```
int gsl_linalg_complex_QR_lssolve_r(const gsl_matrix_complex *QR, const gsl_matrix_complex *T, const
                                    gsl_vector_complex *b, gsl_vector_complex *x, gsl_vector_complex
                                    *work)
```

These functions find the least squares solution to the overdetermined system  $Ax = b$  where the matrix  $A$  has more rows than columns. The least squares solution minimizes the Euclidean norm of the residual,  $\|b - Ax\|$ . The routine requires as input the  $QR$  decomposition of  $A$  into  $(QR, T)$  given by `gsl\_linalg\_QR\_decomp\_r\(\)` or `gsl\_linalg\_complex\_QR\_decomp\_r\(\)`. The parameter  $x$  is of length  $M$ . The solution  $x$  is returned in the first  $N$  rows of  $x$ , i.e.  $x = x[0], x[1], \dots, x[N-1]$ . The last  $M - N$  rows of  $x$  contain a vector whose norm is equal to the residual norm  $\|b - Ax\|$ . This similar to the behavior of LAPACK DGELS. Additional workspace of length  $N$  is required in *work*.

```
int gsl_linalg_QR_QTvec_r(const gsl_matrix *QR, const gsl_matrix *T, gsl_vector *v, gsl_vector *work)
```

```
int gsl_linalg_complex_QR_QHvec_r(const gsl_matrix_complex *QR, const gsl_matrix_complex *T,
                                  gsl_vector_complex *v, gsl_vector_complex *work)
```

These functions apply the matrix  $Q^T$  (or  $Q^\dagger$ ) encoded in the decomposition  $(QR, T)$  to the vector  $v$ , storing the result  $Q^T v$  (or  $Q^\dagger v$ ) in  $v$ . The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix  $Q^T$  (or  $Q^\dagger$ ). Additional workspace of size  $N$  is required in *work*.

```
int gsl_linalg_QR_QTmat_r(const gsl_matrix *QR, const gsl_matrix *T, gsl_matrix *B, gsl_matrix *work)
```

This function applies the matrix  $Q^T$  encoded in the decomposition  $(QR, T)$  to the  $M$ -by- $K$  matrix  $B$ , storing the result  $Q^T B$  in  $B$ . The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix  $Q^T$ . Additional workspace of size  $N$ -by- $K$  is required in *work*.

```
int gsl_linalg_QR_unpack_r(const gsl_matrix *QR, const gsl_matrix *T, gsl_matrix *Q, gsl_matrix *R)
```

```
int gsl_linalg_complex_QR_unpack_r(const gsl_matrix_complex *QR, const gsl_matrix_complex *T,
                                   gsl_matrix_complex *Q, gsl_matrix_complex *R)
```

These functions unpack the encoded  $QR$  decomposition  $(QR, T)$  as output from `gsl\_linalg\_QR\_decomp\_r\(\)` or `gsl\_linalg\_complex\_QR\_decomp\_r\(\)` into the matrices  $Q$  and  $R$ , where  $Q$  is  $M$ -by- $M$  and  $R$  is  $N$ -by- $N$ . Note that the full  $R$  matrix is  $M$ -by- $N$ , however the lower trapezoidal portion is zero, so only the upper triangular factor is stored.

```
int gsl_linalg_QR_rcond(const gsl_matrix *QR, double *rcond, gsl_vector *work)
```

This function estimates the reciprocal condition number (using the 1-norm) of the  $R$  factor, stored in the upper triangle of  $QR$ . The reciprocal condition number estimate, defined as  $1/(\|R\|_1 \cdot \|R^{-1}\|_1)$ , is stored in *rcond*. Additional workspace of size  $3N$  is required in *work*.

### 14.2.1 Level 2 Interface

The functions below are for the slower Level 2 interface to the QR decomposition. It is recommended to use these functions only for  $M < N$ , since the Level 3 interface above performs much faster for  $M \geq N$ .

```
int gsl_linalg_QR_decomp(gsl_matrix *A, gsl_vector *tau)
```

```
int gsl_linalg_complex_QR_decomp(gsl_matrix_complex *A, gsl_vector_complex *tau)
```

These functions factor the  $M$ -by- $N$  matrix  $A$  into the  $QR$  decomposition  $A = QR$ . On output the diagonal and upper triangular part of the input matrix contain the matrix  $R$ . The vector  $\tau$  and the columns of the lower triangular part of the matrix  $A$  contain the Householder coefficients and Householder vectors which encode the orthogonal matrix  $Q$ . The vector  $\tau$  must be of length  $N$ . The matrix  $Q$  is related to these components by the product of  $k = \min(M, N)$  reflector matrices,  $Q = H_k \dots H_2 H_1$  where  $H_i = I - \tau_i v_i v_i^\dagger$  and  $v_i$  is the Householder vector  $v_i = (0, \dots, 1, A(i+1, i), A(i+2, i), \dots, A(m, i))$ . This is the same storage scheme as used by LAPACK.

The algorithm used to perform the decomposition is Householder QR (Golub & Van Loan, “Matrix Computations”, Algorithm 5.2.1).

```
int gsl_linalg_QR_solve(const gsl_matrix *QR, const gsl_vector *tau, const gsl_vector *b, gsl_vector *x)
```

```
int gsl_linalg_complex_QR_solve(const gsl_matrix_complex *QR, const gsl_vector_complex *tau, const
                                gsl_vector_complex *b, gsl_vector_complex *x)
```

These functions solve the square system  $Ax = b$  using the  $QR$  decomposition of  $A$  held in  $(QR, \tau)$ . The least-squares solution for rectangular systems can be found using [gsl\\_linalg\\_QR\\_lassolve\(\)](#).

```
int gsl_linalg_QR_svx(const gsl_matrix *QR, const gsl_vector *tau, gsl_vector *x)
```

```
int gsl_linalg_complex_QR_svx(const gsl_matrix_complex *QR, const gsl_vector_complex *tau,
                               gsl_vector_complex *x)
```

These functions solve the square system  $Ax = b$  in-place using the  $QR$  decomposition of  $A$  held in  $(QR, \tau)$ . On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

```
int gsl_linalg_QR_lassolve(const gsl_matrix *QR, const gsl_vector *tau, const gsl_vector *b, gsl_vector *x,
                           gsl_vector *residual)
```

```
int gsl_linalg_complex_QR_lassolve(const gsl_matrix_complex *QR, const gsl_vector_complex *tau, const
                                    gsl_vector_complex *b, gsl_vector_complex *x, gsl_vector_complex
                                    *residual)
```

These functions find the least squares solution to the overdetermined system  $Ax = b$  where the matrix  $A$  has more rows than columns. The least squares solution minimizes the Euclidean norm of the residual,  $\|Ax - b\|$ . The routine requires as input the  $QR$  decomposition of  $A$  into  $(QR, \tau)$  given by [gsl\\_linalg\\_QR\\_decomp\(\)](#) or [gsl\\_linalg\\_complex\\_QR\\_decomp\(\)](#). The solution is returned in  $x$ . The residual is computed as a by-product and stored in *residual*.

```
int gsl_linalg_QR_QTvec(const gsl_matrix *QR, const gsl_vector *tau, gsl_vector *v)
```

```
int gsl_linalg_complex_QR_QHvec(const gsl_matrix_complex *QR, const gsl_vector_complex *tau,
                                 gsl_vector_complex *v)
```

These functions apply the matrix  $Q^T$  (or  $Q^\dagger$ ) encoded in the decomposition  $(QR, \tau)$  to the vector  $v$ , storing the result  $Q^T v$  (or  $Q^\dagger v$ ) in  $v$ . The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix  $Q^T$  (or  $Q^\dagger$ ).

```
int gsl_linalg_QR_Qvec(const gsl_matrix *QR, const gsl_vector *tau, gsl_vector *v)
```

int **gsl\_linalg\_complex\_QR\_Qvec**(const gsl\_matrix\_complex \*QR, const gsl\_vector\_complex \*tau, gsl\_vector\_complex \*v)

These functions apply the matrix  $Q$  encoded in the decomposition  $(QR, \tau)$  to the vector  $v$ , storing the result  $Qv$  in  $v$ . The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix  $Q$ .

int **gsl\_linalg\_QR\_QTmat**(const gsl\_matrix \*QR, const gsl\_vector \*tau, gsl\_matrix \*B)

This function applies the matrix  $Q^T$  encoded in the decomposition  $(QR, \tau)$  to the  $M$ -by- $K$  matrix  $B$ , storing the result  $Q^T B$  in  $B$ . The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix  $Q^T$ .

int **gsl\_linalg\_QR\_Rsolve**(const gsl\_matrix \*QR, const gsl\_vector \*b, gsl\_vector \*x)

This function solves the triangular system  $Rx = b$  for  $x$ . It may be useful if the product  $b' = Q^T b$  has already been computed using `gsl_linalg_QR_QTvec()`.

int **gsl\_linalg\_QR\_Rsvx**(const gsl\_matrix \*QR, gsl\_vector \*x)

This function solves the triangular system  $Rx = b$  for  $x$  in-place. On input  $x$  should contain the right-hand side  $b$  and is replaced by the solution on output. This function may be useful if the product  $b' = Q^T b$  has already been computed using `gsl_linalg_QR_QTvec()`.

int **gsl\_linalg\_QR\_unpack**(const gsl\_matrix \*QR, const gsl\_vector \*tau, gsl\_matrix \*Q, gsl\_matrix \*R)

This function unpacks the encoded  $QR$  decomposition  $(QR, \tau)$  into the matrices  $Q$  and  $R$ , where  $Q$  is  $M$ -by- $M$  and  $R$  is  $M$ -by- $N$ .

int **gsl\_linalg\_QR\_QRsolve**(gsl\_matrix \*Q, gsl\_matrix \*R, const gsl\_vector \*b, gsl\_vector \*x)

This function solves the system  $Rx = Q^T b$  for  $x$ . It can be used when the  $QR$  decomposition of a matrix is available in unpacked form as  $(Q, R)$ .

int **gsl\_linalg\_QR\_update**(gsl\_matrix \*Q, gsl\_matrix \*R, gsl\_vector \*w, const gsl\_vector \*v)

This function performs a rank-1 update  $wv^T$  of the  $QR$  decomposition  $(Q, R)$ . The update is given by  $Q'R' = Q(R + wv^T)$  where the output matrices  $Q$  and  $R$  are also orthogonal and right triangular. Note that  $w$  is destroyed by the update.

int **gsl\_linalg\_R\_solve**(const gsl\_matrix \*R, const gsl\_vector \*b, gsl\_vector \*x)

This function solves the triangular system  $Rx = b$  for the  $N$ -by- $N$  matrix  $R$ .

int **gsl\_linalg\_R\_svx**(const gsl\_matrix \*R, gsl\_vector \*x)

This function solves the triangular system  $Rx = b$  in-place. On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

## 14.2.2 Triangle on Top of Rectangle

This section provides routines for computing the  $QR$  decomposition of the specialized matrix

$$\begin{pmatrix} U \\ A \end{pmatrix} = QR$$

where  $U$  is an  $N$ -by- $N$  upper triangular matrix, and  $A$  is an  $M$ -by- $N$  dense matrix. This type of matrix arises, for example, in the sequential TSQR algorithm. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. Due to the upper triangular factor, the  $Q$  matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and  $Y$  is dense and of the same dimensions as  $A$ .



int **gsl\_linalg\_QR\_UR\_decomp**(*gsl\_matrix* \*U, *gsl\_matrix* \*A, *gsl\_matrix* \*T)

This function computes the  $QR$  decomposition of the matrix  $(U; A)$ , where  $U$  is  $N$ -by- $N$  upper triangular and  $A$  is  $M$ -by- $N$  dense. On output,  $U$  is replaced by the  $R$  factor, and  $A$  is replaced by  $Y$ . The  $N$ -by- $N$  upper triangular block reflector is stored in  $T$  on output.

### 14.2.3 Triangle on Top of Triangle

This section provides routines for computing the  $QR$  decomposition of the specialized matrix

$$\begin{pmatrix} U_1 \\ U_2 \end{pmatrix} = QR$$

where  $U_1, U_2$  are  $N$ -by- $N$  upper triangular matrices. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. The  $Q$  matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and  $Y$  is  $N$ -by- $N$  upper triangular.

int **gsl\_linalg\_QR\_UU\_decomp**(*gsl\_matrix* \*U1, *gsl\_matrix* \*U2, *gsl\_matrix* \*T)

This function computes the  $QR$  decomposition of the matrix  $(U_1; U_2)$ , where  $U_1, U_2$  are  $N$ -by- $N$  upper triangular. On output,  $U1$  is replaced by the  $R$  factor, and  $U2$  is replaced by  $Y$ . The  $N$ -by- $N$  upper triangular block reflector is stored in  $T$  on output.

int **gsl\_linalg\_QR\_UU\_lsolve**(const *gsl\_matrix* \*R, const *gsl\_matrix* \*Y, const *gsl\_matrix* \*T, const *gsl\_vector* \*b, *gsl\_vector* \*x, *gsl\_vector* \*work)

This function find the least squares solution to the overdetermined system,

$$\min_x \left\| b - \begin{pmatrix} U_1 \\ U_2 \end{pmatrix} x \right\|^2$$

where  $U_1, U_2$  are  $N$ -by- $N$  upper triangular matrices. The routine requires as input the  $QR$  decomposition of  $(U_1; U_2)$  into  $(R, Y, T)$  given by [gsl\\_linalg\\_QR\\_UU\\_decomp\(\)](#). The parameter  $x$  is of length  $2N$ . The solution  $x$  is returned in the first  $N$  rows of  $x$ , i.e.  $x = x[0], x[1], \dots, x[N-1]$ . The last  $N$  rows of  $x$  contain a vector whose norm is equal to the residual norm  $\|b - (U_1; U_2)x\|$ . This similar to the behavior of LAPACK DGELS. Additional workspace of length  $N$  is required in *work*.

int **gsl\_linalg\_QR\_UU\_QTec**(const *gsl\_matrix* \*Y, const *gsl\_matrix* \*T, *gsl\_vector* \*b, *gsl\_vector* \*work)

This function computes  $Q^T b$  using the decomposition  $(Y, T)$  previously computed by [gsl\\_linalg\\_QR\\_UU\\_decomp\(\)](#). On input,  $b$  contains the vector  $b$ , and on output it will contain  $Q^T b$ . Additional workspace of length  $N$  is required in *work*.

### 14.2.4 Triangle on Top of Trapezoidal

This section provides routines for computing the  $QR$  decomposition of the specialized matrix

$$\begin{pmatrix} U \\ A \end{pmatrix} = QR$$

where  $U$  is an  $N$ -by- $N$  upper triangular matrix, and  $A$  is an  $M$ -by- $N$  upper trapezoidal matrix with  $M \geq N$ .  $A$  has the structure,

$$A = \begin{pmatrix} A_d \\ A_u \end{pmatrix}$$

where  $A_d$  is  $(M - N)$ -by- $N$  dense, and  $A_u$  is  $N$ -by- $N$  upper triangular. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. The  $Q$  matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and  $Y$  is upper trapezoidal and of the same dimensions as  $A$ .

int **gsl\_linalg\_QR\_UZ\_decomp**(*gsl\_matrix* \*U, *gsl\_matrix* \*A, *gsl\_matrix* \*T)

This function computes the  $QR$  decomposition of the matrix  $(U; A)$ , where  $U$  is  $N$ -by- $N$  upper triangular and  $A$  is  $M$ -by- $N$  upper trapezoidal. On output,  $U$  is replaced by the  $R$  factor, and  $A$  is replaced by  $Y$ . The  $N$ -by- $N$  upper triangular block reflector is stored in  $T$  on output.

### 14.2.5 Triangle on Top of Diagonal

This section provides routines for computing the  $QR$  decomposition of the specialized matrix

$$\begin{pmatrix} U \\ D \end{pmatrix} = QR$$

where  $U$  is an  $N$ -by- $N$  upper triangular matrix and  $D$  is an  $N$ -by- $N$  diagonal matrix. This type of matrix arises in regularized least squares problems. The Elmroth and Gustavson algorithm is used to efficiently factor this matrix. The  $Q$  matrix takes the form

$$Q = I - VTV^T$$

with

$$V = \begin{pmatrix} I \\ Y \end{pmatrix}$$

and  $Y$  is  $N$ -by- $N$  upper triangular.

int **gsl\_linalg\_QR\_UD\_decomp**(*gsl\_matrix* \*U, const *gsl\_vector* \*D, *gsl\_matrix* \*Y, *gsl\_matrix* \*T)

This function computes the  $QR$  decomposition of the matrix  $(U; D)$ , where  $U$  is  $N$ -by- $N$  upper triangular and  $D$  is  $N$ -by- $N$  diagonal. On output,  $U$  is replaced by the  $R$  factor and  $Y$  is stored in  $Y$ . The  $N$ -by- $N$  upper triangular block reflector is stored in  $T$  on output.

int **gsl\_linalg\_QR\_UD\_lssolve**(const *gsl\_matrix* \*R, const *gsl\_matrix* \*Y, const *gsl\_matrix* \*T, const *gsl\_vector* \*b, *gsl\_vector* \*x, *gsl\_vector* \*work)

This function find the least squares solution to the overdetermined system,

$$\min_x \left\| b - \begin{pmatrix} U \\ D \end{pmatrix} x \right\|^2$$

where  $U$  is  $N$ -by- $N$  upper triangular and  $D$  is  $N$ -by- $N$  diagonal. The routine requires as input the  $QR$  decomposition of  $(U; D)$  into  $(R, Y, T)$  given by `gsl_linalg_QR_UD_decomp()`. The parameter  $x$  is of length  $2N$ . The solution  $x$  is returned in the first  $N$  rows of  $x$ , i.e.  $x = x[0], x[1], \dots, x[N-1]$ . The last  $N$  rows of  $x$  contain a vector whose norm is equal to the residual norm  $\|b - (U; D)x\|$ . This similar to the behavior of LAPACK DGELS. Additional workspace of length  $N$  is required in *work*.

## 14.3 QR Decomposition with Column Pivoting

The  $QR$  decomposition of an  $M$ -by- $N$  matrix  $A$  can be extended to the rank deficient case by introducing a column permutation  $P$ ,

$$AP = QR$$

The first  $r$  columns of  $Q$  form an orthonormal basis for the range of  $A$  for a matrix with column rank  $r$ . This decomposition can also be used to convert the square linear system  $Ax = b$  into the triangular system  $Ry = Q^T b, x = Py$ , which can be solved by back-substitution and permutation. We denote the  $QR$  decomposition with column pivoting by  $QRPT$  since  $A = QRPT$ . When  $A$  is rank deficient with  $r = \text{rank}(A)$ , the matrix  $R$  can be partitioned as

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} \approx \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

where  $R_{11}$  is  $r$ -by- $r$  and nonsingular. In this case, a *basic* least squares solution for the overdetermined system  $Ax = b$  can be obtained as

$$x = P \begin{pmatrix} R_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

where  $c_1$  consists of the first  $r$  elements of  $Q^T b$ . This basic solution is not guaranteed to be the minimum norm solution unless  $R_{12} = 0$  (see *Complete Orthogonal Decomposition*).

int **gsl\_linalg\_QRPT\_decomp**(*gsl\_matrix* \*A, *gsl\_vector* \*tau, *gsl\_permutation* \*p, int \*signum, *gsl\_vector* \*norm)

This function factorizes the  $M$ -by- $N$  matrix  $A$  into the  $QRPT$  decomposition  $A = QRPT$ . On output the diagonal and upper triangular part of the input matrix contain the matrix  $R$ . The permutation matrix  $P$  is stored in the permutation  $p$ . The sign of the permutation is given by *signum*. It has the value  $(-1)^n$ , where  $n$  is the number of interchanges in the permutation. The vector *tau* and the columns of the lower triangular part of the matrix  $A$  contain the Householder coefficients and vectors which encode the orthogonal matrix  $Q$ . The vector *tau* must be of length  $k = \min(M, N)$ . The matrix  $Q$  is related to these components by,  $Q = Q_k \dots Q_2 Q_1$  where  $Q_i = I - \tau_i v_i v_i^T$  and  $v_i$  is the Householder vector

$$v_i = (0, \dots, 1, A(i+1, i), A(i+2, i), \dots, A(m, i))$$

This is the same storage scheme as used by LAPACK. The vector *norm* is a workspace of length  $N$  used for column pivoting.

The algorithm used to perform the decomposition is Householder QR with column pivoting (Golub & Van Loan, “Matrix Computations”, Algorithm 5.4.1).

int **gsl\_linalg\_QRPT\_decomp2**(const *gsl\_matrix* \*A, *gsl\_matrix* \*q, *gsl\_matrix* \*r, *gsl\_vector* \*tau, *gsl\_permutation* \*p, int \*signum, *gsl\_vector* \*norm)

This function factorizes the matrix  $A$  into the decomposition  $A = QRPT$  without modifying  $A$  itself and storing the output in the separate matrices  $q$  and  $r$ .

int **gsl\_linalg\_QRPT\_solve**(const *gsl\_matrix* \*QR, const *gsl\_vector* \*tau, const *gsl\_permutation* \*p, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the square system  $Ax = b$  using the  $QRPT$  decomposition of  $A$  held in  $(QR, \text{tau}, p)$  which must have been computed previously by `gsl_linalg_QRPT_decomp()`.

int **gsl\_linalg\_QRPT\_svx**(const *gsl\_matrix* \*QR, const *gsl\_vector* \*tau, const *gsl\_permutation* \*p, *gsl\_vector* \*x)

This function solves the square system  $Ax = b$  in-place using the  $QRPT$  decomposition of  $A$  held in  $(QR, \text{tau}, p)$ . On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

```
int gsl_linalg_QRPT_lssolve(const gsl_matrix *QR, const gsl_vector *tau, const gsl_permutation *p, const
                           gsl_vector *b, gsl_vector *x, gsl_vector *residual)
```

This function finds the least squares solution to the overdetermined system  $Ax = b$  where the matrix  $A$  has more rows than columns and is assumed to have full rank. The least squares solution minimizes the Euclidean norm of the residual,  $\|b - Ax\|$ . The routine requires as input the  $QR$  decomposition of  $A$  into  $(QR, \tau, p)$  given by `gsl_linalg_QRPT_decomp()`. The solution is returned in  $x$ . The residual is computed as a by-product and stored in `residual`. For rank deficient matrices, `gsl_linalg_QRPT_lssolve2()` should be used instead.

```
int gsl_linalg_QRPT_lssolve2(const gsl_matrix *QR, const gsl_vector *tau, const gsl_permutation *p, const
                             gsl_vector *b, const size_t rank, gsl_vector *x, gsl_vector *residual)
```

This function finds the least squares solution to the overdetermined system  $Ax = b$  where the matrix  $A$  has more rows than columns and has rank given by the input `rank`. If the user does not know the rank of  $A$ , the routine `gsl_linalg_QRPT_rank()` can be called to estimate it. The least squares solution is the so-called “basic” solution discussed above and may not be the minimum norm solution. The routine requires as input the  $QR$  decomposition of  $A$  into  $(QR, \tau, p)$  given by `gsl_linalg_QRPT_decomp()`. The solution is returned in  $x$ . The residual is computed as a by-product and stored in `residual`.

```
int gsl_linalg_QRPT_QRsolve(const gsl_matrix *Q, const gsl_matrix *R, const gsl_permutation *p, const
                             gsl_vector *b, gsl_vector *x)
```

This function solves the square system  $RP^T x = Q^T b$  for  $x$ . It can be used when the  $QR$  decomposition of a matrix is available in unpacked form as  $(Q, R)$ .

```
int gsl_linalg_QRPT_update(gsl_matrix *Q, gsl_matrix *R, const gsl_permutation *p, gsl_vector *w, const
                           gsl_vector *v)
```

This function performs a rank-1 update  $wv^T$  of the  $QRPT$  decomposition  $(Q, R, p)$ . The update is given by  $Q'R' = Q(R + wv^T P)$  where the output matrices  $Q'$  and  $R'$  are also orthogonal and right triangular. Note that  $w$  is destroyed by the update. The permutation  $p$  is not changed.

```
int gsl_linalg_QRPT_Rsolve(const gsl_matrix *QR, const gsl_permutation *p, const gsl_vector *b, gsl_vector
                           *x)
```

This function solves the triangular system  $RP^T x = b$  for the  $N$ -by- $N$  matrix  $R$  contained in `QR`.

```
int gsl_linalg_QRPT_Rsvx(const gsl_matrix *QR, const gsl_permutation *p, gsl_vector *x)
```

This function solves the triangular system  $RP^T x = b$  in-place for the  $N$ -by- $N$  matrix  $R$  contained in `QR`. On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

```
size_t gsl_linalg_QRPT_rank(const gsl_matrix *QR, const double tol)
```

This function estimates the rank of the triangular matrix  $R$  contained in `QR`. The algorithm simply counts the number of diagonal elements of  $R$  whose absolute value is greater than the specified tolerance `tol`. If the input `tol` is negative, a default value of  $20(M + N)eps(\max(|diag(R)|))$  is used.

```
int gsl_linalg_QRPT_rcond(const gsl_matrix *QR, double *rcond, gsl_vector *work)
```

This function estimates the reciprocal condition number (using the 1-norm) of the  $R$  factor, stored in the upper triangle of `QR`. The reciprocal condition number estimate, defined as  $1/(\|R\|_1 \cdot \|R^{-1}\|_1)$ , is stored in `rcond`. Additional workspace of size  $3N$  is required in `work`.

## 14.4 LQ Decomposition

A general rectangular  $M$ -by- $N$  matrix  $A$  has a  $LQ$  decomposition into the product of a lower trapezoidal  $M$ -by- $N$  matrix  $L$  and an orthogonal  $N$ -by- $N$  square matrix  $Q$ :

$$A = LQ$$

If  $M \leq N$ , then  $L$  can be written as  $L = (L_1 \ 0)$  where  $L_1$  is  $M$ -by- $M$  lower triangular, and

$$A = (L_1 \ 0) \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = L_1 Q_1$$

where  $Q_1$  consists of the first  $M$  rows of  $Q$ , and  $Q_2$  contains the remaining  $N - M$  rows. The  $LQ$  factorization of  $A$  is essentially the same as the *QR factorization* of  $A^T$ .

The  $LQ$  factorization may be used to find the minimum norm solution of an underdetermined system of equations  $Ax = b$ , where  $A$  is  $M$ -by- $N$  and  $M \leq N$ . The solution is

$$x = Q^T \begin{pmatrix} L_1^{-1}b \\ 0 \end{pmatrix}$$

int **gsl\_linalg\_LQ\_decomp**(*gsl\_matrix* \*A, *gsl\_vector* \*tau)

This function factorizes the  $M$ -by- $N$  matrix  $A$  into the  $LQ$  decomposition  $A = LQ$ . On output the diagonal and lower trapezoidal part of the input matrix contain the matrix  $L$ . The vector  $\tau$  and the elements above the diagonal of the matrix  $A$  contain the Householder coefficients and Householder vectors which encode the orthogonal matrix  $Q$ . The vector  $\tau$  must be of length  $k = \min(M, N)$ . The matrix  $Q$  is related to these components by,  $Q = Q_k \dots Q_2 Q_1$  where  $Q_i = I - \tau_i v_i v_i^T$  and  $v_i$  is the Householder vector  $v_i = (0, \dots, 1, A(i, i+1), A(i, i+2), \dots, A(i, N))$ . This is the same storage scheme as used by LAPACK.

int **gsl\_linalg\_LQ\_lassolve**(const *gsl\_matrix* \*LQ, const *gsl\_vector* \*tau, const *gsl\_vector* \*b, *gsl\_vector* \*x, *gsl\_vector* \*residual)

This function finds the minimum norm least squares solution to the underdetermined system  $Ax = b$ , where the  $M$ -by- $N$  matrix  $A$  has  $M \leq N$ . The routine requires as input the  $LQ$  decomposition of  $A$  into  $(LQ, \tau)$  given by `gsl_linalg_LQ_decomp()`. The solution is returned in  $x$ . The residual,  $b - Ax$ , is computed as a by-product and stored in *residual*.

int **gsl\_linalg\_LQ\_unpack**(const *gsl\_matrix* \*LQ, const *gsl\_vector* \*tau, *gsl\_matrix* \*Q, *gsl\_matrix* \*L)

This function unpacks the encoded  $LQ$  decomposition  $(LQ, \tau)$  into the matrices  $Q$  and  $L$ , where  $Q$  is  $N$ -by- $N$  and  $L$  is  $M$ -by- $N$ .

int **gsl\_linalg\_LQ\_QTvec**(const *gsl\_matrix* \*LQ, const *gsl\_vector* \*tau, *gsl\_vector* \*v)

This function applies  $Q^T$  to the vector  $v$ , storing the result  $Q^T v$  in  $v$  on output.

## 14.5 QL Decomposition

A general rectangular  $M$ -by- $N$  matrix  $A$  has a  $QL$  decomposition into the product of an orthogonal  $M$ -by- $M$  square matrix  $Q$  (where  $Q^T Q = I$ ) and an  $M$ -by- $N$  left-triangular matrix  $L$ .

When  $M \geq N$ , the decomposition is given by

$$A = Q \begin{pmatrix} 0 \\ L_1 \end{pmatrix}$$

where  $L_1$  is  $N$ -by- $N$  lower triangular. When  $M \leq N$ , the decomposition is given by

$$A = Q \begin{pmatrix} L_1 & L_2 \end{pmatrix}$$

where  $L_1$  is a dense  $M$ -by- $N - M$  matrix and  $L_2$  is a lower triangular  $M$ -by- $M$  matrix.

int **gsl\_linalg\_QL\_decomp**(*gsl\_matrix* \*A, *gsl\_vector* \*tau)

This function factorizes the  $M$ -by- $N$  matrix  $A$  into the  $QL$  decomposition  $A = QL$ . The vector  $\tau$  must be of length  $N$  and contains the Householder coefficients on output. The matrix  $Q$  is stored in packed form in  $A$  on output, using the same storage scheme as LAPACK.

int **gsl\_linalg\_QL\_unpack**(const *gsl\_matrix* \*QL, const *gsl\_vector* \*tau, *gsl\_matrix* \*Q, *gsl\_matrix* \*L)

This function unpacks the encoded  $QL$  decomposition  $(QL, \tau)$  into the matrices  $Q$  and  $L$ , where  $Q$  is  $M$ -by- $M$  and  $L$  is  $M$ -by- $N$ .

## 14.6 Complete Orthogonal Decomposition

The complete orthogonal decomposition of a  $M$ -by- $N$  matrix  $A$  is a generalization of the QR decomposition with column pivoting, given by

$$AP = Q \begin{pmatrix} R_{11} & 0 \\ 0 & 0 \end{pmatrix} Z^T$$

where  $P$  is a  $N$ -by- $N$  permutation matrix,  $Q$  is  $M$ -by- $M$  orthogonal,  $R_{11}$  is  $r$ -by- $r$  upper triangular, with  $r = \text{rank}(A)$ , and  $Z$  is  $N$ -by- $N$  orthogonal. If  $A$  has full rank, then  $R_{11} = R$ ,  $Z = I$  and this reduces to the QR decomposition with column pivoting.

For a rank deficient least squares problem,  $\min_x \|b - Ax\|^2$ , the solution vector  $x$  is not unique. However if we further require that  $\|x\|^2$  is minimized, then the complete orthogonal decomposition gives us the ability to compute the unique minimum norm solution, which is given by

$$x = PZ \begin{pmatrix} R_{11}^{-1} c_1 \\ 0 \end{pmatrix}$$

and the vector  $c_1$  is the first  $r$  elements of  $Q^T b$ .

The COD also enables a straightforward solution of regularized least squares problems in Tikhonov standard form, written as

$$\min_x \|b - Ax\|^2 + \lambda^2 \|x\|^2$$

where  $\lambda > 0$  is a regularization parameter which represents a tradeoff between minimizing the residual norm  $\|b - Ax\|$  and the solution norm  $\|x\|$ . For this system, the solution is given by

$$x = PZ \begin{pmatrix} y_1 \\ 0 \end{pmatrix}$$

where  $y_1$  is a vector of length  $r$  which is found by solving

$$\begin{pmatrix} R_{11} \\ \lambda I_r \end{pmatrix} y_1 = \begin{pmatrix} c_1 \\ 0 \end{pmatrix}$$

and  $c_1$  is defined above. The equation above can be solved efficiently for different values of  $\lambda$  using QR factorizations of the left hand side matrix.

```
int gsl_linalg_COD_decomp(gsl_matrix *A, gsl_vector *tau_Q, gsl_vector *tau_Z, gsl_permutation *p, size_t
                        *rank, gsl_vector *work)
```

```
int gsl_linalg_COD_decomp_e(gsl_matrix *A, gsl_vector *tau_Q, gsl_vector *tau_Z, gsl_permutation *p, double
                           tol, size_t *rank, gsl_vector *work)
```

These functions factor the  $M$ -by- $N$  matrix  $A$  into the decomposition  $A = QRZP^T$ . The rank of  $A$  is computed as the number of diagonal elements of  $R$  greater than the tolerance `tol` and output in `rank`. If `tol` is not specified, a default value is used (see `gsl_linalg_QRPT_rank()`). On output, the permutation matrix  $P$  is stored in `p`. The matrix  $R_{11}$  is stored in the upper `rank`-by-`rank` block of `A`. The matrices  $Q$  and  $Z$  are encoded in packed storage in `A` on output. The vectors `tau_Q` and `tau_Z` contain the Householder scalars corresponding to the matrices  $Q$  and  $Z$  respectively and must be of length  $k = \min(M, N)$ . The vector `work` is additional workspace of length  $N$ .

```
int gsl_linalg_COD_lassolve(const gsl_matrix *QRZT, const gsl_vector *tau_Q, const gsl_vector *tau_Z, const
                           gsl_permutation *p, const size_t rank, const gsl_vector *b, gsl_vector *x,
                           gsl_vector *residual)
```

This function finds the unique minimum norm least squares solution to the overdetermined system  $Ax = b$  where the matrix  $A$  has more rows than columns. The least squares solution minimizes the Euclidean norm

of the residual,  $\|b - Ax\|$  as well as the norm of the solution  $\|x\|$ . The routine requires as input the *QRZT* decomposition of  $A$  into (*QRZT*, *tau\_Q*, *tau\_Z*, *p*, *rank*) given by *gsl\_linalg\_COD\_decomp()*. The solution is returned in *x*. The residual,  $b - Ax$ , is computed as a by-product and stored in *residual*.

```
int gsl_linalg_COD_lssolve2(const double lambda, const gsl_matrix *QRZT, const gsl_vector *tau_Q, const
    gsl_vector *tau_Z, const gsl_permutation *p, const size_t rank, const gsl_vector
    *b, gsl_vector *x, gsl_vector *residual, gsl_matrix *S, gsl_vector *work)
```

This function finds the solution to the regularized least squares problem in Tikhonov standard form,  $\min_x \|b - Ax\|^2 + \lambda^2 \|x\|^2$ . The routine requires as input the *QRZT* decomposition of  $A$  into (*QRZT*, *tau\_Q*, *tau\_Z*, *p*, *rank*) given by *gsl\_linalg\_COD\_decomp()*. The parameter  $\lambda$  is supplied in *lambda*. The solution is returned in *x*. The residual,  $b - Ax$ , is stored in *residual* on output. *S* is additional workspace of size *rank*-by-*rank*. *work* is additional workspace of length *rank*.

```
int gsl_linalg_COD_unpack(const gsl_matrix *QRZT, const gsl_vector *tau_Q, const gsl_vector *tau_Z, const
    size_t rank, gsl_matrix *Q, gsl_matrix *R, gsl_matrix *Z)
```

This function unpacks the encoded *QRZT* decomposition (*QRZT*, *tau\_Q*, *tau\_Z*, *rank*) into the matrices *Q*, *R*, and *Z*, where *Q* is  $M$ -by- $M$ , *R* is  $M$ -by- $N$ , and *Z* is  $N$ -by- $N$ .

```
int gsl_linalg_COD_matZ(const gsl_matrix *QRZT, const gsl_vector *tau_Z, const size_t rank, gsl_matrix *A,
    gsl_vector *work)
```

This function multiplies the input matrix  $A$  on the right by  $Z$ ,  $A' = AZ$  using the encoded *QRZT* decomposition (*QRZT*, *tau\_Z*, *rank*).  $A$  must have  $N$  columns but may have any number of rows. Additional workspace of length  $M$  is provided in *work*.

## 14.7 Singular Value Decomposition

A general rectangular  $M$ -by- $N$  matrix  $A$  has a singular value decomposition (SVD) into the product of an  $M$ -by- $N$  orthogonal matrix  $U$ , an  $N$ -by- $N$  diagonal matrix of singular values  $S$  and the transpose of an  $N$ -by- $N$  orthogonal square matrix  $V$ ,

$$A = USV^T$$

The singular values  $\sigma_i = S_{ii}$  are all non-negative and are generally chosen to form a non-increasing sequence

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_N \geq 0$$

The singular value decomposition of a matrix has many practical uses. The condition number of the matrix is given by the ratio of the largest singular value to the smallest singular value. The presence of a zero singular value indicates that the matrix is singular. The number of non-zero singular values indicates the rank of the matrix. In practice singular value decomposition of a rank-deficient matrix will not produce exact zeroes for singular values, due to finite numerical precision. Small singular values should be edited by choosing a suitable tolerance.

For a rank-deficient matrix, the null space of  $A$  is given by the columns of  $V$  corresponding to the zero singular values. Similarly, the range of  $A$  is given by columns of  $U$  corresponding to the non-zero singular values.

Note that the routines here compute the “thin” version of the SVD with  $U$  as  $M$ -by- $N$  orthogonal matrix. This allows in-place computation and is the most commonly-used form in practice. Mathematically, the “full” SVD is defined with  $U$  as an  $M$ -by- $M$  orthogonal matrix and  $S$  as an  $M$ -by- $N$  diagonal matrix (with additional rows of zeros).

```
int gsl_linalg_SV_decomp(gsl_matrix *A, gsl_matrix *V, gsl_vector *S, gsl_vector *work)
```

This function factorizes the  $M$ -by- $N$  matrix  $A$  into the singular value decomposition  $A = USV^T$  for  $M \geq N$ . On output the matrix  $A$  is replaced by  $U$ . The diagonal elements of the singular value matrix  $S$  are stored in the vector  $S$ . The singular values are non-negative and form a non-increasing sequence from  $S_1$  to  $S_N$ . The matrix  $V$  contains the elements of  $V$  in untransposed form. To form the product  $USV^T$  it is necessary to take the transpose of  $V$ . A workspace of length  $N$  is required in *work*.

This routine uses the Golub-Reinsch SVD algorithm.



int **gsl\_linalg\_SV\_decomp\_mod**(*gsl\_matrix* \*A, *gsl\_matrix* \*X, *gsl\_matrix* \*V, *gsl\_vector* \*S, *gsl\_vector* \*work)  
 This function computes the SVD using the modified Golub-Reinsch algorithm, which is faster for  $M \gg N$ . It requires the vector *work* of length  $N$  and the  $N$ -by- $N$  matrix *X* as additional working space.

int **gsl\_linalg\_SV\_decomp\_jacobi**(*gsl\_matrix* \*A, *gsl\_matrix* \*V, *gsl\_vector* \*S)  
 This function computes the SVD of the  $M$ -by- $N$  matrix *A* using one-sided Jacobi orthogonalization for  $M \geq N$ . The Jacobi method can compute singular values to higher relative accuracy than Golub-Reinsch algorithms (see references for details).

int **gsl\_linalg\_SV\_solve**(const *gsl\_matrix* \*U, const *gsl\_matrix* \*V, const *gsl\_vector* \*S, const *gsl\_vector* \*b, *gsl\_vector* \*x)  
 This function solves the system  $Ax = b$  using the singular value decomposition ( $U, S, V$ ) of *A* which must have been computed previously with `gsl_linalg_SV_decomp()`.

Only non-zero singular values are used in computing the solution. The parts of the solution corresponding to singular values of zero are ignored. Other singular values can be edited out by setting them to zero before calling this function.

In the over-determined case where *A* has more rows than columns the system is solved in the least squares sense, returning the solution  $x$  which minimizes  $\|Ax - b\|_2$ .

int **gsl\_linalg\_SV\_leverage**(const *gsl\_matrix* \*U, *gsl\_vector* \*h)  
 This function computes the statistical leverage values  $h_i$  of a matrix *A* using its singular value decomposition ( $U, S, V$ ) previously computed with `gsl_linalg_SV_decomp()`.  $h_i$  are the diagonal values of the matrix  $A(A^T A)^{-1} A^T$  and depend only on the matrix *U* which is the input to this function.

## 14.8 Cholesky Decomposition

A symmetric, positive definite square matrix *A* has a Cholesky decomposition into a product of a lower triangular matrix *L* and its transpose  $L^T$ ,

$$A = LL^T$$

This is sometimes referred to as taking the square-root of a matrix. The Cholesky decomposition can only be carried out when all the eigenvalues of the matrix are positive. This decomposition can be used to convert the linear system  $Ax = b$  into a pair of triangular systems ( $Ly = b, L^T x = y$ ), which can be solved by forward and back-substitution.

If the matrix *A* is near singular, it is sometimes possible to reduce the condition number and recover a more accurate solution vector  $x$  by scaling as

$$(SAS) (S^{-1}x) = Sb$$

where *S* is a diagonal matrix whose elements are given by  $S_{ii} = 1/\sqrt{A_{ii}}$ . This scaling is also known as Jacobi preconditioning. There are routines below to solve both the scaled and unscaled systems.

int **gsl\_linalg\_cholesky\_decomp1**(*gsl\_matrix* \*A)

int **gsl\_linalg\_complex\_cholesky\_decomp**(*gsl\_matrix\_complex* \*A)

These functions factorize the symmetric, positive-definite square matrix *A* into the Cholesky decomposition  $A = LL^T$  (or  $A = LL^\dagger$  for the complex case). On input, the values from the diagonal and lower-triangular part of the matrix *A* are used (the upper triangular part is ignored). On output the diagonal and lower triangular part of the input matrix *A* contain the matrix *L*, while the upper triangular part contains the original matrix. If the matrix is not positive-definite then the decomposition will fail, returning the error code `GSL_EDOM`.

When testing whether a matrix is positive-definite, disable the error handler first to avoid triggering an error. These functions use Level 3 BLAS to compute the Cholesky factorization (Peise and Bientinesi, 2016).



int **gsl\_linalg\_cholesky\_decomp**(*gsl\_matrix* \*A)

This function is now deprecated and is provided only for backward compatibility.

int **gsl\_linalg\_cholesky\_solve**(const *gsl\_matrix* \*cholesky, const *gsl\_vector* \*b, *gsl\_vector* \*x)

int **gsl\_linalg\_complex\_cholesky\_solve**(const *gsl\_matrix\_complex* \*cholesky, const *gsl\_vector\_complex* \*b, *gsl\_vector\_complex* \*x)

These functions solve the system  $Ax = b$  using the Cholesky decomposition of  $A$  held in the matrix *cholesky* which must have been previously computed by *gsl\_linalg\_cholesky\_decomp()* or *gsl\_linalg\_complex\_cholesky\_decomp()*.

int **gsl\_linalg\_cholesky\_svx**(const *gsl\_matrix* \*cholesky, *gsl\_vector* \*x)

int **gsl\_linalg\_complex\_cholesky\_svx**(const *gsl\_matrix\_complex* \*cholesky, *gsl\_vector\_complex* \*x)

These functions solve the system  $Ax = b$  in-place using the Cholesky decomposition of  $A$  held in the matrix *cholesky* which must have been previously computed by *gsl\_linalg\_cholesky\_decomp()* or *gsl\_linalg\_complex\_cholesky\_decomp()*. On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

int **gsl\_linalg\_cholesky\_invert**(*gsl\_matrix* \*cholesky)

int **gsl\_linalg\_complex\_cholesky\_invert**(*gsl\_matrix\_complex* \*cholesky)

These functions compute the inverse of a matrix from its Cholesky decomposition *cholesky*, which must have been previously computed by *gsl\_linalg\_cholesky\_decomp()* or *gsl\_linalg\_complex\_cholesky\_decomp()*. On output, the inverse is stored in-place in *cholesky*.

int **gsl\_linalg\_cholesky\_decomp2**(*gsl\_matrix* \*A, *gsl\_vector* \*S)

This function calculates a diagonal scaling transformation  $S$  for the symmetric, positive-definite square matrix  $A$ , and then computes the Cholesky decomposition  $SAS = LL^T$ . On input, the values from the diagonal and lower-triangular part of the matrix  $A$  are used (the upper triangular part is ignored). On output the diagonal and lower triangular part of the input matrix  $A$  contain the matrix  $L$ , while the upper triangular part of the input matrix is overwritten with  $L^T$  (the diagonal terms being identical for both  $L$  and  $L^T$ ). If the matrix is not positive-definite then the decomposition will fail, returning the error code *GSL\_EDOM*. The diagonal scale factors are stored in  $S$  on output.

When testing whether a matrix is positive-definite, disable the error handler first to avoid triggering an error.

int **gsl\_linalg\_cholesky\_solve2**(const *gsl\_matrix* \*cholesky, const *gsl\_vector* \*S, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the system  $(SAS)(S^{-1}x) = Sb$  using the Cholesky decomposition of  $SAS$  held in the matrix *cholesky* which must have been previously computed by *gsl\_linalg\_cholesky\_decomp2()*.

int **gsl\_linalg\_cholesky\_svx2**(const *gsl\_matrix* \*cholesky, const *gsl\_vector* \*S, *gsl\_vector* \*x)

This function solves the system  $(SAS)(S^{-1}x) = Sb$  in-place using the Cholesky decomposition of  $SAS$  held in the matrix *cholesky* which must have been previously computed by *gsl\_linalg\_cholesky\_decomp2()*. On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

int **gsl\_linalg\_cholesky\_scale**(const *gsl\_matrix* \*A, *gsl\_vector* \*S)

This function calculates a diagonal scaling transformation of the symmetric, positive definite matrix  $A$ , such that  $SAS$  has a condition number within a factor of  $N$  of the matrix of smallest possible condition number over all possible diagonal scalings. On output,  $S$  contains the scale factors, given by  $S_i = 1/\sqrt{A_{ii}}$ . For any  $A_{ii} \leq 0$ , the corresponding scale factor  $S_i$  is set to 1.

int **gsl\_linalg\_cholesky\_scale\_apply**(*gsl\_matrix* \*A, const *gsl\_vector* \*S)

This function applies the scaling transformation  $S$  to the matrix  $A$ . On output,  $A$  is replaced by  $SAS$ .

int **gsl\_linalg\_cholesky\_rcond**(const *gsl\_matrix* \*cholesky, double \*rcond, *gsl\_vector* \*work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric positive definite

matrix  $A$ , using its Cholesky decomposition provided in *cholesky*. The reciprocal condition number estimate, defined as  $1/(||A||_1 \cdot ||A^{-1}||_1)$ , is stored in *rcond*. Additional workspace of size  $3N$  is required in *work*.

## 14.9 Pivoted Cholesky Decomposition

A symmetric positive semi-definite square matrix  $A$  has an alternate Cholesky decomposition into a product of a lower unit triangular matrix  $L$ , a diagonal matrix  $D$  and  $L^T$ , given by  $LDL^T$ . For positive definite matrices, this is equivalent to the Cholesky formulation discussed above, with the standard Cholesky lower triangular factor given by  $LD^{\frac{1}{2}}$ . For ill-conditioned matrices, it can help to use a pivoting strategy to prevent the entries of  $D$  and  $L$  from growing too large, and also ensure  $D_1 \geq D_2 \geq \dots \geq D_n > 0$ , where  $D_i$  are the diagonal entries of  $D$ . The final decomposition is given by

$$PAP^T = LDL^T$$

where  $P$  is a permutation matrix.

int **gsl\_linalg\_pcholesky\_decomp**(*gsl\_matrix* \*A, *gsl\_permutation* \*p)

This function factors the symmetric, positive-definite square matrix  $A$  into the Pivoted Cholesky decomposition  $PAP^T = LDL^T$ . On input, the values from the diagonal and lower-triangular part of the matrix  $A$  are used to construct the factorization. On output the diagonal of the input matrix  $A$  stores the diagonal elements of  $D$ , and the lower triangular portion of  $A$  contains the matrix  $L$ . Since  $L$  has ones on its diagonal these do not need to be explicitly stored. The upper triangular portion of  $A$  is unmodified. The permutation matrix  $P$  is stored in  $p$  on output.

int **gsl\_linalg\_pcholesky\_solve**(const *gsl\_matrix* \*LDLT, const *gsl\_permutation* \*p, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the system  $Ax = b$  using the Pivoted Cholesky decomposition of  $A$  held in the matrix  $LDLT$  and permutation  $p$  which must have been previously computed by *gsl\_linalg\_pcholesky\_decomp()*.

int **gsl\_linalg\_pcholesky\_svx**(const *gsl\_matrix* \*LDLT, const *gsl\_permutation* \*p, *gsl\_vector* \*x)

This function solves the system  $Ax = b$  in-place using the Pivoted Cholesky decomposition of  $A$  held in the matrix  $LDLT$  and permutation  $p$  which must have been previously computed by *gsl\_linalg\_pcholesky\_decomp()*. On input,  $x$  contains the right hand side vector  $b$  which is replaced by the solution vector on output.

int **gsl\_linalg\_pcholesky\_decomp2**(*gsl\_matrix* \*A, *gsl\_permutation* \*p, *gsl\_vector* \*S)

This function computes the pivoted Cholesky factorization of the matrix  $SAS$ , where the input matrix  $A$  is symmetric and positive definite, and the diagonal scaling matrix  $S$  is computed to reduce the condition number of  $A$  as much as possible. See *Cholesky Decomposition* for more information on the matrix  $S$ . The Pivoted Cholesky decomposition satisfies  $PSASP^T = LDL^T$ . On input, the values from the diagonal and lower-triangular part of the matrix  $A$  are used to construct the factorization. On output the diagonal of the input matrix  $A$  stores the diagonal elements of  $D$ , and the lower triangular portion of  $A$  contains the matrix  $L$ . Since  $L$  has ones on its diagonal these do not need to be explicitly stored. The upper triangular portion of  $A$  is unmodified. The permutation matrix  $P$  is stored in  $p$  on output. The diagonal scaling transformation is stored in  $S$  on output.

int **gsl\_linalg\_pcholesky\_solve2**(const *gsl\_matrix* \*LDLT, const *gsl\_permutation* \*p, const *gsl\_vector* \*S, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the system  $(SAS)(S^{-1}x) = Sb$  using the Pivoted Cholesky decomposition of  $SAS$  held in the matrix  $LDLT$ , permutation  $p$ , and vector  $S$ , which must have been previously computed by *gsl\_linalg\_pcholesky\_decomp2()*.

int **gsl\_linalg\_pcholesky\_svx2**(const *gsl\_matrix* \*LDLT, const *gsl\_permutation* \*p, const *gsl\_vector* \*S, *gsl\_vector* \*x)

This function solves the system  $(SAS)(S^{-1}x) = Sb$  in-place using the Pivoted Cholesky decomposition of  $SAS$  held in the matrix  $LDLT$ , permutation  $p$  and vector  $S$ , which must have been previously computed by

`gsl_linalg_pcholesky_decomp2()`. On input,  $x$  contains the right hand side vector  $b$  which is replaced by the solution vector on output.

int `gsl_linalg_pcholesky_invert`(const `gsl_matrix` \*LDLT, const `gsl_permutation` \*p, `gsl_matrix` \*Ainv)  
This function computes the inverse of the matrix  $A$ , using the Pivoted Cholesky decomposition stored in `LDLT` and `p`. On output, the matrix `Ainv` contains  $A^{-1}$ .

int `gsl_linalg_pcholesky_rcond`(const `gsl_matrix` \*LDLT, const `gsl_permutation` \*p, double \*rcond, `gsl_vector` \*work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric positive definite matrix  $A$ , using its pivoted Cholesky decomposition provided in `LDLT`. The reciprocal condition number estimate, defined as  $1/(||A||_1 \cdot ||A^{-1}||_1)$ , is stored in `rcond`. Additional workspace of size  $3N$  is required in `work`.

## 14.10 Modified Cholesky Decomposition

The modified Cholesky decomposition is suitable for solving systems  $Ax = b$  where  $A$  is a symmetric indefinite matrix. Such matrices arise in nonlinear optimization algorithms. The standard Cholesky decomposition requires a positive definite matrix and would fail in this case. Instead of resorting to a method like QR or SVD, which do not take into account the symmetry of the matrix, we can instead introduce a small perturbation to the matrix  $A$  to make it positive definite, and then use a Cholesky decomposition on the perturbed matrix. The resulting decomposition satisfies

$$P(A + E)P^T = LDL^T$$

where  $P$  is a permutation matrix,  $E$  is a diagonal perturbation matrix,  $L$  is unit lower triangular, and  $D$  is diagonal. If  $A$  is sufficiently positive definite, then the perturbation matrix  $E$  will be zero and this method is equivalent to the pivoted Cholesky algorithm. For indefinite matrices, the perturbation matrix  $E$  is computed to ensure that  $A + E$  is positive definite and well conditioned.

int `gsl_linalg_mcholesky_decomp`(`gsl_matrix` \*A, `gsl_permutation` \*p, `gsl_vector` \*E)

This function factors the symmetric, indefinite square matrix  $A$  into the Modified Cholesky decomposition  $P(A + E)P^T = LDL^T$ . On input, the values from the diagonal and lower-triangular part of the matrix  $A$  are used to construct the factorization. On output the diagonal of the input matrix  $A$  stores the diagonal elements of  $D$ , and the lower triangular portion of  $A$  contains the matrix  $L$ . Since  $L$  has ones on its diagonal these do not need to be explicitly stored. The upper triangular portion of  $A$  is unmodified. The permutation matrix  $P$  is stored in `p` on output. The diagonal perturbation matrix is stored in `E` on output. The parameter `E` may be set to NULL if it is not required.

int `gsl_linalg_mcholesky_solve`(const `gsl_matrix` \*LDLT, const `gsl_permutation` \*p, const `gsl_vector` \*b, `gsl_vector` \*x)

This function solves the perturbed system  $(A + E)x = b$  using the Cholesky decomposition of  $A + E$  held in the matrix `LDLT` and permutation `p` which must have been previously computed by `gsl_linalg_mcholesky_decomp()`.

int `gsl_linalg_mcholesky_svx`(const `gsl_matrix` \*LDLT, const `gsl_permutation` \*p, `gsl_vector` \*x)

This function solves the perturbed system  $(A + E)x = b$  in-place using the Cholesky decomposition of  $A + E$  held in the matrix `LDLT` and permutation `p` which must have been previously computed by `gsl_linalg_mcholesky_decomp()`. On input,  $x$  contains the right hand side vector  $b$  which is replaced by the solution vector on output.

int `gsl_linalg_mcholesky_rcond`(const `gsl_matrix` \*LDLT, const `gsl_permutation` \*p, double \*rcond, `gsl_vector` \*work)

This function estimates the reciprocal condition number (using the 1-norm) of the perturbed matrix  $A + E$ , using its pivoted Cholesky decomposition provided in `LDLT`. The reciprocal condition number estimate, defined as  $1/(||A + E||_1 \cdot ||(A + E)^{-1}||_1)$ , is stored in `rcond`. Additional workspace of size  $3N$  is required in `work`.

## 14.11 LDLT Decomposition

If  $A$  is a symmetric, nonsingular square matrix, then it has a unique factorization of the form

$$A = LDL^T$$

where  $L$  is a unit lower triangular matrix and  $D$  is diagonal. If  $A$  is positive definite, then this factorization is equivalent to the Cholesky factorization, where the lower triangular Cholesky factor is  $LD^{\frac{1}{2}}$ . Some indefinite matrices for which no Cholesky decomposition exists have an  $LDL^T$  decomposition with negative entries in  $D$ . The  $LDL^T$  algorithm is sometimes referred to as the *square root free* Cholesky decomposition, as the algorithm does not require the computation of square roots. The algorithm is stable for positive definite matrices, but is not guaranteed to be stable for indefinite matrices.

int `gsl_linalg_ldlt_decomp`(*gsl\_matrix* \*A)

This function factorizes the symmetric, non-singular square matrix  $A$  into the decomposition  $A = LDL^T$ . On input, the values from the diagonal and lower-triangular part of the matrix  $A$  are used. The upper triangle of  $A$  is used as temporary workspace. On output the diagonal of  $A$  contains the matrix  $D$  and the lower triangle of  $A$  contains the unit lower triangular matrix  $L$ . The matrix 1-norm,  $\|A\|_1$  is stored in the upper right corner on output, for later use by `gsl_linalg_ldlt_rcond`.

If the matrix is detected to be singular, the function returns the error code `GSL_EDOM`.

int `gsl_linalg_ldlt_solve`(const *gsl\_matrix* \*LDLT, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the system  $Ax = b$  using the  $LDL^T$  decomposition of  $A$  held in the matrix `LDLT` which must have been previously computed by `gsl_linalg_ldlt_decomp`.

int `gsl_linalg_ldlt_svx`(const *gsl\_matrix* \*LDLT, *gsl\_vector* \*x)

This function solves the system  $Ax = b$  in-place using the  $LDL^T$  decomposition of  $A$  held in the matrix `LDLT` which must have been previously computed by `gsl_linalg_ldlt_decomp`. On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output.

int `gsl_linalg_ldlt_rcond`(const *gsl\_matrix* \*LDLT, double \*rcond, *gsl\_vector* \*work)

This function estimates the reciprocal condition number (using the 1-norm) of the symmetric nonsingular matrix  $A$ , using its  $LDL^T$  decomposition provided in `LDLT`. The reciprocal condition number estimate, defined as  $1/(\|A\|_1 \cdot \|A^{-1}\|_1)$ , is stored in `rcond`. Additional workspace of size  $3N$  is required in `work`.

## 14.12 Tridiagonal Decomposition of Real Symmetric Matrices

A symmetric matrix  $A$  can be factorized by similarity transformations into the form,

$$A = QTQ^T$$

where  $Q$  is an orthogonal matrix and  $T$  is a symmetric tridiagonal matrix.

int `gsl_linalg_symmtd_decomp`(*gsl\_matrix* \*A, *gsl\_vector* \*tau)

This function factorizes the symmetric square matrix  $A$  into the symmetric tridiagonal decomposition  $QTQ^T$ . On output the diagonal and subdiagonal part of the input matrix  $A$  contain the tridiagonal matrix  $T$ . The remaining lower triangular part of the input matrix contains the Householder vectors which, together with the Householder coefficients `tau`, encode the orthogonal matrix  $Q$ . This storage scheme is the same as used by LAPACK. The upper triangular part of  $A$  is not referenced.

int `gsl_linalg_symmtd_unpack`(const *gsl\_matrix* \*A, const *gsl\_vector* \*tau, *gsl\_matrix* \*Q, *gsl\_vector* \*diag, *gsl\_vector* \*subdiag)

This function unpacks the encoded symmetric tridiagonal decomposition ( $A$ , `tau`) obtained from `gsl_linalg_symmtd_decomp` into the orthogonal matrix  $Q$ , the vector of diagonal elements `diag` and the vector of subdiagonal elements `subdiag`.

```
int gsl_linalg_symmtd_unpack_T(const gsl_matrix *A, gsl_vector *diag, gsl_vector *subdiag)
```

This function unpacks the diagonal and subdiagonal of the encoded symmetric tridiagonal decomposition ( $A$ ,  $\tau$ ) obtained from `gsl_linalg_symmtd_decomp()` into the vectors *diag* and *subdiag*.

## 14.13 Tridiagonal Decomposition of Hermitian Matrices

A hermitian matrix  $A$  can be factorized by similarity transformations into the form,

$$A = UTU^T$$

where  $U$  is a unitary matrix and  $T$  is a real symmetric tridiagonal matrix.

```
int gsl_linalg_hermtdecomp(gsl_matrix_complex *A, gsl_vector_complex *tau)
```

This function factorizes the hermitian matrix  $A$  into the symmetric tridiagonal decomposition  $UTU^T$ . On output the real parts of the diagonal and subdiagonal part of the input matrix  $A$  contain the tridiagonal matrix  $T$ . The remaining lower triangular part of the input matrix contains the Householder vectors which, together with the Householder coefficients  $\tau$ , encode the unitary matrix  $U$ . This storage scheme is the same as used by LAPACK. The upper triangular part of  $A$  and imaginary parts of the diagonal are not referenced.

```
int gsl_linalg_hermtdeunpack(const gsl_matrix_complex *A, const gsl_vector_complex *tau,
                             gsl_matrix_complex *U, gsl_vector *diag, gsl_vector *subdiag)
```

This function unpacks the encoded tridiagonal decomposition ( $A$ ,  $\tau$ ) obtained from `gsl_linalg_hermtdecomp()` into the unitary matrix  $U$ , the real vector of diagonal elements *diag* and the real vector of subdiagonal elements *subdiag*.

```
int gsl_linalg_hermtdeunpack_T(const gsl_matrix_complex *A, gsl_vector *diag, gsl_vector *subdiag)
```

This function unpacks the diagonal and subdiagonal of the encoded tridiagonal decomposition ( $A$ ,  $\tau$ ) obtained from the `gsl_linalg_hermtdecomp()` into the real vectors *diag* and *subdiag*.

## 14.14 Hessenberg Decomposition of Real Matrices

A general real matrix  $A$  can be decomposed by orthogonal similarity transformations into the form

$$A = UHU^T$$

where  $U$  is orthogonal and  $H$  is an upper Hessenberg matrix, meaning that it has zeros below the first subdiagonal. The Hessenberg reduction is the first step in the Schur decomposition for the nonsymmetric eigenvalue problem, but has applications in other areas as well.

```
int gsl_linalg_hessenberg_decomp(gsl_matrix *A, gsl_vector *tau)
```

This function computes the Hessenberg decomposition of the matrix  $A$  by applying the similarity transformation  $H = U^T A U$ . On output,  $H$  is stored in the upper portion of  $A$ . The information required to construct the matrix  $U$  is stored in the lower triangular portion of  $A$ .  $U$  is a product of  $N - 2$  Householder matrices. The Householder vectors are stored in the lower portion of  $A$  (below the subdiagonal) and the Householder coefficients are stored in the vector  $\tau$ .  $\tau$  must be of length  $N$ .

```
int gsl_linalg_hessenberg_unpack(gsl_matrix *H, gsl_vector *tau, gsl_matrix *U)
```

This function constructs the orthogonal matrix  $U$  from the information stored in the Hessenberg matrix  $H$  along with the vector  $\tau$ .  $H$  and  $\tau$  are outputs from `gsl_linalg_hessenberg_decomp()`.

```
int gsl_linalg_hessenberg_unpack_accum(gsl_matrix *H, gsl_vector *tau, gsl_matrix *V)
```

This function is similar to `gsl_linalg_hessenberg_unpack()`, except it accumulates the matrix  $U$  into  $V$ , so that  $V' = VU$ . The matrix  $V$  must be initialized prior to calling this function. Setting  $V$  to the identity matrix provides the same result as `gsl_linalg_hessenberg_unpack()`. If  $H$  is order  $N$ , then  $V$  must have  $N$  columns but may have any number of rows.



int **gsl\_linalg\_hessenberg\_set\_zero**(*gsl\_matrix* \*H)

This function sets the lower triangular portion of  $H$ , below the subdiagonal, to zero. It is useful for clearing out the Householder vectors after calling `gsl_linalg_hessenberg_decomp()`.

## 14.15 Hessenberg-Triangular Decomposition of Real Matrices

A general real matrix pair  $(A, B)$  can be decomposed by orthogonal similarity transformations into the form

$$\begin{aligned}A &= UHV^T \\ B &= URV^T\end{aligned}$$

where  $U$  and  $V$  are orthogonal,  $H$  is an upper Hessenberg matrix, and  $R$  is upper triangular. The Hessenberg-Triangular reduction is the first step in the generalized Schur decomposition for the generalized eigenvalue problem.

int **gsl\_linalg\_hesstri\_decomp**(*gsl\_matrix* \*A, *gsl\_matrix* \*B, *gsl\_matrix* \*U, *gsl\_matrix* \*V, *gsl\_vector* \*work)

This function computes the Hessenberg-Triangular decomposition of the matrix pair  $(A, B)$ . On output,  $H$  is stored in  $A$ , and  $R$  is stored in  $B$ . If  $U$  and  $V$  are provided (they may be null), the similarity transformations are stored in them. Additional workspace of length  $N$  is needed in *work*.

## 14.16 Bidiagonalization

A general matrix  $A$  can be factorized by similarity transformations into the form,

$$A = UBV^T$$

where  $U$  and  $V$  are orthogonal matrices and  $B$  is a  $N$ -by- $N$  bidiagonal matrix with non-zero entries only on the diagonal and superdiagonal. The size of  $U$  is  $M$ -by- $N$  and the size of  $V$  is  $N$ -by- $N$ .

int **gsl\_linalg\_bidiag\_decomp**(*gsl\_matrix* \*A, *gsl\_vector* \*tau\_U, *gsl\_vector* \*tau\_V)

This function factorizes the  $M$ -by- $N$  matrix  $A$  into bidiagonal form  $UBV^T$ . The diagonal and superdiagonal of the matrix  $B$  are stored in the diagonal and superdiagonal of  $A$ . The orthogonal matrices  $U$  and  $V$  are stored as compressed Householder vectors in the remaining elements of  $A$ . The Householder coefficients are stored in the vectors *tau\_U* and *tau\_V*. The length of *tau\_U* must equal the number of elements in the diagonal of  $A$  and the length of *tau\_V* should be one element shorter.

int **gsl\_linalg\_bidiag\_unpack**(const *gsl\_matrix* \*A, const *gsl\_vector* \*tau\_U, *gsl\_matrix* \*U, const *gsl\_vector* \*tau\_V, *gsl\_matrix* \*V, *gsl\_vector* \*diag, *gsl\_vector* \*superdiag)

This function unpacks the bidiagonal decomposition of  $A$  produced by `gsl_linalg_bidiag_decomp()`,  $(A, \textit{tau}_U, \textit{tau}_V)$  into the separate orthogonal matrices  $U$ ,  $V$  and the diagonal vector *diag* and superdiagonal *superdiag*. Note that  $U$  is stored as a compact  $M$ -by- $N$  orthogonal matrix satisfying  $U^T U = I$  for efficiency.

int **gsl\_linalg\_bidiag\_unpack2**(*gsl\_matrix* \*A, *gsl\_vector* \*tau\_U, *gsl\_vector* \*tau\_V, *gsl\_matrix* \*V)

This function unpacks the bidiagonal decomposition of  $A$  produced by `gsl_linalg_bidiag_decomp()`,  $(A, \textit{tau}_U, \textit{tau}_V)$  into the separate orthogonal matrices  $U$ ,  $V$  and the diagonal vector *diag* and superdiagonal *superdiag*. The matrix  $U$  is stored in-place in  $A$ .

int **gsl\_linalg\_bidiag\_unpack\_B**(const *gsl\_matrix* \*A, *gsl\_vector* \*diag, *gsl\_vector* \*superdiag)

This function unpacks the diagonal and superdiagonal of the bidiagonal decomposition of  $A$  from `gsl_linalg_bidiag_decomp()`, into the diagonal vector *diag* and superdiagonal vector *superdiag*.

## 14.17 Givens Rotations

A Givens rotation is a rotation in the plane acting on two elements of a given vector. It can be represented in matrix form as

$$G(i, j, \theta) = \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & \cos \theta & \dots & -\sin \theta & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & \sin \theta & \dots & \cos \theta & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix}$$

where the  $\cos \theta$  and  $\sin \theta$  appear at the intersection of the  $i$ -th and  $j$ -th rows and columns. When acting on a vector  $x$ ,  $G(i, j, \theta)x$  performs a rotation of the  $(i, j)$  elements of  $x$ . Givens rotations are typically used to introduce zeros in vectors, such as during the QR decomposition of a matrix. In this case, it is typically desired to find  $c$  and  $s$  such that

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$$

with  $r = \sqrt{a^2 + b^2}$ .

void **gsl\_linalg\_givens**(const double a, const double b, double \*c, double \*s)

This function computes  $c = \cos \theta$  and  $s = \sin \theta$  so that the Givens matrix  $G(\theta)$  acting on the vector  $(a, b)$  produces  $(r, 0)$ , with  $r = \sqrt{a^2 + b^2}$ .

void **gsl\_linalg\_givens\_gv**(gsl\_vector \*v, const size\_t i, const size\_t j, const double c, const double s)

This function applies the Givens rotation defined by  $c = \cos \theta$  and  $s = \sin \theta$  to the  $i$  and  $j$  elements of  $v$ . On output,  $(v(i), v(j)) \leftarrow G(\theta)(v(i), v(j))$ .

## 14.18 Householder Transformations

A Householder transformation is a rank-1 modification of the identity matrix which can be used to zero out selected elements of a vector. A Householder matrix  $H$  takes the form,

$$H = I - \tau v v^T$$

where  $v$  is a vector (called the *Householder vector*) and  $\tau = 2/(v^T v)$ . The functions described in this section use the rank-1 structure of the Householder matrix to create and apply Householder transformations efficiently.

double **gsl\_linalg\_householder\_transform**(gsl\_vector \*w)

gsl\_complex **gsl\_linalg\_complex\_householder\_transform**(gsl\_vector\_complex \*w)

This function prepares a Householder transformation  $H = I - \tau v v^T$  which can be used to zero all the elements of the input vector  $w$  except the first. On output the Householder vector  $v$  is stored in  $w$  and the scalar  $\tau$  is returned. The householder vector  $v$  is normalized so that  $v[0] = 1$ , however this 1 is not stored in the output vector. Instead,  $w[0]$  is set to the first element of the transformed vector, so that if  $u = Hw$ ,  $w[0] = u[0]$  on output and the remainder of  $u$  is zero.

int **gsl\_linalg\_householder\_hm**(double tau, const gsl\_vector \*v, gsl\_matrix \*A)

```
int gsl_linalg_complex_householder_hm(gsl_complex tau, const gsl_vector_complex *v, gsl_matrix_complex *A)
```

This function applies the Householder matrix  $H$  defined by the scalar  $\tau$  and the vector  $v$  to the left-hand side of the matrix  $A$ . On output the result  $HA$  is stored in  $A$ .

```
int gsl_linalg_householder_mh(double tau, const gsl_vector *v, gsl_matrix *A)
```

```
int gsl_linalg_complex_householder_mh(gsl_complex tau, const gsl_vector_complex *v, gsl_matrix_complex *A)
```

This function applies the Householder matrix  $H$  defined by the scalar  $\tau$  and the vector  $v$  to the right-hand side of the matrix  $A$ . On output the result  $AH$  is stored in  $A$ .

```
int gsl_linalg_householder_hv(double tau, const gsl_vector *v, gsl_vector *w)
```

```
int gsl_linalg_complex_householder_hv(gsl_complex tau, const gsl_vector_complex *v, gsl_vector_complex *w)
```

This function applies the Householder transformation  $H$  defined by the scalar  $\tau$  and the vector  $v$  to the vector  $w$ . On output the result  $Hw$  is stored in  $w$ .

## 14.19 Householder solver for linear systems

```
int gsl_linalg_HH_solve(gsl_matrix *A, const gsl_vector *b, gsl_vector *x)
```

This function solves the system  $Ax = b$  directly using Householder transformations. On output the solution is stored in  $x$  and  $b$  is not modified. The matrix  $A$  is destroyed by the Householder transformations.

```
int gsl_linalg_HH_svx(gsl_matrix *A, gsl_vector *x)
```

This function solves the system  $Ax = b$  in-place using Householder transformations. On input  $x$  should contain the right-hand side  $b$ , which is replaced by the solution on output. The matrix  $A$  is destroyed by the Householder transformations.

## 14.20 Tridiagonal Systems

The functions described in this section efficiently solve symmetric, non-symmetric and cyclic tridiagonal systems with minimal storage. Note that the current implementations of these functions use a variant of Cholesky decomposition, so the tridiagonal matrix must be positive definite. For non-positive definite matrices, the functions return the error code `GSL_ESING`.

```
int gsl_linalg_solve_tridiag(const gsl_vector *diag, const gsl_vector *e, const gsl_vector *f, const gsl_vector *b, gsl_vector *x)
```

This function solves the general  $N$ -by- $N$  system  $Ax = b$  where  $A$  is tridiagonal ( $N \geq 2$ ). The super-diagonal and sub-diagonal vectors  $e$  and  $f$  must be one element shorter than the diagonal vector  $diag$ . The form of  $A$  for the 4-by-4 case is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ 0 & 0 & f_2 & d_3 \end{pmatrix}$$

```
int gsl_linalg_solve_symm_tridiag(const gsl_vector *diag, const gsl_vector *e, const gsl_vector *b, gsl_vector *x)
```

This function solves the general  $N$ -by- $N$  system  $Ax = b$  where  $A$  is symmetric tridiagonal ( $N \geq 2$ ). The off-diagonal vector  $e$  must be one element shorter than the diagonal vector  $diag$ . The form of  $A$  for the 4-by-4 case



is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & 0 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ 0 & 0 & e_2 & d_3 \end{pmatrix}$$

int **gsl\_linalg\_solve\_cyc\_tridiag**(const *gsl\_vector* \*diag, const *gsl\_vector* \*e, const *gsl\_vector* \*f, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the general  $N$ -by- $N$  system  $Ax = b$  where  $A$  is cyclic tridiagonal ( $N \geq 3$ ). The cyclic super-diagonal and sub-diagonal vectors  $e$  and  $f$  must have the same number of elements as the diagonal vector  $diag$ . The form of  $A$  for the 4-by-4 case is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & f_3 \\ f_0 & d_1 & e_1 & 0 \\ 0 & f_1 & d_2 & e_2 \\ e_3 & 0 & f_2 & d_3 \end{pmatrix}$$

int **gsl\_linalg\_solve\_symm\_cyc\_tridiag**(const *gsl\_vector* \*diag, const *gsl\_vector* \*e, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the general  $N$ -by- $N$  system  $Ax = b$  where  $A$  is symmetric cyclic tridiagonal ( $N \geq 3$ ). The cyclic off-diagonal vector  $e$  must have the same number of elements as the diagonal vector  $diag$ . The form of  $A$  for the 4-by-4 case is shown below,

$$A = \begin{pmatrix} d_0 & e_0 & 0 & e_3 \\ e_0 & d_1 & e_1 & 0 \\ 0 & e_1 & d_2 & e_2 \\ e_3 & 0 & e_2 & d_3 \end{pmatrix}$$

## 14.21 Triangular Systems

int **gsl\_linalg\_tri\_invert**(CBLAS\_UPLO\_t Uplo, CBLAS\_DIAG\_t Diag, *gsl\_matrix* \*T)

int **gsl\_linalg\_complex\_tri\_invert**(CBLAS\_UPLO\_t Uplo, CBLAS\_DIAG\_t Diag, *gsl\_matrix\_complex* \*T)

These functions compute the in-place inverse of the triangular matrix  $T$ , stored in the lower triangle when  $Uplo = CblasLower$  and upper triangle when  $Uplo = CblasUpper$ . The parameter  $Diag = CblasUnit$ ,  $CblasNonUnit$  specifies whether the matrix is unit triangular.

int **gsl\_linalg\_tri\_LTL**(*gsl\_matrix* \*L)

int **gsl\_linalg\_complex\_tri\_LHL**(*gsl\_matrix\_complex* \*L)

These functions compute the product  $L^T L$  (or  $L^\dagger L$ ) in-place and stores it in the lower triangle of  $L$  on output.

int **gsl\_linalg\_tri\_UL**(*gsl\_matrix* \*LU)

int **gsl\_linalg\_complex\_tri\_UL**(*gsl\_matrix\_complex* \*LU)

These functions compute the product  $UL$  where  $U$  is upper triangular and  $L$  is unit lower triangular, stored in  $LU$ , as computed by *gsl\_linalg\_LU\_decomp*() or *gsl\_linalg\_complex\_LU\_decomp*(). The product is computed in-place using Level 3 BLAS.

int **gsl\_linalg\_tri\_rcond**(CBLAS\_UPLO\_t Uplo, const *gsl\_matrix* \*A, double \*rcond, *gsl\_vector* \*work)

This function estimates the 1-norm reciprocal condition number of the triangular matrix  $A$ , using the lower triangle when  $Uplo$  is  $CblasLower$  and upper triangle when  $Uplo$  is  $CblasUpper$ . The reciprocal condition number  $1 / (\|A\|_1 \|A^{-1}\|_1)$  is stored in  $rcond$  on output. Additional workspace of size  $3N$  is required in  $work$ .

## 14.22 Banded Systems

Band matrices are sparse matrices whose non-zero entries are confined to a diagonal *band*. From a storage point of view, significant savings can be achieved by storing only the non-zero diagonals of a banded matrix. Algorithms such as LU and Cholesky factorizations preserve the band structure of these matrices. Computationally, working with compact banded matrices is preferable to working on the full dense matrix with many zero entries.

### 14.22.1 General Banded Format

An example of a general banded matrix is given below.

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & 0 \\ \delta_1 & \alpha_2 & \beta_2 & \gamma_2 & 0 & 0 \\ 0 & \delta_2 & \alpha_3 & \beta_3 & \gamma_3 & 0 \\ 0 & 0 & \delta_3 & \alpha_4 & \beta_4 & \gamma_4 \\ 0 & 0 & 0 & \delta_4 & \alpha_5 & \beta_5 \\ 0 & 0 & 0 & 0 & \delta_5 & \alpha_6 \end{pmatrix}$$

This matrix has a *lower bandwidth* of 1 and an *upper bandwidth* of 2. The lower bandwidth is the number of non-zero subdiagonals, and the upper bandwidth is the number of non-zero superdiagonals. A  $(p, q)$  banded matrix has a lower bandwidth  $p$  and upper bandwidth  $q$ . For example, diagonal matrices are  $(0, 0)$ , tridiagonal matrices are  $(1, 1)$ , and upper triangular matrices are  $(0, N - 1)$  banded matrices.

The corresponding 6-by-4 packed banded matrix looks like

$$AB = \begin{pmatrix} * & * & \alpha_1 & \delta_1 \\ * & \beta_1 & \alpha_2 & \delta_2 \\ \gamma_1 & \beta_2 & \alpha_3 & \delta_3 \\ \gamma_2 & \beta_3 & \alpha_4 & \delta_4 \\ \gamma_3 & \beta_4 & \alpha_5 & \delta_5 \\ \gamma_4 & \beta_5 & \alpha_6 & * \end{pmatrix}$$

where the superdiagonals are stored in columns, followed by the diagonal, followed by the subdiagonals. The entries marked by  $*$  are not referenced by the banded routines. With this format, each row of  $AB$  corresponds to the non-zero entries of the corresponding column of  $A$ . For an  $N$ -by- $N$  matrix  $A$ , the dimension of  $AB$  will be  $N$ -by- $(p + q + 1)$ .

### 14.22.2 Symmetric Banded Format

Symmetric banded matrices allow for additional storage savings. As an example, consider the following  $6 \times 6$  symmetric banded matrix with lower bandwidth  $p = 2$ :

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & 0 \\ \beta_1 & \alpha_2 & \beta_2 & \gamma_2 & 0 & 0 \\ \gamma_1 & \beta_2 & \alpha_3 & \beta_3 & \gamma_3 & 0 \\ 0 & \gamma_2 & \beta_3 & \alpha_4 & \beta_4 & \gamma_4 \\ 0 & 0 & \gamma_3 & \beta_4 & \alpha_5 & \beta_5 \\ 0 & 0 & 0 & \gamma_4 & \beta_5 & \alpha_6 \end{pmatrix}$$

The packed symmetric banded  $6 \times 3$  matrix will look like:

$$AB = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 \\ \alpha_2 & \beta_2 & \gamma_2 \\ \alpha_3 & \beta_3 & \gamma_3 \\ \alpha_4 & \beta_4 & \gamma_4 \\ \alpha_5 & \beta_5 & * \\ \alpha_6 & * & * \end{pmatrix}$$

The entries marked by \* are not referenced by the symmetric banded routines. The relationship between the packed format and original matrix is,

$$AB(i, j) = A(i, i + j) = A(i + j, i)$$

for  $i = 0, \dots, N - 1, j = 0, \dots, p$ . Conversely,

$$A(i, j) = AB(j, i - j)$$

for  $i = 0, \dots, N - 1, j = \max(0, i - p), \dots, i$ .

**Warning:** Note that this format is the transpose of the symmetric banded format used by LAPACK. In order to develop efficient routines for symmetric banded matrices, it helps to have the nonzero elements in each column in contiguous memory locations. Since C uses row-major order, GSL stores the columns in the rows of the packed banded format, while LAPACK, written in Fortran, uses the transposed format.

### 14.22.3 Banded LU Decomposition

The routines in this section are designed to factor banded  $M$ -by- $N$  matrices with an LU factorization,  $PA = LU$ . The matrix  $A$  is banded of type  $(p, q)$ , i.e. a lower bandwidth of  $p$  and an upper bandwidth of  $q$ . See *LU Decomposition* for more information on the factorization. For banded  $(p, q)$  matrices, the  $U$  factor will have an upper bandwidth of  $p + q$ , while the  $L$  factor will have a lower bandwidth of at most  $p$ . Therefore, additional storage is needed to store the  $p$  additional bands of  $U$ .

As an example, consider the  $M = N = 7$  matrix with lower bandwidth  $p = 3$  and upper bandwidth  $q = 2$ ,

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & 0 & 0 \\ \delta_1 & \alpha_2 & \beta_2 & \gamma_2 & 0 & 0 & 0 \\ \epsilon_1 & \delta_2 & \alpha_3 & \beta_3 & \gamma_3 & 0 & 0 \\ \zeta_1 & \epsilon_2 & \delta_3 & \alpha_4 & \beta_4 & \gamma_4 & 0 \\ 0 & \zeta_2 & \epsilon_3 & \delta_4 & \alpha_5 & \beta_5 & \gamma_5 \\ 0 & 0 & \zeta_3 & \epsilon_4 & \delta_5 & \alpha_6 & \beta_6 \\ 0 & 0 & 0 & \zeta_4 & \epsilon_5 & \delta_6 & \alpha_7 \end{pmatrix}$$

The corresponding  $N$ -by- $2p + q + 1$  packed banded matrix looks like

$$AB = \begin{pmatrix} * & * & * & * & * & \alpha_1 & \delta_1 & \epsilon_1 & \zeta_1 \\ * & * & * & * & \beta_1 & \alpha_2 & \delta_2 & \epsilon_2 & \zeta_2 \\ * & * & * & \gamma_1 & \beta_2 & \alpha_3 & \delta_3 & \epsilon_3 & \zeta_3 \\ * & * & - & \gamma_2 & \beta_3 & \alpha_4 & \delta_4 & \epsilon_4 & \zeta_4 \\ * & - & - & \gamma_3 & \beta_4 & \alpha_5 & \delta_5 & \epsilon_5 & * \\ - & - & - & \gamma_4 & \beta_5 & \alpha_6 & \delta_6 & * & * \\ - & - & - & \gamma_5 & \beta_6 & \alpha_7 & * & * & * \end{pmatrix}$$

Entries marked with  $-$  are used to store the additional  $p$  diagonals of the  $U$  factor. Entries marked with  $*$  are not referenced by the banded routines.

int **gsl\_linalg\_LU\_band\_decomp**(const size\_t M, const size\_t lb, const size\_t ub, *gsl\_matrix* \*AB, *gsl\_vector\_uint* \*piv)

This function computes the LU factorization of the banded matrix  $AB$  which is stored in packed band format (see above) and has dimension  $N$ -by- $2p + q + 1$ . The number of rows  $M$  of the original matrix is provided in  $M$ . The lower bandwidth  $p$  is provided in  $lb$  and the upper bandwidth  $q$  is provided in  $ub$ . The vector  $piv$  has length  $\min(M, N)$  and stores the pivot indices on output (for  $0 \leq i < \min(M, N)$ , row  $i$  of the matrix was interchanged with row  $piv[i]$ ). On output,  $AB$  contains both the  $L$  and  $U$  factors in packed format.

int **gsl\_linalg\_LU\_band\_solve**(const size\_t lb, const size\_t ub, const *gsl\_matrix* \*LUB, const *gsl\_vector\_uint* \*piv, const *gsl\_vector* \*b, *gsl\_vector* \*x)

This function solves the square system  $Ax = b$  using the banded LU factorization ( $LUB$ ,  $piv$ ) computed by `gsl_linalg_LU_band_decomp()`. The lower and upper bandwidths are provided in  $lb$  and  $ub$  respectively. The right hand side vector is provided in  $b$ . The solution vector is stored in  $x$  on output.

int **gsl\_linalg\_LU\_band\_svx**(const size\_t lb, const size\_t ub, const *gsl\_matrix* \*LUB, const *gsl\_vector\_uint* \*piv, *gsl\_vector* \*x)

This function solves the square system  $Ax = b$  in-place, using the banded LU factorization ( $LUB$ ,  $piv$ ) computed by `gsl_linalg_LU_band_decomp()`. The lower and upper bandwidths are provided in  $lb$  and  $ub$  respectively. On input, the right hand side vector  $b$  is provided in  $x$ , which is replaced by the solution vector  $x$  on output.

int **gsl\_linalg\_LU\_band\_unpack**(const size\_t M, const size\_t lb, const size\_t ub, const *gsl\_matrix* \*LUB, const *gsl\_vector\_uint* \*piv, *gsl\_matrix* \*L, *gsl\_matrix* \*U)

This function unpacks the banded LU factorization ( $LUB$ ,  $piv$ ) previously computed by `gsl_linalg_LU_band_decomp()` into the matrices  $L$  and  $U$ . The matrix  $U$  has dimension  $\min(M, N)$ -by- $N$  and stores the upper triangular factor on output. The matrix  $L$  has dimension  $M$ -by- $\min(M, N)$  and stores the matrix  $P^T L$  on output.

#### 14.22.4 Banded Cholesky Decomposition

The routines in this section are designed to factor and solve  $N$ -by- $N$  linear systems of the form  $Ax = b$  where  $A$  is a banded, symmetric, and positive definite matrix with lower bandwidth  $p$ . See *Cholesky Decomposition* for more information on the factorization. The lower triangular factor of the Cholesky decomposition preserves the same banded structure as the matrix  $A$ , enabling an efficient algorithm which overwrites the original matrix with the  $L$  factor.

int **gsl\_linalg\_cholesky\_band\_decomp**(*gsl\_matrix* \*A)

This function factorizes the symmetric, positive-definite square matrix  $A$  into the Cholesky decomposition  $A = LL^T$ . The input matrix  $A$  is given in *symmetric banded format*, and has dimensions  $N$ -by- $(p + 1)$ , where  $p$  is the lower bandwidth of the matrix. On output, the entries of  $A$  are replaced by the entries of the matrix  $L$  in the same format. In addition, the lower right element of  $A$  is used to store the matrix 1-norm, used later by `gsl_linalg_cholesky_band_rcond()` to calculate the reciprocal condition number.

If the matrix is not positive-definite then the decomposition will fail, returning the error code `GSL_EDOM`. When testing whether a matrix is positive-definite, disable the error handler first to avoid triggering an error.

int **gsl\_linalg\_cholesky\_band\_solve**(const *gsl\_matrix* \*LLT, const *gsl\_vector* \*b, *gsl\_vector* \*x)

int **gsl\_linalg\_cholesky\_band\_solve**m(const *gsl\_matrix* \*LLT, const *gsl\_matrix* \*B, *gsl\_matrix* \*X)

This function solves the symmetric banded system  $Ax = b$  (or  $AX = B$ ) using the Cholesky decomposition of  $A$  held in the matrix  $LLT$  which must have been previously computed by `gsl_linalg_cholesky_band_decomp()`.

int **gsl\_linalg\_cholesky\_band\_svx**(const *gsl\_matrix* \*LLT, *gsl\_vector* \*x)

int **gsl\_linalg\_cholesky\_band\_svx**m(const *gsl\_matrix* \*LLT, *gsl\_matrix* \*X)

This function solves the symmetric banded system  $Ax = b$  (or  $AX = B$ ) in-place using the Cholesky decomposition of  $A$  held in the matrix  $LLT$  which must have been previously computed by `gsl_linalg_cholesky_band_decomp()`. On input  $x$  (or  $X$ ) should contain the right-hand side  $b$  (or  $B$ ), which is replaced by the solution on output.

int **gsl\_linalg\_cholesky\_band\_invert**(const *gsl\_matrix* \*LLT, *gsl\_matrix* \*Ainv)

This function computes the inverse of a symmetric banded matrix from its Cholesky decomposition  $LLT$ , which must have been previously computed by `gsl_linalg_cholesky_band_decomp()`. On output, the inverse is stored in  $Ainv$ , using both the lower and upper portions.

- int **gsl\_linalg\_cholesky\_band\_unpack**(const *gsl\_matrix* \*LLT, *gsl\_matrix* \*L)  
 This function unpacks the lower triangular Cholesky factor from *LLT* and stores it in the lower triangular portion of the  $N$ -by- $N$  matrix *L*. The upper triangular portion of *L* is not referenced.
- int **gsl\_linalg\_cholesky\_band\_scale**(const *gsl\_matrix* \*A, *gsl\_vector* \*S)  
 This function calculates a diagonal scaling transformation of the symmetric, positive definite banded matrix *A*, such that *SAS* has a condition number within a factor of  $N$  of the matrix of smallest possible condition number over all possible diagonal scalings. On output, *S* contains the scale factors, given by  $S_i = 1/\sqrt{A_{ii}}$ . For any  $A_{ii} \leq 0$ , the corresponding scale factor  $S_i$  is set to 1.
- int **gsl\_linalg\_cholesky\_band\_scale\_apply**(*gsl\_matrix* \*A, const *gsl\_vector* \*S)  
 This function applies the scaling transformation *S* to the banded symmetric positive definite matrix *A*. On output, *A* is replaced by *SAS*.
- int **gsl\_linalg\_cholesky\_band\_rcond**(const *gsl\_matrix* \*LLT, double \*rcond, *gsl\_vector* \*work)  
 This function estimates the reciprocal condition number (using the 1-norm) of the symmetric banded positive definite matrix *A*, using its Cholesky decomposition provided in *LLT*. The reciprocal condition number estimate, defined as  $1/(\|A\|_1 \cdot \|A^{-1}\|_1)$ , is stored in *rcond*. Additional workspace of size  $3N$  is required in *work*.

### 14.22.5 Banded LDLT Decomposition

The routines in this section are designed to factor and solve  $N$ -by- $N$  linear systems of the form  $Ax = b$  where *A* is a banded, symmetric, and non-singular matrix with lower bandwidth  $p$ . See *LDLT Decomposition* for more information on the factorization. The lower triangular factor of the  $LDL^T$  decomposition preserves the same banded structure as the matrix *A*, enabling an efficient algorithm which overwrites the original matrix with the *L* and *D* factors.

- int **gsl\_linalg\_ldlt\_band\_decomp**(*gsl\_matrix* \*A)  
 This function factorizes the symmetric, non-singular square matrix *A* into the decomposition  $A = LDL^T$ . The input matrix *A* is given in *symmetric banded format*, and has dimensions  $N$ -by- $(p + 1)$ , where  $p$  is the lower bandwidth of the matrix. On output, the entries of *A* are replaced by the entries of the matrices *D* and *L* in the same format.
- If the matrix is singular then the decomposition will fail, returning the error code *GSL\_EDOM*.
- int **gsl\_linalg\_ldlt\_band\_solve**(const *gsl\_matrix* \*LDLT, const *gsl\_vector* \*b, *gsl\_vector* \*x)  
 This function solves the symmetric banded system  $Ax = b$  using the  $LDL^T$  decomposition of *A* held in the matrix *LDLT* which must have been previously computed by *gsl\_linalg\_ldlt\_band\_decomp()*.
- int **gsl\_linalg\_ldlt\_band\_svx**(const *gsl\_matrix* \*LDLT, *gsl\_vector* \*x)  
 This function solves the symmetric banded system  $Ax = b$  in-place using the  $LDL^T$  decomposition of *A* held in the matrix *LDLT* which must have been previously computed by *gsl\_linalg\_ldlt\_band\_decomp()*. On input *x* should contain the right-hand side *b*, which is replaced by the solution on output.
- int **gsl\_linalg\_ldlt\_band\_unpack**(const *gsl\_matrix* \*LDLT, *gsl\_matrix* \*L, *gsl\_vector* \*D)  
 This function unpacks the unit lower triangular factor *L* from *LDLT* and stores it in the lower triangular portion of the  $N$ -by- $N$  matrix *L*. The upper triangular portion of *L* is not referenced. The diagonal matrix *D* is stored in the vector *D*.
- int **gsl\_linalg\_ldlt\_band\_rcond**(const *gsl\_matrix* \*LDLT, double \*rcond, *gsl\_vector* \*work)  
 This function estimates the reciprocal condition number (using the 1-norm) of the symmetric banded nonsingular matrix *A*, using its  $LDL^T$  decomposition provided in *LDLT*. The reciprocal condition number estimate, defined as  $1/(\|A\|_1 \cdot \|A^{-1}\|_1)$ , is stored in *rcond*. Additional workspace of size  $3N$  is required in *work*.

## 14.23 Balancing

The process of balancing a matrix applies similarity transformations to make the rows and columns have comparable norms. This is useful, for example, to reduce roundoff errors in the solution of eigenvalue problems. Balancing a matrix  $A$  consists of replacing  $A$  with a similar matrix

$$A' = D^{-1}AD$$

where  $D$  is a diagonal matrix whose entries are powers of the floating point radix.

int `gsl_linalg_balance_matrix`(*gsl\_matrix* \*A, *gsl\_vector* \*D)

This function replaces the matrix  $A$  with its balanced counterpart and stores the diagonal elements of the similarity transformation into the vector  $D$ .

## 14.24 Examples

The following program solves the linear system  $Ax = b$ . The system to be solved is,

$$\begin{pmatrix} 0.18 & 0.60 & 0.57 & 0.96 \\ 0.41 & 0.24 & 0.99 & 0.58 \\ 0.14 & 0.30 & 0.97 & 0.66 \\ 0.51 & 0.13 & 0.19 & 0.85 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}$$

and the solution is found using LU decomposition of the matrix  $A$ .

```
#include <stdio.h>
#include <gsl/gsl_linalg.h>

int
main (void)
{
  double a_data[] = { 0.18, 0.60, 0.57, 0.96,
                     0.41, 0.24, 0.99, 0.58,
                     0.14, 0.30, 0.97, 0.66,
                     0.51, 0.13, 0.19, 0.85 };

  double b_data[] = { 1.0, 2.0, 3.0, 4.0 };

  gsl_matrix_view m
    = gsl_matrix_view_array (a_data, 4, 4);

  gsl_vector_view b
    = gsl_vector_view_array (b_data, 4);

  gsl_vector *x = gsl_vector_alloc (4);

  int s;

  gsl_permutation * p = gsl_permutation_alloc (4);

  gsl_linalg_LU_decomp (&m.matrix, p, &s);
```

(continues on next page)

(continued from previous page)

```

gsl_linalg_LU_solve (&m.matrix, p, &b.vector, x);

printf ("x = \n");
gsl_vector_fprintf (stdout, x, "%g");

gsl_permutation_free (p);
gsl_vector_free (x);
return 0;
}

```

Here is the output from the program,

```

x =
-4.05205
-12.6056
1.66091
8.69377

```

This can be verified by multiplying the solution  $x$  by the original matrix  $A$  using GNU octave,

```

octave> A = [ 0.18, 0.60, 0.57, 0.96;
             0.41, 0.24, 0.99, 0.58;
             0.14, 0.30, 0.97, 0.66;
             0.51, 0.13, 0.19, 0.85 ];

octave> x = [ -4.05205; -12.6056; 1.66091; 8.69377];

octave> A * x
ans =
 1.0000
 2.0000
 3.0000
 4.0000

```

This reproduces the original right-hand side vector,  $b$ , in accordance with the equation  $Ax = b$ .

## 14.25 References and Further Reading

Further information on the algorithms described in this section can be found in the following book,

- G. H. Golub, C. F. Van Loan, “Matrix Computations” (3rd Ed, 1996), Johns Hopkins University Press, ISBN 0-8018-5414-8.

The LAPACK library is described in the following manual,

- *LAPACK Users’ Guide* (Third Edition, 1999), Published by SIAM, ISBN 0-89871-447-8

The LAPACK source code can be found at <http://www.netlib.org/lapack>, along with an online copy of the users guide.

Further information on recursive Level 3 BLAS algorithms may be found in the following paper,

- E. Peise and P. Bientinesi, “Recursive algorithms for dense linear algebra: the ReLAPACK collection”, <http://arxiv.org/abs/1602.06763>, 2016.

The recursive Level 3 BLAS QR decomposition is described in the following paper,

- E. Elmroth and F. G. Gustavson, 2000. Applying recursion to serial and parallel QR factorization leads to better performance. IBM Journal of Research and Development, 44(4), pp.605-624.

The Modified Golub-Reinsch algorithm is described in the following paper,

- T.F. Chan, “An Improved Algorithm for Computing the Singular Value Decomposition”, ACM Transactions on Mathematical Software, 8 (1982), pp 72–83.

The Jacobi algorithm for singular value decomposition is described in the following papers,

- J.C. Nash, “A one-sided transformation method for the singular value decomposition and algebraic eigenproblem”, Computer Journal, Volume 18, Number 1 (1975), p 74–76
- J.C. Nash and S. Shlien “Simple algorithms for the partial singular value decomposition”, Computer Journal, Volume 30 (1987), p 268–275.
- J. Demmel, K. Veselic, “Jacobi’s Method is more accurate than QR”, Lapack Working Note 15 (LAWN-15), October 1989. Available from netlib, <http://www.netlib.org/lapack/> in the lawns or lawnspdf directories.

The algorithm for estimating a matrix condition number is described in the following paper,

- N. J. Higham, “FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation”, ACM Trans. Math. Soft., vol. 14, no. 4, pp. 381-396, December 1988.



## EIGENSYSTEMS

This chapter describes functions for computing eigenvalues and eigenvectors of matrices. There are routines for real symmetric, real nonsymmetric, complex hermitian, real generalized symmetric-definite, complex generalized hermitian-definite, and real generalized nonsymmetric eigensystems. Eigenvalues can be computed with or without eigenvectors. The hermitian and real symmetric matrix algorithms are symmetric bidiagonalization followed by QR reduction. The nonsymmetric algorithm is the Francis QR double-shift. The generalized nonsymmetric algorithm is the QZ method due to Moler and Stewart.

The functions described in this chapter are declared in the header file `gsl_eigen.h`.

### 15.1 Real Symmetric Matrices

For real symmetric matrices, the library uses the symmetric bidiagonalization and QR reduction method. This is described in Golub & van Loan, section 8.3. The computed eigenvalues are accurate to an absolute accuracy of  $\epsilon \|A\|_2$ , where  $\epsilon$  is the machine precision.

type **gsl\_eigen\_symm\_workspace**

This workspace contains internal parameters used for solving symmetric eigenvalue problems.

*gsl\_eigen\_symm\_workspace* \***gsl\_eigen\_symm\_alloc**(const size\_t n)

This function allocates a workspace for computing eigenvalues of  $n$ -by- $n$  real symmetric matrices. The size of the workspace is  $O(2n)$ .

void **gsl\_eigen\_symm\_free**(*gsl\_eigen\_symm\_workspace* \*w)

This function frees the memory associated with the workspace  $w$ .

int **gsl\_eigen\_symm**(*gsl\_matrix* \*A, *gsl\_vector* \*eval, *gsl\_eigen\_symm\_workspace* \*w)

This function computes the eigenvalues of the real symmetric matrix  $A$ . Additional workspace of the appropriate size must be provided in  $w$ . The diagonal and lower triangular part of  $A$  are destroyed during the computation, but the strict upper triangular part is not referenced. The eigenvalues are stored in the vector *eval* and are unordered.

type **gsl\_eigen\_symmv\_workspace**

This workspace contains internal parameters used for solving symmetric eigenvalue and eigenvector problems.

*gsl\_eigen\_symmv\_workspace* \***gsl\_eigen\_symmv\_alloc**(const size\_t n)

This function allocates a workspace for computing eigenvalues and eigenvectors of  $n$ -by- $n$  real symmetric matrices. The size of the workspace is  $O(4n)$ .

void **gsl\_eigen\_symmv\_free**(*gsl\_eigen\_symmv\_workspace* \*w)

This function frees the memory associated with the workspace  $w$ .

int **gsl\_eigen\_symmv**(*gsl\_matrix* \*A, *gsl\_vector* \*eval, *gsl\_matrix* \*evec, *gsl\_eigen\_symmv\_workspace* \*w)

This function computes the eigenvalues and eigenvectors of the real symmetric matrix  $A$ . Additional workspace of the appropriate size must be provided in  $w$ . The diagonal and lower triangular part of  $A$  are destroyed during the computation, but the strict upper triangular part is not referenced. The eigenvalues are stored in the vector

*eval* and are unordered. The corresponding eigenvectors are stored in the columns of the matrix *eval*. For example, the eigenvector in the first column corresponds to the first eigenvalue. The eigenvectors are guaranteed to be mutually orthogonal and normalised to unit magnitude.

## 15.2 Complex Hermitian Matrices

For hermitian matrices, the library uses the complex form of the symmetric bidiagonalization and QR reduction method.

type **gsl\_eigen\_herm\_workspace**

This workspace contains internal parameters used for solving hermitian eigenvalue problems.

*gsl\_eigen\_herm\_workspace* \***gsl\_eigen\_herm\_alloc**(const size\_t n)

This function allocates a workspace for computing eigenvalues of *n*-by-*n* complex hermitian matrices. The size of the workspace is  $O(3n)$ .

void **gsl\_eigen\_herm\_free**(*gsl\_eigen\_herm\_workspace* \*w)

This function frees the memory associated with the workspace *w*.

int **gsl\_eigen\_herm**(*gsl\_matrix\_complex* \*A, *gsl\_vector* \*eval, *gsl\_eigen\_herm\_workspace* \*w)

This function computes the eigenvalues of the complex hermitian matrix *A*. Additional workspace of the appropriate size must be provided in *w*. The diagonal and lower triangular part of *A* are destroyed during the computation, but the strict upper triangular part is not referenced. The imaginary parts of the diagonal are assumed to be zero and are not referenced. The eigenvalues are stored in the vector *eval* and are unordered.

type **gsl\_eigen\_hermv\_workspace**

This workspace contains internal parameters used for solving hermitian eigenvalue and eigenvector problems.

*gsl\_eigen\_hermv\_workspace* \***gsl\_eigen\_hermv\_alloc**(const size\_t n)

This function allocates a workspace for computing eigenvalues and eigenvectors of *n*-by-*n* complex hermitian matrices. The size of the workspace is  $O(5n)$ .

void **gsl\_eigen\_hermv\_free**(*gsl\_eigen\_hermv\_workspace* \*w)

This function frees the memory associated with the workspace *w*.

int **gsl\_eigen\_hermv**(*gsl\_matrix\_complex* \*A, *gsl\_vector* \*eval, *gsl\_matrix\_complex* \*evec, *gsl\_eigen\_hermv\_workspace* \*w)

This function computes the eigenvalues and eigenvectors of the complex hermitian matrix *A*. Additional workspace of the appropriate size must be provided in *w*. The diagonal and lower triangular part of *A* are destroyed during the computation, but the strict upper triangular part is not referenced. The imaginary parts of the diagonal are assumed to be zero and are not referenced. The eigenvalues are stored in the vector *eval* and are unordered. The corresponding complex eigenvectors are stored in the columns of the matrix *evec*. For example, the eigenvector in the first column corresponds to the first eigenvalue. The eigenvectors are guaranteed to be mutually orthogonal and normalised to unit magnitude.

## 15.3 Real Nonsymmetric Matrices

The solution of the real nonsymmetric eigensystem problem for a matrix *A* involves computing the Schur decomposition

$$A = ZTZ^T$$

where *Z* is an orthogonal matrix of Schur vectors and *T*, the Schur form, is quasi upper triangular with diagonal 1-by-1 blocks which are real eigenvalues of *A*, and diagonal 2-by-2 blocks whose eigenvalues are complex conjugate eigenvalues of *A*. The algorithm used is the double-shift Francis method.

type **gsl\_eigen\_nonsymm\_workspace**

This workspace contains internal parameters used for solving nonsymmetric eigenvalue problems.

`gsl_eigen_nonsymm_workspace *gsl_eigen_nonsymm_alloc(const size_t n)`

This function allocates a workspace for computing eigenvalues of  $n$ -by- $n$  real nonsymmetric matrices. The size of the workspace is  $O(2n)$ .

void `gsl_eigen_nonsymm_free(gsl_eigen_nonsymm_workspace *w)`

This function frees the memory associated with the workspace  $w$ .

void `gsl_eigen_nonsymm_params(const int compute_t, const int balance, gsl_eigen_nonsymm_workspace *w)`

This function sets some parameters which determine how the eigenvalue problem is solved in subsequent calls to `gsl_eigen_nonsymm()`.

If `compute_t` is set to 1, the full Schur form  $T$  will be computed by `gsl_eigen_nonsymm()`. If it is set to 0,  $T$  will not be computed (this is the default setting). Computing the full Schur form  $T$  requires approximately 1.5–2 times the number of flops.

If `balance` is set to 1, a balancing transformation is applied to the matrix prior to computing eigenvalues. This transformation is designed to make the rows and columns of the matrix have comparable norms, and can result in more accurate eigenvalues for matrices whose entries vary widely in magnitude. See *Balancing* for more information. Note that the balancing transformation does not preserve the orthogonality of the Schur vectors, so if you wish to compute the Schur vectors with `gsl_eigen_nonsymm_Z()` you will obtain the Schur vectors of the balanced matrix instead of the original matrix. The relationship will be

$$T = Q^T D^{-1} A D Q$$

where  $Q$  is the matrix of Schur vectors for the balanced matrix, and  $D$  is the balancing transformation. Then `gsl_eigen_nonsymm_Z()` will compute a matrix  $Z$  which satisfies

$$T = Z^{-1} A Z$$

with  $Z = DQ$ . Note that  $Z$  will not be orthogonal. For this reason, balancing is not performed by default.

int `gsl_eigen_nonsymm(gsl_matrix *A, gsl_vector_complex *eval, gsl_eigen_nonsymm_workspace *w)`

This function computes the eigenvalues of the real nonsymmetric matrix  $A$  and stores them in the vector `eval`. If  $T$  is desired, it is stored in the upper portion of  $A$  on output. Otherwise, on output, the diagonal of  $A$  will contain the 1-by-1 real eigenvalues and 2-by-2 complex conjugate eigenvalue systems, and the rest of  $A$  is destroyed. In rare cases, this function may fail to find all eigenvalues. If this happens, an error code is returned and the number of converged eigenvalues is stored in `w->n_evals`. The converged eigenvalues are stored in the beginning of `eval`.

int `gsl_eigen_nonsymm_Z(gsl_matrix *A, gsl_vector_complex *eval, gsl_matrix *Z, gsl_eigen_nonsymm_workspace *w)`

This function is identical to `gsl_eigen_nonsymm()` except that it also computes the Schur vectors and stores them into  $Z$ .

type `gsl_eigen_nonsymmv_workspace`

This workspace contains internal parameters used for solving nonsymmetric eigenvalue and eigenvector problems.

`gsl_eigen_nonsymmv_workspace *gsl_eigen_nonsymmv_alloc(const size_t n)`

This function allocates a workspace for computing eigenvalues and eigenvectors of  $n$ -by- $n$  real nonsymmetric matrices. The size of the workspace is  $O(5n)$ .

void `gsl_eigen_nonsymmv_free(gsl_eigen_nonsymmv_workspace *w)`

This function frees the memory associated with the workspace  $w$ .

void `gsl_eigen_nonsymmv_params(const int balance, gsl_eigen_nonsymm_workspace *w)`

This function sets parameters which determine how the eigenvalue problem is solved in subsequent calls to `gsl_eigen_nonsymmv()`. If `balance` is set to 1, a balancing transformation is applied to the matrix. See `gsl_eigen_nonsymm_params()` for more information. Balancing is turned off by default since it does not preserve the orthogonality of the Schur vectors.

```
int gsl_eigen_nonsymmv(gsl_matrix *A, gsl_vector_complex *eval, gsl_matrix_complex *evec,
                      gsl_eigen_nonsymmv_workspace *w)
```

This function computes eigenvalues and right eigenvectors of the  $n$ -by- $n$  real nonsymmetric matrix  $A$ . It first calls `gsl_eigen_nonsymm()` to compute the eigenvalues, Schur form  $T$ , and Schur vectors. Then it finds eigenvectors of  $T$  and backtransforms them using the Schur vectors. The Schur vectors are destroyed in the process, but can be saved by using `gsl_eigen_nonsymmv_Z()`. The computed eigenvectors are normalized to have unit magnitude. On output, the upper portion of  $A$  contains the Schur form  $T$ . If `gsl_eigen_nonsymm()` fails, no eigenvectors are computed, and an error code is returned.

```
int gsl_eigen_nonsymmv_Z(gsl_matrix *A, gsl_vector_complex *eval, gsl_matrix_complex *evec, gsl_matrix *Z,
                        gsl_eigen_nonsymmv_workspace *w)
```

This function is identical to `gsl_eigen_nonsymmv()` except that it also saves the Schur vectors into  $Z$ .

## 15.4 Real Generalized Symmetric-Definite Eigensystems

The real generalized symmetric-definite eigenvalue problem is to find eigenvalues  $\lambda$  and eigenvectors  $x$  such that

$$Ax = \lambda Bx$$

where  $A$  and  $B$  are symmetric matrices, and  $B$  is positive-definite. This problem reduces to the standard symmetric eigenvalue problem by applying the Cholesky decomposition to  $B$ :

$$\begin{aligned} Ax &= \lambda Bx \\ Ax &= \lambda LL^T x \\ (L^{-1}AL^{-T})L^T x &= \lambda L^T x \end{aligned}$$

Therefore, the problem becomes  $Cy = \lambda y$  where  $C = L^{-1}AL^{-T}$  is symmetric, and  $y = L^T x$ . The standard symmetric eigensolver can be applied to the matrix  $C$ . The resulting eigenvectors are backtransformed to find the vectors of the original problem. The eigenvalues and eigenvectors of the generalized symmetric-definite eigenproblem are always real.

```
type gsl_eigen_gensymm_workspace
```

This workspace contains internal parameters used for solving generalized symmetric eigenvalue problems.

```
gsl_eigen_gensymm_workspace *gsl_eigen_gensymm_alloc(const size_t n)
```

This function allocates a workspace for computing eigenvalues of  $n$ -by- $n$  real generalized symmetric-definite eigensystems. The size of the workspace is  $O(2n)$ .

```
void gsl_eigen_gensymm_free(gsl_eigen_gensymm_workspace *w)
```

This function frees the memory associated with the workspace  $w$ .

```
int gsl_eigen_gensymm(gsl_matrix *A, gsl_matrix *B, gsl_vector *eval, gsl_eigen_gensymm_workspace *w)
```

This function computes the eigenvalues of the real generalized symmetric-definite matrix pair  $(A, B)$ , and stores them in  $eval$ , using the method outlined above. On output,  $B$  contains its Cholesky decomposition and  $A$  is destroyed.

```
type gsl_eigen_gensymmv_workspace
```

This workspace contains internal parameters used for solving generalized symmetric eigenvalue and eigenvector problems.

```
gsl_eigen_gensymmv_workspace *gsl_eigen_gensymmv_alloc(const size_t n)
```

This function allocates a workspace for computing eigenvalues and eigenvectors of  $n$ -by- $n$  real generalized symmetric-definite eigensystems. The size of the workspace is  $O(4n)$ .

```
void gsl_eigen_gensymmv_free(gsl_eigen_gensymmv_workspace *w)
```

This function frees the memory associated with the workspace  $w$ .

```
int gsl_eigen_gensymm(gsl_matrix *A, gsl_matrix *B, gsl_vector *eval, gsl_matrix *evec,
                    gsl_eigen_gensymm_workspace *w)
```

This function computes the eigenvalues and eigenvectors of the real generalized symmetric-definite matrix pair  $(A, B)$ , and stores them in *eval* and *evec* respectively. The computed eigenvectors are normalized to have unit magnitude. On output, *B* contains its Cholesky decomposition and *A* is destroyed.

## 15.5 Complex Generalized Hermitian-Definite Eigensystems

The complex generalized hermitian-definite eigenvalue problem is to find eigenvalues  $\lambda$  and eigenvectors  $x$  such that

$$Ax = \lambda Bx$$

where  $A$  and  $B$  are hermitian matrices, and  $B$  is positive-definite. Similarly to the real case, this can be reduced to  $Cy = \lambda y$  where  $C = L^{-1}AL^{-\dagger}$  is hermitian, and  $y = L^\dagger x$ . The standard hermitian eigensolver can be applied to the matrix  $C$ . The resulting eigenvectors are backtransformed to find the vectors of the original problem. The eigenvalues of the generalized hermitian-definite eigenproblem are always real.

type **gsl\_eigen\_genherm\_workspace**

This workspace contains internal parameters used for solving generalized hermitian eigenvalue problems.

```
gsl_eigen_genherm_workspace *gsl_eigen_genherm_alloc(const size_t n)
```

This function allocates a workspace for computing eigenvalues of  $n$ -by- $n$  complex generalized hermitian-definite eigensystems. The size of the workspace is  $O(3n)$ .

```
void gsl_eigen_genherm_free(gsl_eigen_genherm_workspace *w)
```

This function frees the memory associated with the workspace *w*.

```
int gsl_eigen_genherm(gsl_matrix_complex *A, gsl_matrix_complex *B, gsl_vector *eval,
                    gsl_eigen_genherm_workspace *w)
```

This function computes the eigenvalues of the complex generalized hermitian-definite matrix pair  $(A, B)$ , and stores them in *eval*, using the method outlined above. On output, *B* contains its Cholesky decomposition and *A* is destroyed.

type **gsl\_eigen\_genhermv\_workspace**

This workspace contains internal parameters used for solving generalized hermitian eigenvalue and eigenvector problems.

```
gsl_eigen_genhermv_workspace *gsl_eigen_genhermv_alloc(const size_t n)
```

This function allocates a workspace for computing eigenvalues and eigenvectors of  $n$ -by- $n$  complex generalized hermitian-definite eigensystems. The size of the workspace is  $O(5n)$ .

```
void gsl_eigen_genhermv_free(gsl_eigen_genhermv_workspace *w)
```

This function frees the memory associated with the workspace *w*.

```
int gsl_eigen_genhermv(gsl_matrix_complex *A, gsl_matrix_complex *B, gsl_vector *eval, gsl_matrix_complex
                    *evec, gsl_eigen_genhermv_workspace *w)
```

This function computes the eigenvalues and eigenvectors of the complex generalized hermitian-definite matrix pair  $(A, B)$ , and stores them in *eval* and *evec* respectively. The computed eigenvectors are normalized to have unit magnitude. On output, *B* contains its Cholesky decomposition and *A* is destroyed.

## 15.6 Real Generalized Nonsymmetric Eigensystems

Given two square matrices ( $A$ ,  $B$ ), the generalized nonsymmetric eigenvalue problem is to find eigenvalues  $\lambda$  and eigenvectors  $x$  such that

$$Ax = \lambda Bx$$

We may also define the problem as finding eigenvalues  $\mu$  and eigenvectors  $y$  such that

$$\mu Ay = By$$

Note that these two problems are equivalent (with  $\lambda = 1/\mu$ ) if neither  $\lambda$  nor  $\mu$  is zero. If say,  $\lambda$  is zero, then it is still a well defined eigenproblem, but its alternate problem involving  $\mu$  is not. Therefore, to allow for zero (and infinite) eigenvalues, the problem which is actually solved is

$$\beta Ax = \alpha Bx$$

The eigensolver routines below will return two values  $\alpha$  and  $\beta$  and leave it to the user to perform the divisions  $\lambda = \alpha/\beta$  and  $\mu = \beta/\alpha$ .

If the determinant of the matrix pencil  $A - \lambda B$  is zero for all  $\lambda$ , the problem is said to be singular; otherwise it is called regular. Singularity normally leads to some  $\alpha = \beta = 0$  which means the eigenproblem is ill-conditioned and generally does not have well defined eigenvalue solutions. The routines below are intended for regular matrix pencils and could yield unpredictable results when applied to singular pencils.

The solution of the real generalized nonsymmetric eigensystem problem for a matrix pair ( $A$ ,  $B$ ) involves computing the generalized Schur decomposition

$$\begin{aligned} A &= QSZ^T \\ B &= QTZ^T \end{aligned}$$

where  $Q$  and  $Z$  are orthogonal matrices of left and right Schur vectors respectively, and  $(S, T)$  is the generalized Schur form whose diagonal elements give the  $\alpha$  and  $\beta$  values. The algorithm used is the QZ method due to Moler and Stewart (see references).

type **`gsl_eigen_gen_workspace`**

This workspace contains internal parameters used for solving generalized eigenvalue problems.

`gsl_eigen_gen_workspace *gsl_eigen_gen_alloc(const size_t n)`

This function allocates a workspace for computing eigenvalues of  $n$ -by- $n$  real generalized nonsymmetric eigensystems. The size of the workspace is  $O(n)$ .

void **`gsl_eigen_gen_free(gsl_eigen_gen_workspace *w)`**

This function frees the memory associated with the workspace  $w$ .

void **`gsl_eigen_gen_params(const int compute_s, const int compute_t, const int balance, gsl_eigen_gen_workspace *w)`**

This function sets some parameters which determine how the eigenvalue problem is solved in subsequent calls to `gsl_eigen_gen()`.

If `compute_s` is set to 1, the full Schur form  $S$  will be computed by `gsl_eigen_gen()`. If it is set to 0,  $S$  will not be computed (this is the default setting).  $S$  is a quasi upper triangular matrix with 1-by-1 and 2-by-2 blocks on its diagonal. 1-by-1 blocks correspond to real eigenvalues, and 2-by-2 blocks correspond to complex eigenvalues.

If `compute_t` is set to 1, the full Schur form  $T$  will be computed by `gsl_eigen_gen()`. If it is set to 0,  $T$  will not be computed (this is the default setting).  $T$  is an upper triangular matrix with non-negative elements on its diagonal. Any 2-by-2 blocks in  $S$  will correspond to a 2-by-2 diagonal block in  $T$ .

The `balance` parameter is currently ignored, since generalized balancing is not yet implemented.



```
int gsl_eigen_gen(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta,
                 gsl_eigen_gen_workspace *w)
```

This function computes the eigenvalues of the real generalized nonsymmetric matrix pair  $(A, B)$ , and stores them as pairs in  $(\alpha, \beta)$ , where  $\alpha$  is complex and  $\beta$  is real. If  $\beta_i$  is non-zero, then  $\lambda = \alpha_i/\beta_i$  is an eigenvalue. Likewise, if  $\alpha_i$  is non-zero, then  $\mu = \beta_i/\alpha_i$  is an eigenvalue of the alternate problem  $\mu Ay = By$ . The elements of  $\beta$  are normalized to be non-negative.

If  $S$  is desired, it is stored in  $A$  on output. If  $T$  is desired, it is stored in  $B$  on output. The ordering of eigenvalues in  $(\alpha, \beta)$  follows the ordering of the diagonal blocks in the Schur forms  $S$  and  $T$ . In rare cases, this function may fail to find all eigenvalues. If this occurs, an error code is returned.

```
int gsl_eigen_gen_QZ(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta, gsl_matrix
                    *Q, gsl_matrix *Z, gsl_eigen_gen_workspace *w)
```

This function is identical to `gsl_eigen_gen()` except that it also computes the left and right Schur vectors and stores them into  $Q$  and  $Z$  respectively.

```
type gsl_eigen_genv_workspace
```

This workspace contains internal parameters used for solving generalized eigenvalue and eigenvector problems.

```
gsl_eigen_genv_workspace *gsl_eigen_genv_alloc(const size_t n)
```

This function allocates a workspace for computing eigenvalues and eigenvectors of  $n$ -by- $n$  real generalized nonsymmetric eigensystems. The size of the workspace is  $O(7n)$ .

```
void gsl_eigen_genv_free(gsl_eigen_genv_workspace *w)
```

This function frees the memory associated with the workspace  $w$ .

```
int gsl_eigen_genv(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta,
                  gsl_matrix_complex *evec, gsl_eigen_genv_workspace *w)
```

This function computes eigenvalues and right eigenvectors of the  $n$ -by- $n$  real generalized nonsymmetric matrix pair  $(A, B)$ . The eigenvalues are stored in  $(\alpha, \beta)$  and the eigenvectors are stored in  $evec$ . It first calls `gsl_eigen_gen()` to compute the eigenvalues, Schur forms, and Schur vectors. Then it finds eigenvectors of the Schur forms and backtransforms them using the Schur vectors. The Schur vectors are destroyed in the process, but can be saved by using `gsl_eigen_genv_QZ()`. The computed eigenvectors are normalized to have unit magnitude. On output,  $(A, B)$  contains the generalized Schur form  $(S, T)$ . If `gsl_eigen_gen()` fails, no eigenvectors are computed, and an error code is returned.

```
int gsl_eigen_genv_QZ(gsl_matrix *A, gsl_matrix *B, gsl_vector_complex *alpha, gsl_vector *beta,
                    gsl_matrix_complex *evec, gsl_matrix *Q, gsl_matrix *Z, gsl_eigen_genv_workspace
                    *w)
```

This function is identical to `gsl_eigen_genv()` except that it also computes the left and right Schur vectors and stores them into  $Q$  and  $Z$  respectively.

## 15.7 Sorting Eigenvalues and Eigenvectors

```
int gsl_eigen_symmv_sort(gsl_vector *eval, gsl_matrix *evec, gsl_eigen_sort_t sort_type)
```

This function simultaneously sorts the eigenvalues stored in the vector  $eval$  and the corresponding real eigenvectors stored in the columns of the matrix  $evec$  into ascending or descending order according to the value of the parameter  $sort\_type$ ,

```
type gsl_eigen_sort_t
```

GSL_EIGEN_SORT_VAL_ASC	ascending order in numerical value
GSL_EIGEN_SORT_VAL_DESC	descending order in numerical value
GSL_EIGEN_SORT_ABS_ASC	ascending order in magnitude
GSL_EIGEN_SORT_ABS_DESC	descending order in magnitude

```
int gsl_eigen_hermv_sort(gsl_vector *eval, gsl_matrix_complex *evec, gsl_eigen_sort_t sort_type)
```

This function simultaneously sorts the eigenvalues stored in the vector *eval* and the corresponding complex eigenvectors stored in the columns of the matrix *evec* into ascending or descending order according to the value of the parameter *sort\_type* as shown above.

```
int gsl_eigen_nonsymmv_sort(gsl_vector_complex *eval, gsl_matrix_complex *evec, gsl_eigen_sort_t
                           sort_type)
```

This function simultaneously sorts the eigenvalues stored in the vector *eval* and the corresponding complex eigenvectors stored in the columns of the matrix *evec* into ascending or descending order according to the value of the parameter *sort\_type* as shown above. Only `GSL_EIGEN_SORT_ABS_ASC` and `GSL_EIGEN_SORT_ABS_DESC` are supported due to the eigenvalues being complex.

```
int gsl_eigen_gensymmv_sort(gsl_vector *eval, gsl_matrix *evec, gsl_eigen_sort_t sort_type)
```

This function simultaneously sorts the eigenvalues stored in the vector *eval* and the corresponding real eigenvectors stored in the columns of the matrix *evec* into ascending or descending order according to the value of the parameter *sort\_type* as shown above.

```
int gsl_eigen_genhermv_sort(gsl_vector *eval, gsl_matrix_complex *evec, gsl_eigen_sort_t sort_type)
```

This function simultaneously sorts the eigenvalues stored in the vector *eval* and the corresponding complex eigenvectors stored in the columns of the matrix *evec* into ascending or descending order according to the value of the parameter *sort\_type* as shown above.

```
int gsl_eigen_genmv_sort(gsl_vector_complex *alpha, gsl_vector *beta, gsl_matrix_complex *evec,
                        gsl_eigen_sort_t sort_type)
```

This function simultaneously sorts the eigenvalues stored in the vectors (*alpha*, *beta*) and the corresponding complex eigenvectors stored in the columns of the matrix *evec* into ascending or descending order according to the value of the parameter *sort\_type* as shown above. Only `GSL_EIGEN_SORT_ABS_ASC` and `GSL_EIGEN_SORT_ABS_DESC` are supported due to the eigenvalues being complex.

## 15.8 Examples

The following program computes the eigenvalues and eigenvectors of the 4-th order Hilbert matrix,  $H(i, j) = 1/(i + j + 1)$ .

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>

int
main (void)
{
  double data[] = { 1.0  , 1/2.0, 1/3.0, 1/4.0,
                   1/2.0, 1/3.0, 1/4.0, 1/5.0,
                   1/3.0, 1/4.0, 1/5.0, 1/6.0,
                   1/4.0, 1/5.0, 1/6.0, 1/7.0 };

  gsl_matrix_view m
    = gsl_matrix_view_array (data, 4, 4);

  gsl_vector *eval = gsl_vector_alloc (4);
  gsl_matrix *evec = gsl_matrix_alloc (4, 4);

  gsl_eigen_symmv_workspace * w =
    gsl_eigen_symmv_alloc (4);
```

(continues on next page)



(continued from previous page)

```

gsl_eigen_symmv (&m.matrix, eval, evec, w);
gsl_eigen_symmv_free (w);
gsl_eigen_symmv_sort (eval, evec,
                     GSL_EIGEN_SORT_ABS_ASC);

{
  int i;

  for (i = 0; i < 4; i++)
    {
      double eval_i
        = gsl_vector_get (eval, i);
      gsl_vector_view evec_i
        = gsl_matrix_column (evec, i);

      printf ("eigenvalue = %g\n", eval_i);
      printf ("eigenvector = \n");
      gsl_vector_fprintf (stdout,
                          &evec_i.vector, "%g");
    }
}

gsl_vector_free (eval);
gsl_matrix_free (evec);

return 0;
}

```

Here is the beginning of the output from the program:

```

$ ./a.out
eigenvalue = 9.67023e-05
eigenvector =
-0.0291933
0.328712
-0.791411
0.514553
...

```

This can be compared with the corresponding output from GNU octave:

```

octave> [v,d] = eig(hilb(4));
octave> diag(d)
ans =

 9.6702e-05
 6.7383e-03
 1.6914e-01
 1.5002e+00

```

(continues on next page)

(continued from previous page)

```

octave> v
v =

  0.029193  0.179186 -0.582076  0.792608
-0.328712 -0.741918  0.370502  0.451923
  0.791411  0.100228  0.509579  0.322416
-0.514553  0.638283  0.514048  0.252161

```

Note that the eigenvectors can differ by a change of sign, since the sign of an eigenvector is arbitrary.

The following program illustrates the use of the nonsymmetric eigensolver, by computing the eigenvalues and eigenvectors of the Vandermonde matrix  $V(x; i, j) = x_i^{n-j}$  with  $x = (-1, -2, 3, 4)$ .

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>

int
main (void)
{
  double data[] = { -1.0, 1.0, -1.0, 1.0,
                   -8.0, 4.0, -2.0, 1.0,
                   27.0, 9.0, 3.0, 1.0,
                   64.0, 16.0, 4.0, 1.0 };

  gsl_matrix_view m
    = gsl_matrix_view_array (data, 4, 4);

  gsl_vector_complex *eval = gsl_vector_complex_alloc (4);
  gsl_matrix_complex *evec = gsl_matrix_complex_alloc (4, 4);

  gsl_eigen_nonsymmv_workspace * w =
    gsl_eigen_nonsymmv_alloc (4);

  gsl_eigen_nonsymmv (&m.matrix, eval, evec, w);

  gsl_eigen_nonsymmv_free (w);

  gsl_eigen_nonsymmv_sort (eval, evec,
                          GSL_EIGEN_SORT_ABS_DESC);

  {
    int i, j;

    for (i = 0; i < 4; i++)
      {
        gsl_complex eval_i
          = gsl_vector_complex_get (eval, i);
        gsl_vector_complex_view evec_i
          = gsl_matrix_complex_column (evec, i);

        printf ("eigenvalue = %g + %gi\n",

```

(continues on next page)

(continued from previous page)

```

        GSL_REAL(eval_i), GSL_IMAG(eval_i));
printf ("eigenvector = \n");
for (j = 0; j < 4; ++j)
    {
        gsl_complex z =
            gsl_vector_complex_get(&vec_i.vector, j);
        printf("%g + %gi\n", GSL_REAL(z), GSL_IMAG(z));
    }
}

gsl_vector_complex_free(eval);
gsl_matrix_complex_free(vec);

return 0;
}

```

Here is the beginning of the output from the program:

```

$ ./a.out
eigenvalue = -6.41391 + 0i
eigenvector =
-0.0998822 + 0i
-0.111251 + 0i
0.292501 + 0i
0.944505 + 0i
eigenvalue = 5.54555 + 3.08545i
eigenvector =
-0.043487 + -0.0076308i
0.0642377 + -0.142127i
-0.515253 + 0.0405118i
-0.840592 + -0.00148565i
...

```

This can be compared with the corresponding output from GNU octave:

```

octave> [v,d] = eig(vander([-1 -2 3 4]));
octave> diag(d)
ans =

-6.4139 + 0.0000i
 5.5456 + 3.0854i
 5.5456 - 3.0854i
 2.3228 + 0.0000i

octave> v
v =

Columns 1 through 3:

-0.09988 + 0.00000i  -0.04350 - 0.00755i  -0.04350 + 0.00755i
-0.11125 + 0.00000i   0.06399 - 0.14224i   0.06399 + 0.14224i

```

(continues on next page)

(continued from previous page)

```
0.29250 + 0.00000i -0.51518 + 0.04142i -0.51518 - 0.04142i
0.94451 + 0.00000i -0.84059 + 0.00000i -0.84059 - 0.00000i
```

Column 4:

```
-0.14493 + 0.00000i
0.35660 + 0.00000i
0.91937 + 0.00000i
0.08118 + 0.00000i
```

Note that the eigenvectors corresponding to the eigenvalue  $5.54555 + 3.08545i$  differ by the multiplicative constant  $0.9999984 + 0.0017674i$  which is an arbitrary phase factor of magnitude 1.

## 15.9 References and Further Reading

Further information on the algorithms described in this section can be found in the following book,

- G. H. Golub, C. F. Van Loan, “Matrix Computations” (3rd Ed, 1996), Johns Hopkins University Press, ISBN 0-8018-5414-8.

Further information on the generalized eigensystems QZ algorithm can be found in this paper,

- C. Moler, G. Stewart, “An Algorithm for Generalized Matrix Eigenvalue Problems”, SIAM J. Numer. Anal., Vol 10, No 2, 1973.

Eigensystem routines for very large matrices can be found in the Fortran library LAPACK. The LAPACK library is described in,

- LAPACK Users’ Guide (Third Edition, 1999), Published by SIAM, ISBN 0-89871-447-8.

The LAPACK source code can be found at the website <http://www.netlib.org/lapack> along with an online copy of the users guide.

## FAST FOURIER TRANSFORMS (FFTS)

This chapter describes functions for performing Fast Fourier Transforms (FFTs). The library includes radix-2 routines (for lengths which are a power of two) and mixed-radix routines (which work for any length). For efficiency there are separate versions of the routines for real data and for complex data. The mixed-radix routines are a reimplementaion of the FFTPACK library of Paul Swarztrauber. Fortran code for FFTPACK is available on Netlib (FFTPACK also includes some routines for sine and cosine transforms but these are currently not available in GSL). For details and derivations of the underlying algorithms consult the document “GSL FFT Algorithms” (see *References and Further Reading*)

### 16.1 Mathematical Definitions

Fast Fourier Transforms are efficient algorithms for calculating the discrete Fourier transform (DFT),

$$x_j = \sum_{k=0}^{n-1} z_k \exp(-2\pi i j k / n)$$

The DFT usually arises as an approximation to the continuous Fourier transform when functions are sampled at discrete intervals in space or time. The naive evaluation of the discrete Fourier transform is a matrix-vector multiplication  $W\vec{z}$ . A general matrix-vector multiplication takes  $O(n^2)$  operations for  $n$  data-points. Fast Fourier transform algorithms use a divide-and-conquer strategy to factorize the matrix  $W$  into smaller sub-matrices, corresponding to the integer factors of the length  $n$ . If  $n$  can be factorized into a product of integers  $f_1 f_2 \dots f_m$  then the DFT can be computed in  $O(n \sum f_i)$  operations. For a radix-2 FFT this gives an operation count of  $O(n \log_2 n)$ .

All the FFT functions offer three types of transform: forwards, inverse and backwards, based on the same mathematical definitions. The definition of the *forward Fourier transform*,  $x = \text{FFT}(z)$ , is,

$$x_j = \sum_{k=0}^{n-1} z_k \exp(-2\pi i j k / n)$$

and the definition of the *inverse Fourier transform*,  $x = \text{IFFT}(z)$ , is,

$$z_j = \frac{1}{n} \sum_{k=0}^{n-1} x_k \exp(2\pi i j k / n).$$

The factor of  $1/n$  makes this a true inverse. For example, a call to `gsl_fft_complex_forward()` followed by a call to `gsl_fft_complex_inverse()` should return the original data (within numerical errors).

In general there are two possible choices for the sign of the exponential in the transform/ inverse-transform pair. GSL follows the same convention as FFTPACK, using a negative exponential for the forward transform. The advantage of this convention is that the inverse transform recreates the original function with simple Fourier synthesis. Numerical Recipes uses the opposite convention, a positive exponential in the forward transform.

The *backwards FFT* is simply our terminology for an unscaled version of the inverse FFT,

$$z_j^{\text{backwards}} = \sum_{k=0}^{n-1} x_k \exp(2\pi i j k / n)$$

When the overall scale of the result is unimportant it is often convenient to use the backwards FFT instead of the inverse to save unnecessary divisions.

## 16.2 Overview of complex data FFTs

The inputs and outputs for the complex FFT routines are *packed arrays* of floating point numbers. In a packed array the real and imaginary parts of each complex number are placed in alternate neighboring elements. For example, the following definition of a packed array of length 6:

```
double x[3*2];
gsl_complex_packed_array data = x;
```

can be used to hold an array of three complex numbers,  $z[3]$ , in the following way:

```
data[0] = Re(z[0])
data[1] = Im(z[0])
data[2] = Re(z[1])
data[3] = Im(z[1])
data[4] = Re(z[2])
data[5] = Im(z[2])
```

The array indices for the data have the same ordering as those in the definition of the DFT—i.e. there are no index transformations or permutations of the data.

A *stride* parameter allows the user to perform transforms on the elements  $z[\text{stride}*i]$  instead of  $z[i]$ . A stride greater than 1 can be used to take an in-place FFT of the column of a matrix. A stride of 1 accesses the array without any additional spacing between elements.

To perform an FFT on a vector argument, such as `gsl_vector_complex * v`, use the following definitions (or their equivalents) when calling the functions described in this chapter:

```
gsl_complex_packed_array data = v->data;
size_t stride = v->stride;
size_t n = v->size;
```

For physical applications it is important to remember that the index appearing in the DFT does not correspond directly to a physical frequency. If the time-step of the DFT is  $\Delta$  then the frequency-domain includes both positive and negative frequencies, ranging from  $-1/(2\Delta)$  through 0 to  $+1/(2\Delta)$ . The positive frequencies are stored from the beginning of the array up to the middle, and the negative frequencies are stored backwards from the end of the array.

Here is a table which shows the layout of the array data, and the correspondence between the time-domain data  $z$ , and the frequency-domain data  $x$ :

index	$z$	$x = \text{FFT}(z)$
0	$z(t = 0)$	$x(f = 0)$
1	$z(t = 1)$	$x(f = 1/(n \Delta))$
2	$z(t = 2)$	$x(f = 2/(n \Delta))$
.	.....	.....

(continues on next page)

(continued from previous page)

$n/2$	$z(t = n/2)$	$x(f = +1/(2 \Delta),$ $-1/(2 \Delta))$
.	.....	.....
$n-3$	$z(t = n-3)$	$x(f = -3/(n \Delta))$
$n-2$	$z(t = n-2)$	$x(f = -2/(n \Delta))$
$n-1$	$z(t = n-1)$	$x(f = -1/(n \Delta))$

When  $n$  is even the location  $n/2$  contains the most positive and negative frequencies ( $+1/(2\Delta)$ ,  $-1/(2\Delta)$ ) which are equivalent. If  $n$  is odd then general structure of the table above still applies, but  $n/2$  does not appear.

## 16.3 Radix-2 FFT routines for complex data

The radix-2 algorithms described in this section are simple and compact, although not necessarily the most efficient. They use the Cooley-Tukey algorithm to compute in-place complex FFTs for lengths which are a power of 2—no additional storage is required. The corresponding self-sorting mixed-radix routines offer better performance at the expense of requiring additional working space.

All the functions described in this section are declared in the header file `gsl_fft_complex.h`.

```
int gsl_fft_complex_radix2_forward(gsl_complex_packed_array data, size_t stride, size_t n)
```

```
int gsl_fft_complex_radix2_transform(gsl_complex_packed_array data, size_t stride, size_t n,  
                                     gsl_fft_direction sign)
```

```
int gsl_fft_complex_radix2_backward(gsl_complex_packed_array data, size_t stride, size_t n)
```

```
int gsl_fft_complex_radix2_inverse(gsl_complex_packed_array data, size_t stride, size_t n)
```

These functions compute forward, backward and inverse FFTs of length  $n$  with stride *stride*, on the packed complex array *data* using an in-place radix-2 decimation-in-time algorithm. The length of the transform  $n$  is restricted to powers of two. For the transform version of the function the *sign* argument can be either `forward` ( $-1$ ) or `backward` ( $+1$ ).

The functions return a value of `GSL_SUCCESS` if no errors were detected, or `GSL_EDOM` if the length of the data  $n$  is not a power of two.

```
int gsl_fft_complex_radix2_dif_forward(gsl_complex_packed_array data, size_t stride, size_t n)
```

```
int gsl_fft_complex_radix2_dif_transform(gsl_complex_packed_array data, size_t stride, size_t n,  
                                         gsl_fft_direction sign)
```

```
int gsl_fft_complex_radix2_dif_backward(gsl_complex_packed_array data, size_t stride, size_t n)
```

```
int gsl_fft_complex_radix2_dif_inverse(gsl_complex_packed_array data, size_t stride, size_t n)
```

These are decimation-in-frequency versions of the radix-2 FFT functions.

Here is an example program which computes the FFT of a short pulse in a sample of length 128. To make the resulting Fourier transform real the pulse is defined for equal positive and negative times ( $-10 \dots 10$ ), where the negative times wrap around the end of the array.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
```

(continues on next page)

```

#include <gsl/gsl_fft_complex.h>

#define REAL(z,i) ((z)[2*(i)])
#define IMAG(z,i) ((z)[2*(i)+1])

int
main (void)
{
  int i; double data[2*128];

  for (i = 0; i < 128; i++)
    {
      REAL(data,i) = 0.0; IMAG(data,i) = 0.0;
    }

  REAL(data,0) = 1.0;

  for (i = 1; i <= 10; i++)
    {
      REAL(data,i) = REAL(data,128-i) = 1.0;
    }

  for (i = 0; i < 128; i++)
    {
      printf ("%d %e %e\n", i,
              REAL(data,i), IMAG(data,i));
    }
  printf ("\n\n");

  gsl_fft_complex_radix2_forward (data, 1, 128);

  for (i = 0; i < 128; i++)
    {
      printf ("%d %e %e\n", i,
              REAL(data,i)/sqrt(128),
              IMAG(data,i)/sqrt(128));
    }

  return 0;
}

```

Note that we have assumed that the program is using the default error handler (which calls `abort()` for any errors). If you are not using a safe error handler you would need to check the return status of `gsl_fft_complex_radix2_forward()`.

The transformed data is rescaled by  $1/\sqrt{n}$  so that it fits on the same plot as the input. Only the real part is shown, by the choice of the input data the imaginary part is zero. Allowing for the wrap-around of negative times at  $t = 128$ , and working in units of  $k/n$ , the DFT approximates the continuum Fourier transform, giving a modulated sine function.

$$\int_{-a}^{+a} e^{-2\pi i k x} dx = \frac{\sin(2\pi k a)}{\pi k}$$

The output of the example program is plotted in Fig. 16.1.



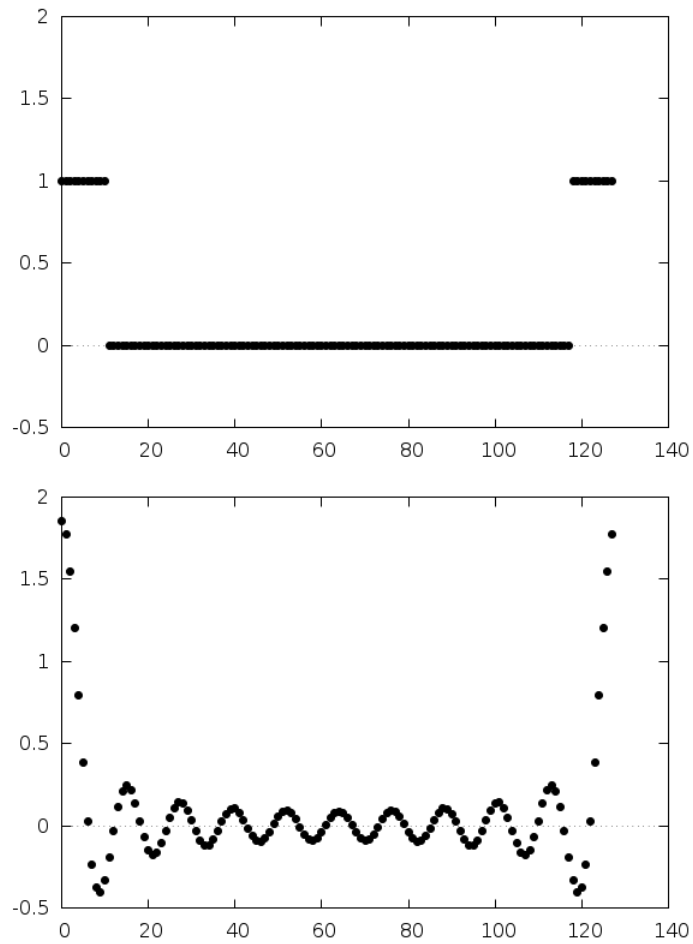


Fig. 16.1: A pulse and its discrete Fourier transform, output from the example program.

## 16.4 Mixed-radix FFT routines for complex data

This section describes mixed-radix FFT algorithms for complex data. The mixed-radix functions work for FFTs of any length. They are a reimplementation of Paul Swarztrauber’s Fortran FFTPACK library. The theory is explained in the review article “Self-sorting Mixed-radix FFTs” by Clive Temperton. The routines here use the same indexing scheme and basic algorithms as FFTPACK.

The mixed-radix algorithm is based on sub-transform modules—highly optimized small length FFTs which are combined to create larger FFTs. There are efficient modules for factors of 2, 3, 4, 5, 6 and 7. The modules for the composite factors of 4 and 6 are faster than combining the modules for  $2 * 2$  and  $2 * 3$ .

For factors which are not implemented as modules there is a fall-back to a general length- $n$  module which uses Singleton’s method for efficiently computing a DFT. This module is  $O(n^2)$ , and slower than a dedicated module would be but works for any length  $n$ . Of course, lengths which use the general length- $n$  module will still be factorized as much as possible. For example, a length of 143 will be factorized into  $11 * 13$ . Large prime factors are the worst case scenario, e.g. as found in  $n = 2 * 3 * 99991$ , and should be avoided because their  $O(n^2)$  scaling will dominate the run-time (consult the document “GSL FFT Algorithms” included in the GSL distribution if you encounter this problem).

The mixed-radix initialization function `gsl_fft_complex_wavetable_alloc()` returns the list of factors chosen by the library for a given length  $n$ . It can be used to check how well the length has been factorized, and estimate the run-time. To a first approximation the run-time scales as  $n \sum f_i$ , where the  $f_i$  are the factors of  $n$ . For programs under user control you may wish to issue a warning that the transform will be slow when the length is poorly factorized. If you frequently encounter data lengths which cannot be factorized using the existing small-prime modules consult “GSL FFT Algorithms” for details on adding support for other factors.

All the functions described in this section are declared in the header file `gsl_fft_complex.h`.

`gsl_fft_complex_wavetable *gsl_fft_complex_wavetable_alloc(size_t n)`

This function prepares a trigonometric lookup table for a complex FFT of length  $n$ . The function returns a pointer to the newly allocated `gsl_fft_complex_wavetable` if no errors were detected, and a null pointer in the case of error. The length  $n$  is factorized into a product of subtransforms, and the factors and their trigonometric coefficients are stored in the wavetable. The trigonometric coefficients are computed using direct calls to `sin` and `cos`, for accuracy. Recursion relations could be used to compute the lookup table faster, but if an application performs many FFTs of the same length then this computation is a one-off overhead which does not affect the final throughput.

The wavetable structure can be used repeatedly for any transform of the same length. The table is not modified by calls to any of the other FFT functions. The same wavetable can be used for both forward and backward (or inverse) transforms of a given length.

void `gsl_fft_complex_wavetable_free(gsl_fft_complex_wavetable *wavetable)`

This function frees the memory associated with the wavetable `wavetable`. The wavetable can be freed if no further FFTs of the same length will be needed.

These functions operate on a `gsl_fft_complex_wavetable` structure which contains internal parameters for the FFT. It is not necessary to set any of the components directly but it can sometimes be useful to examine them. For example, the chosen factorization of the FFT length is given and can be used to provide an estimate of the run-time or numerical error. The wavetable structure is declared in the header file `gsl_fft_complex.h`.

type `gsl_fft_complex_wavetable`

This is a structure that holds the factorization and trigonometric lookup tables for the mixed radix fft algorithm. It has the following components:

size_t n	This is the number of complex data points
size_t nf	This is the number of factors that the length n was decomposed into.
size_t factor[64]	This is the array of factors. Only the first nf elements are used.
gsl_complex * trig	This is a pointer to a preallocated trigonometric lookup table of n complex elements.
gsl_complex * twiddle[64]	This is an array of pointers into trig, giving the twiddle factors for each pass.

type **gsl\_fft\_complex\_workspace**

The mixed radix algorithms require additional working space to hold the intermediate steps of the transform.

*gsl\_fft\_complex\_workspace* \***gsl\_fft\_complex\_workspace\_alloc**(size\_t n)

This function allocates a workspace for a complex transform of length *n*.

void **gsl\_fft\_complex\_workspace\_free**(*gsl\_fft\_complex\_workspace* \*workspace)

This function frees the memory associated with the workspace *workspace*. The workspace can be freed if no further FFTs of the same length will be needed.

The following functions compute the transform,

int **gsl\_fft\_complex\_forward**(gsl\_complex\_packed\_array data, size\_t stride, size\_t n, const *gsl\_fft\_complex\_wavetable* \*wavetable, *gsl\_fft\_complex\_workspace* \*work)

int **gsl\_fft\_complex\_transform**(gsl\_complex\_packed\_array data, size\_t stride, size\_t n, const *gsl\_fft\_complex\_wavetable* \*wavetable, *gsl\_fft\_complex\_workspace* \*work, gsl\_fft\_direction sign)

int **gsl\_fft\_complex\_backward**(gsl\_complex\_packed\_array data, size\_t stride, size\_t n, const *gsl\_fft\_complex\_wavetable* \*wavetable, *gsl\_fft\_complex\_workspace* \*work)

int **gsl\_fft\_complex\_inverse**(gsl\_complex\_packed\_array data, size\_t stride, size\_t n, const *gsl\_fft\_complex\_wavetable* \*wavetable, *gsl\_fft\_complex\_workspace* \*work)

These functions compute forward, backward and inverse FFTs of length *n* with stride *stride*, on the packed complex array *data*, using a mixed radix decimation-in-frequency algorithm. There is no restriction on the length *n*. Efficient modules are provided for subtransforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow,  $O(n^2)$ , general-*n* module. The caller must supply a *wavetable* containing the trigonometric lookup tables and a workspace *work*. For the transform version of the function the *sign* argument can be either forward (-1) or backward (+1).

The functions return a value of 0 if no errors were detected. The following *gsl\_errno* conditions are defined for these functions:

<i>GSL_EDOM</i>	The length of the data <i>n</i> is not a positive integer (i.e. <i>n</i> is zero).
<i>GSL_EINVAL</i>	The length of the data <i>n</i> and the length used to compute the given <i>wavetable</i> do not match.

Here is an example program which computes the FFT of a short pulse in a sample of length 630 (= 2 \* 3 \* 3 \* 5 \* 7) using the mixed-radix algorithm.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>
```

(continues on next page)

(continued from previous page)

```

#define REAL(z,i) ((z)[2*(i)])
#define IMAG(z,i) ((z)[2*(i)+1])

int
main (void)
{
    int i;
    const int n = 630;
    double data[2*n];

    gsl_fft_complex_wavetable * wavetable;
    gsl_fft_complex_workspace * workspace;

    for (i = 0; i < n; i++)
    {
        REAL(data,i) = 0.0;
        IMAG(data,i) = 0.0;
    }

    data[0] = 1.0;

    for (i = 1; i <= 10; i++)
    {
        REAL(data,i) = REAL(data,n-i) = 1.0;
    }

    for (i = 0; i < n; i++)
    {
        printf ("%d: %e %e\n", i, REAL(data,i),
                IMAG(data,i));
    }
    printf ("\n");

    wavetable = gsl_fft_complex_wavetable_alloc (n);
    workspace = gsl_fft_complex_workspace_alloc (n);

    for (i = 0; i < (int) wavetable->nf; i++)
    {
        printf ("# factor %d: %zu\n", i,
                wavetable->factor[i]);
    }

    gsl_fft_complex_forward (data, 1, n,
                             wavetable, workspace);

    for (i = 0; i < n; i++)
    {
        printf ("%d: %e %e\n", i, REAL(data,i),
                IMAG(data,i));
    }

    gsl_fft_complex_wavetable_free (wavetable);

```

(continues on next page)

(continued from previous page)

```

gsl_fft_complex_workspace_free (workspace);
return 0;
}

```

Note that we have assumed that the program is using the default `gsl` error handler (which calls `abort()` for any errors). If you are not using a safe error handler you would need to check the return status of all the `gsl` routines.

## 16.5 Overview of real data FFTs

The functions for real data are similar to those for complex data. However, there is an important difference between forward and inverse transforms. The Fourier transform of a real sequence is not real. It is a complex sequence with a special symmetry:

$$z_k = z_{n-k}^*$$

A sequence with this symmetry is called *conjugate-complex* or *half-complex*. This different structure requires different storage layouts for the forward transform (from real to half-complex) and inverse transform (from half-complex back to real). As a consequence the routines are divided into two sets: functions in `gsl_fft_real` which operate on real sequences and functions in `gsl_fft_halfcomplex` which operate on half-complex sequences.

Functions in `gsl_fft_real` compute the frequency coefficients of a real sequence. The half-complex coefficients  $c$  of a real sequence  $x$  are given by Fourier analysis,

$$c_k = \sum_{j=0}^{n-1} x_j \exp(-2\pi i j k / n)$$

Functions in `gsl_fft_halfcomplex` compute inverse or backwards transforms. They reconstruct real sequences by Fourier synthesis from their half-complex frequency coefficients,  $c$ ,

$$x_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \exp(2\pi i j k / n)$$

The symmetry of the half-complex sequence implies that only half of the complex numbers in the output need to be stored. The remaining half can be reconstructed using the half-complex symmetry condition. This works for all lengths, even and odd—when the length is even the middle value where  $k = n/2$  is also real. Thus only  $n$  real numbers are required to store the half-complex sequence, and the transform of a real sequence can be stored in the same size array as the original data.

The precise storage arrangements depend on the algorithm, and are different for radix-2 and mixed-radix routines. The radix-2 function operates in-place, which constrains the locations where each element can be stored. The restriction forces real and imaginary parts to be stored far apart. The mixed-radix algorithm does not have this restriction, and it stores the real and imaginary parts of a given term in neighboring locations (which is desirable for better locality of memory accesses).

## 16.6 Radix-2 FFT routines for real data

This section describes radix-2 FFT algorithms for real data. They use the Cooley-Tukey algorithm to compute in-place FFTs for lengths which are a power of 2.

The radix-2 FFT functions for real data are declared in the header files `gsl_fft_real.h`

```
int gsl_fft_real_radix2_transform(double data[], size_t stride, size_t n)
```

This function computes an in-place radix-2 FFT of length  $n$  and stride `stride` on the real array `data`. The output is a half-complex sequence, which is stored in-place. The arrangement of the half-complex terms uses the following scheme: for  $k < n/2$  the real part of the  $k$ -th term is stored in location  $k$ , and the corresponding imaginary part is stored in location  $n - k$ . Terms with  $k > n/2$  can be reconstructed using the symmetry  $z_k = z_{n-k}^*$ . The terms for  $k = 0$  and  $k = n/2$  are both purely real, and count as a special case. Their real parts are stored in locations 0 and  $n/2$  respectively, while their imaginary parts which are zero are not stored.

The following table shows the correspondence between the output `data` and the equivalent results obtained by considering the input data as a complex sequence with zero imaginary part (assuming `stride = 1`):

<code>complex[0].real</code>	=	<code>data[0]</code>	
<code>complex[0].imag</code>	=	<code>0</code>	
<code>complex[1].real</code>	=	<code>data[1]</code>	
<code>complex[1].imag</code>	=	<code>data[n-1]</code>	
.....		.....	
<code>complex[k].real</code>	=	<code>data[k]</code>	
<code>complex[k].imag</code>	=	<code>data[n-k]</code>	
.....		.....	
<code>complex[n/2].real</code>	=	<code>data[n/2]</code>	
<code>complex[n/2].imag</code>	=	<code>0</code>	
.....		.....	
<code>complex[k'].real</code>	=	<code>data[k]</code>	<code>k' = n - k</code>
<code>complex[k'].imag</code>	=	<code>-data[n-k]</code>	
.....		.....	
<code>complex[n-1].real</code>	=	<code>data[1]</code>	
<code>complex[n-1].imag</code>	=	<code>-data[n-1]</code>	

Note that the output data can be converted into the full complex sequence using the function `gsl_fft_halfcomplex_radix2_unpack()` described below.

The radix-2 FFT functions for halfcomplex data are declared in the header file `gsl_fft_halfcomplex.h`.

```
int gsl_fft_halfcomplex_radix2_inverse(double data[], size_t stride, size_t n)
```

```
int gsl_fft_halfcomplex_radix2_backward(double data[], size_t stride, size_t n)
```

These functions compute the inverse or backwards in-place radix-2 FFT of length  $n$  and stride `stride` on the half-complex sequence `data` stored according to the output scheme used by `gsl_fft_real_radix2()`. The result is a real array stored in natural order.

```
int gsl_fft_halfcomplex_radix2_unpack(const double halfcomplex_coefficient[], gsl_complex_packed_array  
complex_coefficient, size_t stride, size_t n)
```

This function converts `halfcomplex_coefficient`, an array of half-complex coefficients as returned by `gsl_fft_real_radix2_transform()`, into an ordinary complex array, `complex_coefficient`. It fills in the complex array using the symmetry  $z_k = z_{n-k}^*$  to reconstruct the redundant elements. The algorithm for the conversion is:

<code>complex_coefficient[0].real</code>	=	<code>halfcomplex_coefficient[0];</code>
<code>complex_coefficient[0].imag</code>	=	<code>0.0;</code>

(continues on next page)

(continued from previous page)

```

for (i = 1; i < n - i; i++)
{
    double hc_real = halfcomplex_coefficient[i*stride];
    double hc_imag = halfcomplex_coefficient[(n-i)*stride];
    complex_coefficient[i*stride].real = hc_real;
    complex_coefficient[i*stride].imag = hc_imag;
    complex_coefficient[(n - i)*stride].real = hc_real;
    complex_coefficient[(n - i)*stride].imag = -hc_imag;
}

if (i == n - i)
{
    complex_coefficient[i*stride].real = halfcomplex_coefficient[(n - 1)*stride];
    complex_coefficient[i*stride].imag = 0.0;
}

```

## 16.7 Mixed-radix FFT routines for real data

This section describes mixed-radix FFT algorithms for real data. The mixed-radix functions work for FFTs of any length. They are a reimplementaion of the real-FFT routines in the Fortran FFTPACK library by Paul Swarztrauber. The theory behind the algorithm is explained in the article “Fast Mixed-Radix Real Fourier Transforms” by Clive Temperton. The routines here use the same indexing scheme and basic algorithms as FFTPACK.

The functions use the FFTPACK storage convention for half-complex sequences. In this convention the half-complex transform of a real sequence is stored with frequencies in increasing order, starting at zero, with the real and imaginary parts of each frequency in neighboring locations. When a value is known to be real the imaginary part is not stored. The imaginary part of the zero-frequency component is never stored. It is known to be zero (since the zero frequency component is simply the sum of the input data (all real)). For a sequence of even length the imaginary part of the frequency  $n/2$  is not stored either, since the symmetry  $z_k = z_{n-k}^*$  implies that this is purely real too.

The storage scheme is best shown by some examples. The table below shows the output for an odd-length sequence,  $n = 5$ . The two columns give the correspondence between the 5 values in the half-complex sequence returned by `gsl_fft_real_transform()`, `halfcomplex[]` and the values `complex[]` that would be returned if the same real input sequence were passed to `gsl_fft_complex_backward()` as a complex sequence (with imaginary parts set to 0):

```

complex[0].real = halfcomplex[0]
complex[0].imag = 0
complex[1].real = halfcomplex[1]
complex[1].imag = halfcomplex[2]
complex[2].real = halfcomplex[3]
complex[2].imag = halfcomplex[4]
complex[3].real = halfcomplex[3]
complex[3].imag = -halfcomplex[4]
complex[4].real = halfcomplex[1]
complex[4].imag = -halfcomplex[2]

```

The upper elements of the `complex` array, `complex[3]` and `complex[4]` are filled in using the symmetry condition. The imaginary part of the zero-frequency term `complex[0].imag` is known to be zero by the symmetry.

The next table shows the output for an even-length sequence,  $n = 6$ . In the even case there are two values which are

purely real:

```

complex[0].real = halfcomplex[0]
complex[0].imag = 0
complex[1].real = halfcomplex[1]
complex[1].imag = halfcomplex[2]
complex[2].real = halfcomplex[3]
complex[2].imag = halfcomplex[4]
complex[3].real = halfcomplex[5]
complex[3].imag = 0
complex[4].real = halfcomplex[3]
complex[4].imag = -halfcomplex[4]
complex[5].real = halfcomplex[1]
complex[5].imag = -halfcomplex[2]

```

The upper elements of the complex array, `complex[4]` and `complex[5]` are filled in using the symmetry condition. Both `complex[0].imag` and `complex[3].imag` are known to be zero.

All these functions are declared in the header files `gsl_fft_real.h` and `gsl_fft_halfcomplex.h`.

type **gsl\_fft\_real\_wavetable**

type **gsl\_fft\_halfcomplex\_wavetable**

These data structures contain lookup tables for an FFT of a fixed size.

*gsl\_fft\_real\_wavetable* \***gsl\_fft\_real\_wavetable\_alloc**(size\_t n)

*gsl\_fft\_halfcomplex\_wavetable* \***gsl\_fft\_halfcomplex\_wavetable\_alloc**(size\_t n)

These functions prepare trigonometric lookup tables for an FFT of size  $n$  real elements. The functions return a pointer to the newly allocated struct if no errors were detected, and a null pointer in the case of error. The length  $n$  is factorized into a product of subtransforms, and the factors and their trigonometric coefficients are stored in the wavetable. The trigonometric coefficients are computed using direct calls to `sin` and `cos`, for accuracy. Recursion relations could be used to compute the lookup table faster, but if an application performs many FFTs of the same length then computing the wavetable is a one-off overhead which does not affect the final throughput.

The wavetable structure can be used repeatedly for any transform of the same length. The table is not modified by calls to any of the other FFT functions. The appropriate type of wavetable must be used for forward real or inverse half-complex transforms.

void **gsl\_fft\_real\_wavetable\_free**(*gsl\_fft\_real\_wavetable* \*wavetable)

void **gsl\_fft\_halfcomplex\_wavetable\_free**(*gsl\_fft\_halfcomplex\_wavetable* \*wavetable)

These functions free the memory associated with the wavetable *wavetable*. The wavetable can be freed if no further FFTs of the same length will be needed.

The mixed radix algorithms require additional working space to hold the intermediate steps of the transform,

type **gsl\_fft\_real\_workspace**

This workspace contains parameters needed to compute a real FFT.

*gsl\_fft\_real\_workspace* \***gsl\_fft\_real\_workspace\_alloc**(size\_t n)

This function allocates a workspace for a real transform of length  $n$ . The same workspace can be used for both forward real and inverse halfcomplex transforms.

void **gsl\_fft\_real\_workspace\_free**(*gsl\_fft\_real\_workspace* \*workspace)

This function frees the memory associated with the workspace *workspace*. The workspace can be freed if no further FFTs of the same length will be needed.

The following functions compute the transforms of real and half-complex data,



```
int gsl_fft_real_transform(double data[], size_t stride, size_t n, const gsl_fft_real_wavetable *wavetable,
                           gsl_fft_real_workspace *work)
```

```
int gsl_fft_halfcomplex_transform(double data[], size_t stride, size_t n, const gsl_fft_halfcomplex_wavetable
                                   *wavetable, gsl_fft_real_workspace *work)
```

These functions compute the FFT of *data*, a real or half-complex array of length *n*, using a mixed radix decimation-in-frequency algorithm. For *gsl\_fft\_real\_transform()* *data* is an array of time-ordered real data. For *gsl\_fft\_halfcomplex\_transform()* *data* contains Fourier coefficients in the half-complex ordering described above. There is no restriction on the length *n*. Efficient modules are provided for subtransforms of length 2, 3, 4 and 5. Any remaining factors are computed with a slow,  $O(n^2)$ , general-*n* module. The caller must supply a *wavetable* containing trigonometric lookup tables and a workspace *work*.

```
int gsl_fft_real_unpack(const double real_coefficient[], gsl_complex_packed_array complex_coefficient, size_t
                        stride, size_t n)
```

This function converts a single real array, *real\_coefficient* into an equivalent complex array, *complex\_coefficient*, (with imaginary part set to zero), suitable for *gsl\_fft\_complex* routines. The algorithm for the conversion is simply:

```
for (i = 0; i < n; i++)
{
    complex_coefficient[i*stride].real = real_coefficient[i*stride];
    complex_coefficient[i*stride].imag = 0.0;
}
```

```
int gsl_fft_halfcomplex_unpack(const double halfcomplex_coefficient[], gsl_complex_packed_array
                               complex_coefficient, size_t stride, size_t n)
```

This function converts *halfcomplex\_coefficient*, an array of half-complex coefficients as returned by *gsl\_fft\_real\_transform()*, into an ordinary complex array, *complex\_coefficient*. It fills in the complex array using the symmetry  $z_k = z_{n-k}^*$  to reconstruct the redundant elements. The algorithm for the conversion is:

```
complex_coefficient[0].real = halfcomplex_coefficient[0];
complex_coefficient[0].imag = 0.0;

for (i = 1; i < n - i; i++)
{
    double hc_real = halfcomplex_coefficient[(2 * i - 1)*stride];
    double hc_imag = halfcomplex_coefficient[(2 * i)*stride];
    complex_coefficient[i*stride].real = hc_real;
    complex_coefficient[i*stride].imag = hc_imag;
    complex_coefficient[(n - i)*stride].real = hc_real;
    complex_coefficient[(n - i)*stride].imag = -hc_imag;
}

if (i == n - i)
{
    complex_coefficient[i*stride].real = halfcomplex_coefficient[(n - 1)*stride];
    complex_coefficient[i*stride].imag = 0.0;
}
```

Here is an example program using *gsl\_fft\_real\_transform()* and *gsl\_fft\_halfcomplex\_inverse()*. It generates a real signal in the shape of a square pulse. The pulse is Fourier transformed to frequency space, and all but the lowest ten frequency components are removed from the array of Fourier coefficients returned by *gsl\_fft\_real\_transform()*.

The remaining Fourier coefficients are transformed back to the time-domain, to give a filtered version of the square pulse. Since Fourier coefficients are stored using the half-complex symmetry both positive and negative frequencies are removed and the final filtered signal is also real.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_real.h>
#include <gsl/gsl_fft_halfcomplex.h>

int
main (void)
{
  int i, n = 100;
  double data[n];

  gsl_fft_real_wavetable * real;
  gsl_fft_halfcomplex_wavetable * hc;
  gsl_fft_real_workspace * work;

  for (i = 0; i < n; i++)
    {
      data[i] = 0.0;
    }

  for (i = n / 3; i < 2 * n / 3; i++)
    {
      data[i] = 1.0;
    }

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e\n", i, data[i]);
    }
  printf ("\n");

  work = gsl_fft_real_workspace_alloc (n);
  real = gsl_fft_real_wavetable_alloc (n);

  gsl_fft_real_transform (data, 1, n,
                          real, work);

  gsl_fft_real_wavetable_free (real);

  for (i = 11; i < n; i++)
    {
      data[i] = 0;
    }

  hc = gsl_fft_halfcomplex_wavetable_alloc (n);

  gsl_fft_halfcomplex_inverse (data, 1, n,
                               hc, work);
}
```

(continues on next page)

(continued from previous page)

```
gsl_fft_halfcomplex_wavetable_free (hc);  
  
for (i = 0; i < n; i++)  
{  
    printf ("%d: %e\n", i, data[i]);  
}  
  
gsl_fft_real_workspace_free (work);  
return 0;  
}
```

The program output is shown in Fig. 16.2.

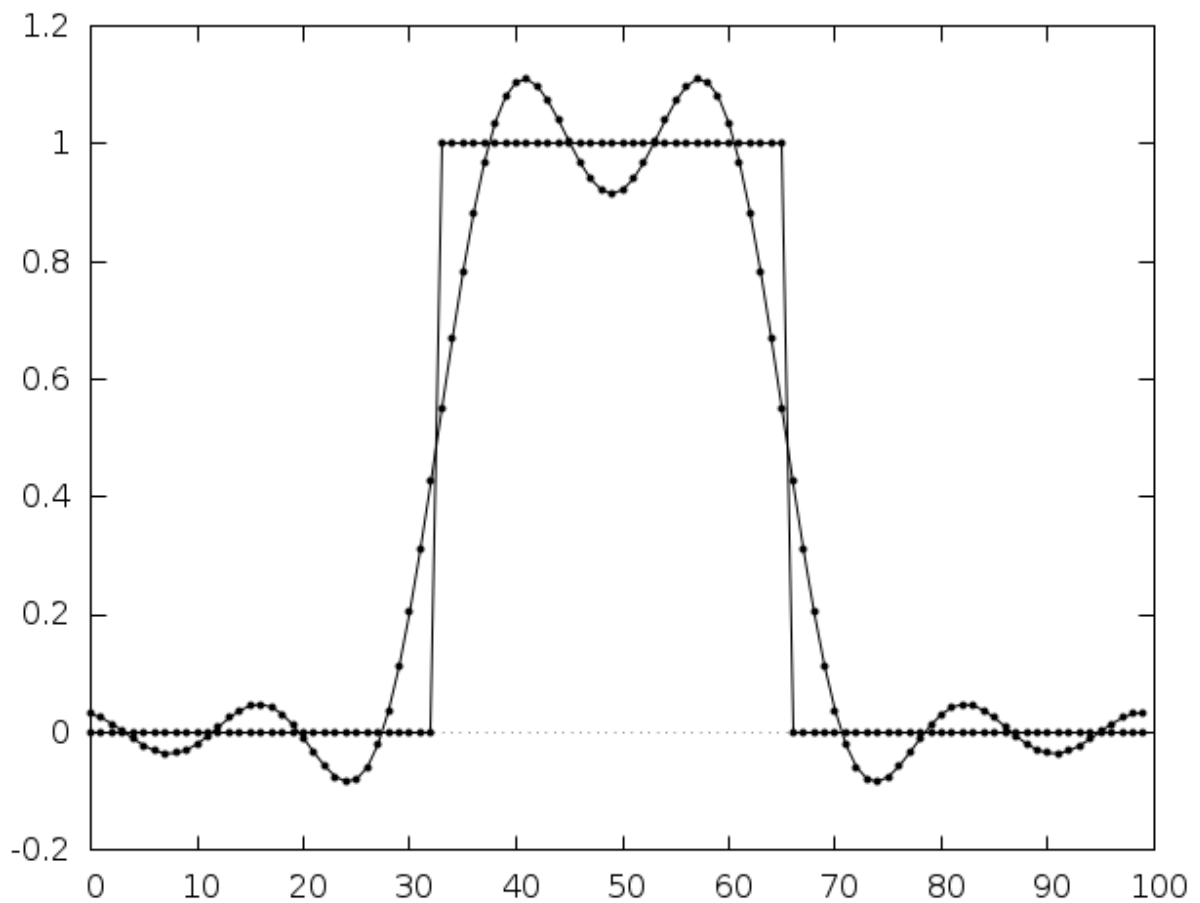


Fig. 16.2: Low-pass filtered version of a real pulse, output from the example program.

## 16.8 References and Further Reading

A good starting point for learning more about the FFT is the following review article,

- P. Duhamel and M. Vetterli. Fast Fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19:259–299, 1990.

To find out about the algorithms used in the GSL routines you may want to consult the document “GSL FFT Algorithms” (it is included in GSL, as `doc/fftalgorithms.tex`). This has general information on FFTs and explicit derivations of the implementation for each routine. There are also references to the relevant literature. For convenience some of the more important references are reproduced below.

There are several introductory books on the FFT with example programs, such as “The Fast Fourier Transform” by Brigham and “DFT/FFT and Convolution Algorithms” by Burrus and Parks,

- E. Oran Brigham. “The Fast Fourier Transform”. Prentice Hall, 1974.
- C. S. Burrus and T. W. Parks. “DFT/FFT and Convolution Algorithms”, Wiley, 1984.

Both these introductory books cover the radix-2 FFT in some detail. The mixed-radix algorithm at the heart of the FFTPACK routines is reviewed in Clive Temperton’s paper,

- Clive Temperton, Self-sorting mixed-radix fast Fourier transforms, *Journal of Computational Physics*, 52(1):1–23, 1983.

The derivation of FFTs for real-valued data is explained in the following two articles,

- Henrik V. Sorenson, Douglas L. Jones, Michael T. Heideman, and C. Sidney Burrus. Real-valued fast Fourier transform algorithms. “*IEEE Transactions on Acoustics, Speech, and Signal Processing*”, ASSP-35(6):849–863, 1987.
- Clive Temperton. Fast mixed-radix real Fourier transforms. “*Journal of Computational Physics*”, 52:340–350, 1983.

In 1979 the IEEE published a compendium of carefully-reviewed Fortran FFT programs in “Programs for Digital Signal Processing”. It is a useful reference for implementations of many different FFT algorithms,

- Digital Signal Processing Committee and IEEE Acoustics, Speech, and Signal Processing Committee, editors. *Programs for Digital Signal Processing*. IEEE Press, 1979.

For large-scale FFT work we recommend the use of the dedicated FFTW library by Frigo and Johnson. The FFTW library is self-optimizing—it automatically tunes itself for each hardware platform in order to achieve maximum performance. It is available under the GNU GPL.

- FFTW Website, <http://www.fftw.org/>

The source code for FFTPACK is available from <http://www.netlib.org/fftpack/>

## NUMERICAL INTEGRATION

This chapter describes routines for performing numerical integration (quadrature) of a function in one dimension. There are routines for adaptive and non-adaptive integration of general functions, with specialised routines for specific cases. These include integration over infinite and semi-infinite ranges, singular integrals, including logarithmic singularities, computation of Cauchy principal values and oscillatory integrals. The library reimplements the algorithms used in QUADPACK, a numerical integration package written by Piessens, de Doncker-Kapenga, Ueberhuber and Kahaner. Fortran code for QUADPACK is available on Netlib. Also included are non-adaptive, fixed-order Gauss-Legendre integration routines with high precision coefficients, as well as fixed-order quadrature rules for a variety of weighting functions from IQPACK.

The functions described in this chapter are declared in the header file `gsl_integration.h`.

### 17.1 Introduction

Each algorithm computes an approximation to a definite integral of the form,

$$I = \int_a^b f(x)w(x)dx$$

where  $w(x)$  is a weight function (for general integrands  $w(x) = 1$ ). The user provides absolute and relative error bounds ( $epsabs, epsrel$ ) which specify the following accuracy requirement,

$$|RESULT - I| \leq \max(epsabs, epsrel|I|)$$

where  $RESULT$  is the numerical approximation obtained by the algorithm. The algorithms attempt to estimate the absolute error  $ABSERR = |RESULT - I|$  in such a way that the following inequality holds,

$$|RESULT - I| \leq ABSERR \leq \max(epsabs, epsrel|I|)$$

In short, the routines return the first approximation which has an absolute error smaller than  $epsabs$  or a relative error smaller than  $epsrel$ .

Note that this is an *either-or* constraint, not simultaneous. To compute to a specified absolute error, set  $epsrel$  to zero. To compute to a specified relative error, set  $epsabs$  to zero. The routines will fail to converge if the error bounds are too stringent, but always return the best approximation obtained up to that stage.

The algorithms in QUADPACK use a naming convention based on the following letters:

Q - quadrature routine
N - non-adaptive integrator
A - adaptive integrator

(continues on next page)

G - general integrand (user-defined)
W - weight function <b>with</b> integrand
S - singularities can be more readily integrated
P - points of special difficulty can be supplied
I - infinite <b>range</b> of integration
O - oscillatory weight function, <b>cos</b> <b>or</b> <b>sin</b>
F - Fourier integral
C - Cauchy principal value

The algorithms are built on pairs of quadrature rules, a higher order rule and a lower order rule. The higher order rule is used to compute the best approximation to an integral over a small range. The difference between the results of the higher order rule and the lower order rule gives an estimate of the error in the approximation.

### 17.1.1 Integrands without weight functions

The algorithms for general functions (without a weight function) are based on Gauss-Kronrod rules.

A Gauss-Kronrod rule begins with a classical Gaussian quadrature rule of order  $m$ . This is extended with additional points between each of the abscissae to give a higher order Kronrod rule of order  $2m + 1$ . The Kronrod rule is efficient because it reuses existing function evaluations from the Gaussian rule.

The higher order Kronrod rule is used as the best approximation to the integral, and the difference between the two rules is used as an estimate of the error in the approximation.

### 17.1.2 Integrands with weight functions

For integrands with weight functions the algorithms use Clenshaw-Curtis quadrature rules.

A Clenshaw-Curtis rule begins with an  $n$ -th order Chebyshev polynomial approximation to the integrand. This polynomial can be integrated exactly to give an approximation to the integral of the original function. The Chebyshev expansion can be extended to higher orders to improve the approximation and provide an estimate of the error.

### 17.1.3 Integrands with singular weight functions

The presence of singularities (or other behavior) in the integrand can cause slow convergence in the Chebyshev approximation. The modified Clenshaw-Curtis rules used in QUADPACK separate out several common weight functions which cause slow convergence.

These weight functions are integrated analytically against the Chebyshev polynomials to precompute *modified Chebyshev moments*. Combining the moments with the Chebyshev approximation to the function gives the desired integral. The use of analytic integration for the singular part of the function allows exact cancellations and substantially improves the overall convergence behavior of the integration.

## 17.2 QNG non-adaptive Gauss-Kronrod integration

The QNG algorithm is a non-adaptive procedure which uses fixed Gauss-Kronrod-Patterson abscissae to sample the integrand at a maximum of 87 points. It is provided for fast integration of smooth functions.

```
int gsl_integration_qng(const gsl_function *f, double a, double b, double epsabs, double epsrel, double *result,
                      double *abserr, size_t *neval)
```

This function applies the Gauss-Kronrod 10-point, 21-point, 43-point and 87-point integration rules in succession until an estimate of the integral of  $f$  over  $(a, b)$  is achieved within the desired absolute and relative error limits, *epsabs* and *epsrel*. The function returns the final approximation, *result*, an estimate of the absolute error, *abserr* and the number of function evaluations used, *neval*. The Gauss-Kronrod rules are designed in such a way that each rule uses all the results of its predecessors, in order to minimize the total number of function evaluations.

## 17.3 QAG adaptive integration

The QAG algorithm is a simple adaptive integration procedure. The integration region is divided into subintervals, and on each iteration the subinterval with the largest estimated error is bisected. This reduces the overall error rapidly, as the subintervals become concentrated around local difficulties in the integrand. These subintervals are managed by the following struct,

```
type gsl_integration_workspace
```

This workspace handles the memory for the subinterval ranges, results and error estimates.

```
gsl_integration_workspace *gsl_integration_workspace_alloc(size_t n)
```

This function allocates a workspace sufficient to hold  $n$  double precision intervals, their integration results and error estimates. One workspace may be used multiple times as all necessary reinitialization is performed automatically by the integration routines.

```
void gsl_integration_workspace_free(gsl_integration_workspace *w)
```

This function frees the memory associated with the workspace  $w$ .

```
int gsl_integration_qag(const gsl_function *f, double a, double b, double epsabs, double epsrel, size_t limit, int
                      key, gsl_integration_workspace *workspace, double *result, double *abserr)
```

This function applies an integration rule adaptively until an estimate of the integral of  $f$  over  $(a, b)$  is achieved within the desired absolute and relative error limits, *epsabs* and *epsrel*. The function returns the final approximation, *result*, and an estimate of the absolute error, *abserr*. The integration rule is determined by the value of *key*, which should be chosen from the following symbolic names,

Symbolic Name	Key
GSL_INTEG_GAUSS15	1
GSL_INTEG_GAUSS21	2
GSL_INTEG_GAUSS31	3
GSL_INTEG_GAUSS41	4
GSL_INTEG_GAUSS51	5
GSL_INTEG_GAUSS61	6

corresponding to the 15, 21, 31, 41, 51 and 61 point Gauss-Kronrod rules. The higher-order rules give better accuracy for smooth functions, while lower-order rules save time when the function contains local difficulties, such as discontinuities.

On each iteration the adaptive integration strategy bisects the interval with the largest error estimate. The subintervals and their results are stored in the memory provided by *workspace*. The maximum number of subintervals is given by *limit*, which may not exceed the allocated size of the workspace.

## 17.4 QAGS adaptive integration with singularities

The presence of an integrable singularity in the integration region causes an adaptive routine to concentrate new subintervals around the singularity. As the subintervals decrease in size the successive approximations to the integral converge in a limiting fashion. This approach to the limit can be accelerated using an extrapolation procedure. The QAGS algorithm combines adaptive bisection with the Wynn epsilon-algorithm to speed up the integration of many types of integrable singularities.

```
int gsl_integration_qags(const gsl_function *f, double a, double b, double epsabs, double epsrel, size_t limit,
                        gsl_integration_workspace *workspace, double *result, double *abserr)
```

This function applies the Gauss-Kronrod 21-point integration rule adaptively until an estimate of the integral of  $f$  over  $(a, b)$  is achieved within the desired absolute and relative error limits, *epsabs* and *epsrel*. The results are extrapolated using the epsilon-algorithm, which accelerates the convergence of the integral in the presence of discontinuities and integrable singularities. The function returns the final approximation from the extrapolation, *result*, and an estimate of the absolute error, *abserr*. The subintervals and their results are stored in the memory provided by *workspace*. The maximum number of subintervals is given by *limit*, which may not exceed the allocated size of the workspace.

## 17.5 QAGP adaptive integration with known singular points

```
int gsl_integration_qagp(const gsl_function *f, double *pts, size_t npts, double epsabs, double epsrel, size_t
                        limit, gsl_integration_workspace *workspace, double *result, double *abserr)
```

This function applies the adaptive integration algorithm QAGS taking account of the user-supplied locations of singular points. The array *pts* of length *npts* should contain the endpoints of the integration ranges defined by the integration region and locations of the singularities. For example, to integrate over the region  $(a, b)$  with break-points at  $x_1, x_2, x_3$  (where  $a < x_1 < x_2 < x_3 < b$ ) the following *pts* array should be used:

```
pts[0] = a
pts[1] = x_1
pts[2] = x_2
pts[3] = x_3
pts[4] = b
```

with *npts* = 5.

If you know the locations of the singular points in the integration region then this routine will be faster than *gsl\_integration\_qags()*.

## 17.6 QAGI adaptive integration on infinite intervals

```
int gsl_integration_qagi(gsl_function *f, double epsabs, double epsrel, size_t limit, gsl_integration_workspace
                        *workspace, double *result, double *abserr)
```

This function computes the integral of the function  $f$  over the infinite interval  $(-\infty, +\infty)$ . The integral is mapped onto the semi-open interval  $(0, 1]$  using the transformation  $x = (1 - t)/t$ ,

$$\int_{-\infty}^{+\infty} dx f(x) = \int_0^1 dt (f((1-t)/t) + f(-(1-t)/t))/t^2.$$

It is then integrated using the QAGS algorithm. The normal 21-point Gauss-Kronrod rule of QAGS is replaced by a 15-point rule, because the transformation can generate an integrable singularity at the origin. In this case a lower-order rule is more efficient.



int **gsl\_integration\_qagi**(*gsl\_function* \*f, double a, double epsabs, double epsrel, size\_t limit, *gsl\_integration\_workspace* \*workspace, double \*result, double \*abserr)

This function computes the integral of the function  $f$  over the semi-infinite interval  $(a, +\infty)$ . The integral is mapped onto the semi-open interval  $(0, 1]$  using the transformation  $x = a + (1 - t)/t$ ,

$$\int_a^{+\infty} dx f(x) = \int_0^1 dt f(a + (1 - t)/t)/t^2$$

and then integrated using the QAGS algorithm.

int **gsl\_integration\_qagil**(*gsl\_function* \*f, double b, double epsabs, double epsrel, size\_t limit, *gsl\_integration\_workspace* \*workspace, double \*result, double \*abserr)

This function computes the integral of the function  $f$  over the semi-infinite interval  $(-\infty, b)$ . The integral is mapped onto the semi-open interval  $(0, 1]$  using the transformation  $x = b - (1 - t)/t$ ,

$$\int_{-\infty}^b dx f(x) = \int_0^1 dt f(b - (1 - t)/t)/t^2$$

and then integrated using the QAGS algorithm.

## 17.7 QAWC adaptive integration for Cauchy principal values

int **gsl\_integration\_qawc**(*gsl\_function* \*f, double a, double b, double c, double epsabs, double epsrel, size\_t limit, *gsl\_integration\_workspace* \*workspace, double \*result, double \*abserr)

This function computes the Cauchy principal value of the integral of  $f$  over  $(a, b)$ , with a singularity at  $c$ ,

$$I = \int_a^b dx \frac{f(x)}{x - c} = \lim_{\epsilon \rightarrow 0} \left\{ \int_a^{c-\epsilon} dx \frac{f(x)}{x - c} + \int_{c+\epsilon}^b dx \frac{f(x)}{x - c} \right\}$$

The adaptive bisection algorithm of QAG is used, with modifications to ensure that subdivisions do not occur at the singular point  $x = c$ . When a subinterval contains the point  $x = c$  or is close to it then a special 25-point modified Clenshaw-Curtis rule is used to control the singularity. Further away from the singularity the algorithm uses an ordinary 15-point Gauss-Kronrod integration rule.

## 17.8 QAWS adaptive integration for singular functions

The QAWS algorithm is designed for integrands with algebraic-logarithmic singularities at the end-points of an integration region. In order to work efficiently the algorithm requires a precomputed table of Chebyshev moments.

type **gsl\_integration\_qaws\_table**

This structure contains precomputed quantities for the QAWS algorithm.

*gsl\_integration\_qaws\_table* \***gsl\_integration\_qaws\_table\_alloc**(double alpha, double beta, int mu, int nu)

This function allocates space for a *gsl\_integration\_qaws\_table* struct describing a singular weight function  $w(x)$  with the parameters  $(\alpha, \beta, \mu, \nu)$ ,

$$w(x) = (x - a)^\alpha (b - x)^\beta \log^\mu(x - a) \log^\nu(b - x)$$

where  $\alpha > -1$ ,  $\beta > -1$ , and  $\mu = 0, 1$ ,  $\nu = 0, 1$ . The weight function can take four different forms depending on the values of  $\mu$  and  $\nu$ ,

Weight function $w(x)$	$(\mu, \nu)$
$(x - a)^\alpha (b - x)^\beta$	(0, 0)
$(x - a)^\alpha (b - x)^\beta \log(x - a)$	(1, 0)
$(x - a)^\alpha (b - x)^\beta \log(b - x)$	(0, 1)
$(x - a)^\alpha (b - x)^\beta \log(x - a) \log(b - x)$	(1, 1)

The singular points  $(a, b)$  do not have to be specified until the integral is computed, where they are the endpoints of the integration range.

The function returns a pointer to the newly allocated table `gsl_integration_qaws_table` if no errors were detected, and 0 in the case of error.

int `gsl_integration_qaws_table_set`(`gsl_integration_qaws_table` \*t, double alpha, double beta, int mu, int nu)

This function modifies the parameters  $(\alpha, \beta, \mu, \nu)$  of an existing `gsl_integration_qaws_table` struct `t`.

void `gsl_integration_qaws_table_free`(`gsl_integration_qaws_table` \*t)

This function frees all the memory associated with the `gsl_integration_qaws_table` struct `t`.

int `gsl_integration_qaws`(`gsl_function` \*f, const double a, const double b, `gsl_integration_qaws_table` \*t, const double epsabs, const double epsrel, const size\_t limit, `gsl_integration_workspace` \*workspace, double \*result, double \*abserr)

This function computes the integral of the function  $f(x)$  over the interval  $(a, b)$  with the singular weight function  $(x - a)^\alpha (b - x)^\beta \log^\mu(x - a) \log^\nu(b - x)$ . The parameters of the weight function  $(\alpha, \beta, \mu, \nu)$  are taken from the table `t`. The integral is,

$$I = \int_a^b dx f(x) (x - a)^\alpha (b - x)^\beta \log^\mu(x - a) \log^\nu(b - x).$$

The adaptive bisection algorithm of QAG is used. When a subinterval contains one of the endpoints then a special 25-point modified Clenshaw-Curtis rule is used to control the singularities. For subintervals which do not include the endpoints an ordinary 15-point Gauss-Kronrod integration rule is used.

## 17.9 QAWO adaptive integration for oscillatory functions

The QAWO algorithm is designed for integrands with an oscillatory factor,  $\sin(\omega x)$  or  $\cos(\omega x)$ . In order to work efficiently the algorithm requires a table of Chebyshev moments which must be pre-computed with calls to the functions below.

`gsl_integration_qawo_table` \*`gsl_integration_qawo_table_alloc`(double omega, double L, enum `gsl_integration_qawo_enum` sine, size\_t n)

This function allocates space for a `gsl_integration_qawo_table` struct and its associated workspace describing a sine or cosine weight function  $w(x)$  with the parameters  $(\omega, L)$ ,

$$w(x) = \begin{cases} \sin(\omega x) \\ \cos(\omega x) \end{cases}$$

The parameter  $L$  must be the length of the interval over which the function will be integrated  $L = b - a$ . The choice of sine or cosine is made with the parameter `sine` which should be chosen from one of the two following symbolic values:

**GSL\_INTEG\_COSINE**

**GSL\_INTEG\_SINE**

The `gsl_integration_qawo_table` is a table of the trigonometric coefficients required in the integration process. The parameter  $n$  determines the number of levels of coefficients that are computed. Each level corresponds

to one bisection of the interval  $L$ , so that  $n$  levels are sufficient for subintervals down to the length  $L/2^n$ . The integration routine `gsl_integration_qawo()` returns the error `GSL_ETABLE` if the number of levels is insufficient for the requested accuracy.

```
int gsl_integration_qawo_table_set(gsl_integration_qawo_table *t, double omega, double L, enum
                                gsl_integration_qawo_enum sine)
```

This function changes the parameters *omega*, *L* and *sine* of the existing workspace *t*.

```
int gsl_integration_qawo_table_set_length(gsl_integration_qawo_table *t, double L)
```

This function allows the length parameter *L* of the workspace *t* to be changed.

```
void gsl_integration_qawo_table_free(gsl_integration_qawo_table *t)
```

This function frees all the memory associated with the workspace *t*.

```
int gsl_integration_qawo(gsl_function *f, const double a, const double epsabs, const double epsrel, const size_t
                        limit, gsl_integration_workspace *workspace, gsl_integration_qawo_table *wf,
                        double *result, double *abserr)
```

This function uses an adaptive algorithm to compute the integral of  $f$  over  $(a, b)$  with the weight function  $\sin(\omega x)$  or  $\cos(\omega x)$  defined by the table *wf*,

$$I = \int_a^b dx f(x) \begin{Bmatrix} \sin(\omega x) \\ \cos(\omega x) \end{Bmatrix}$$

The results are extrapolated using the epsilon-algorithm to accelerate the convergence of the integral. The function returns the final approximation from the extrapolation, *result*, and an estimate of the absolute error, *abserr*. The subintervals and their results are stored in the memory provided by *workspace*. The maximum number of subintervals is given by *limit*, which may not exceed the allocated size of the workspace.

Those subintervals with “large” widths  $d$  where  $d\omega > 4$  are computed using a 25-point Clenshaw-Curtis integration rule, which handles the oscillatory behavior. Subintervals with a “small” widths where  $d\omega < 4$  are computed using a 15-point Gauss-Kronrod integration.

## 17.10 QAWF adaptive integration for Fourier integrals

```
int gsl_integration_qawf(gsl_function *f, const double a, const double epsabs, const size_t limit,
                       gsl_integration_workspace *workspace, gsl_integration_workspace
                       *cycle_workspace, gsl_integration_qawo_table *wf, double *result, double *abserr)
```

This function attempts to compute a Fourier integral of the function  $f$  over the semi-infinite interval  $[a, +\infty)$

$$I = \int_a^{+\infty} dx f(x) \begin{Bmatrix} \sin(\omega x) \\ \cos(\omega x) \end{Bmatrix}$$

The parameter  $\omega$  and choice of  $\sin$  or  $\cos$  is taken from the table *wf* (the length *L* can take any value, since it is overridden by this function to a value appropriate for the Fourier integration). The integral is computed using the QAWO algorithm over each of the subintervals,

$$\begin{aligned} C_1 &= [a, a + c] \\ C_2 &= [a + c, a + 2c] \\ &\dots = \dots \\ C_k &= [a + (k - 1)c, a + kc] \end{aligned}$$

where  $c = (2\text{floor}(|\omega|) + 1)\pi/|\omega|$ . The width  $c$  is chosen to cover an odd number of periods so that the contributions from the intervals alternate in sign and are monotonically decreasing when  $f$  is positive and monotonically decreasing. The sum of this sequence of contributions is accelerated using the epsilon-algorithm.

This function works to an overall absolute tolerance of *abserr*. The following strategy is used: on each interval  $C_k$  the algorithm tries to achieve the tolerance

$$TOL_k = u_k \text{abserr}$$

where  $u_k = (1 - p)p^{k-1}$  and  $p = 9/10$ . The sum of the geometric series of contributions from each interval gives an overall tolerance of *abserr*.

If the integration of a subinterval leads to difficulties then the accuracy requirement for subsequent intervals is relaxed,

$$TOL_k = u_k \max(\text{abserr}, \max_{i < k}(E_i))$$

where  $E_k$  is the estimated error on the interval  $C_k$ .

The subintervals and their results are stored in the memory provided by *workspace*. The maximum number of subintervals is given by *limit*, which may not exceed the allocated size of the workspace. The integration over each subinterval uses the memory provided by *cycle\_workspace* as workspace for the QAWO algorithm.

## 17.11 CQUAD doubly-adaptive integration

CQUAD is a new doubly-adaptive general-purpose quadrature routine which can handle most types of singularities, non-numerical function values such as Inf or NaN, as well as some divergent integrals. It generally requires more function evaluations than the integration routines in QUADPACK, yet fails less often for difficult integrands.

The underlying algorithm uses a doubly-adaptive scheme in which Clenshaw-Curtis quadrature rules of increasing degree are used to compute the integral in each interval. The  $L_2$ -norm of the difference between the underlying interpolatory polynomials of two successive rules is used as an error estimate. The interval is subdivided if the difference between two successive rules is too large or a rule of maximum degree has been reached.

`gsl_integration_cquad_workspace *gsl_integration_cquad_workspace_alloc(size_t n)`

This function allocates a workspace sufficient to hold the data for  $n$  intervals. The number  $n$  is not the maximum number of intervals that will be evaluated. If the workspace is full, intervals with smaller error estimates will be discarded. A minimum of 3 intervals is required and for most functions, a workspace of size 100 is sufficient.

`void gsl_integration_cquad_workspace_free(gsl_integration_cquad_workspace *w)`

This function frees the memory associated with the workspace  $w$ .

`int gsl_integration_cquad(const gsl_function *f, double a, double b, double epsabs, double epsrel, gsl_integration_cquad_workspace *workspace, double *result, double *abserr, size_t *nevals)`

This function computes the integral of  $f$  over  $(a, b)$  within the desired absolute and relative error limits, *epsabs* and *epsrel* using the CQUAD algorithm. The function returns the final approximation, *result*, an estimate of the absolute error, *abserr*, and the number of function evaluations required, *nevals*.

The CQUAD algorithm divides the integration region into subintervals, and in each iteration, the subinterval with the largest estimated error is processed. The algorithm uses Clenshaw-Curtis quadrature rules of degree 4, 8, 16 and 32 over 5, 9, 17 and 33 nodes respectively. Each interval is initialized with the lowest-degree rule. When an interval is processed, the next-higher degree rule is evaluated and an error estimate is computed based on the  $L_2$ -norm of the difference between the underlying interpolating polynomials of both rules. If the highest-degree rule has already been used, or the interpolatory polynomials differ significantly, the interval is bisected.

The subintervals and their results are stored in the memory provided by *workspace*. If the error estimate or the number of function evaluations is not needed, the pointers *abserr* and *nevals* can be set to NULL.

## 17.12 Romberg integration

The Romberg integration method estimates the definite integral

$$I = \int_a^b f(x)dx$$

by applying Richardson extrapolation on the trapezoidal rule, using equally spaced points with spacing

$$h_k = (b - a)2^{-k}$$

for  $k = 1, \dots, n$ . For each  $k$ , Richardson extrapolation is used  $k - 1$  times on previous approximations to improve the order of accuracy as much as possible. Romberg integration typically works well (and converges quickly) for smooth integrands with no singularities in the interval or at the end points.

`gsl_integration_romberg_workspace *gsl_integration_romberg_alloc`(const size\_t n)

This function allocates a workspace for Romberg integration, specifying a maximum of  $n$  iterations, or divisions of the interval. Since the number of divisions is  $2^n + 1$ ,  $n$  can be kept relatively small (i.e. 10 or 20). It is capped at a maximum value of 30 to prevent overflow. The size of the workspace is  $O(2n)$ .

void `gsl_integration_romberg_free`(gsl\_integration\_romberg\_workspace \*w)

This function frees the memory associated with the workspace  $w$ .

int `gsl_integration_romberg`(const *gsl\_function* \*f, const double a, const double b, const double epsabs, const double epsrel, double \*result, size\_t \*neval, gsl\_integration\_romberg\_workspace \*w)

This function integrates  $f(x)$ , specified by  $f$ , from  $a$  to  $b$ , storing the answer in *result*. At each step in the iteration, convergence is tested by checking:

$$|I_k - I_{k-1}| \leq \max(\text{epsabs}, \text{epsrel} \times |I_k|)$$

where  $I_k$  is the current approximation and  $I_{k-1}$  is the approximation of the previous iteration. If the method does not converge within the previously specified  $n$  iterations, the function stores the best current estimate in *result* and returns `GSL_EMAXITER`. If the method converges, the function returns `GSL_SUCCESS`. The total number of function evaluations is returned in *neval*.

## 17.13 Gauss-Legendre integration

The fixed-order Gauss-Legendre integration routines are provided for fast integration of smooth functions with known polynomial order. The  $n$ -point Gauss-Legendre rule is exact for polynomials of order  $2n - 1$  or less. For example, these rules are useful when integrating basis functions to form mass matrices for the Galerkin method. Unlike other numerical integration routines within the library, these routines do not accept absolute or relative error bounds.

`gsl_integration_glfixed_table *gsl_integration_glfixed_table_alloc`(size\_t n)

This function determines the Gauss-Legendre abscissae and weights necessary for an  $n$ -point fixed order integration scheme. If possible, high precision precomputed coefficients are used. If precomputed weights are not available, lower precision coefficients are computed on the fly.

double `gsl_integration_glfixed`(const *gsl\_function* \*f, double a, double b, const gsl\_integration\_glfixed\_table \*t)

This function applies the Gauss-Legendre integration rule contained in table  $t$  and returns the result.

int `gsl_integration_glfixed_point`(double a, double b, size\_t i, double \*xi, double \*wi, const gsl\_integration\_glfixed\_table \*t)

For  $i$  in  $[0, \dots, n - 1]$ , this function obtains the  $i$ -th Gauss-Legendre point  $xi$  and weight  $wi$  on the interval  $[a, b]$ . The points and weights are ordered by increasing point value. A function  $f$  may be integrated on  $[a, b]$  by summing  $wi * f(xi)$  over  $i$ .

void **gsl\_integration\_glfixed\_table\_free**(gsl\_integration\_glfixed\_table \*t)

This function frees the memory associated with the table *t*.

## 17.14 Fixed point quadratures

The routines in this section approximate an integral by the sum

$$\int_a^b w(x)f(x)dx = \sum_{i=1}^n w_i f(x_i)$$

where  $f(x)$  is the function to be integrated and  $w(x)$  is a weighting function. The  $n$  weights  $w_i$  and nodes  $x_i$  are carefully chosen so that the result is exact when  $f(x)$  is a polynomial of degree  $2n - 1$  or less. Once the user chooses the order  $n$  and weighting function  $w(x)$ , the weights  $w_i$  and nodes  $x_i$  can be precomputed and used to efficiently evaluate integrals for any number of functions  $f(x)$ .

This method works best when  $f(x)$  is well approximated by a polynomial on the interval  $(a, b)$ , and so is not suitable for functions with singularities. Since the user specifies ahead of time how many quadrature nodes will be used, these routines do not accept absolute or relative error bounds. The table below lists the weighting functions currently supported.

Name	Interval	Weighting function $w(x)$	Constraints
Legendre	$(a, b)$	1	$b > a$
Chebyshev Type 1	$(a, b)$	$1/\sqrt{(b-x)(x-a)}$	$b > a$
Gegenbauer	$(a, b)$	$((b-x)(x-a))^\alpha$	$\alpha > -1, b > a$
Jacobi	$(a, b)$	$(b-x)^\alpha(x-a)^\beta$	$\alpha, \beta > -1, b > a$
Laguerre	$(a, \infty)$	$(x-a)^\alpha \exp(-b(x-a))$	$\alpha > -1, b > 0$
Hermite	$(-\infty, \infty)$	$ x-a ^\alpha \exp(-b(x-a)^2)$	$\alpha > -1, b > 0$
Exponential	$(a, b)$	$ x - (a+b)/2 ^\alpha$	$\alpha > -1, b > a$
Rational	$(a, \infty)$	$(x-a)^\alpha(x+b)^\beta$	$\alpha > -1, \alpha + \beta + 2n < 0, a + b > 0$
Chebyshev Type 2	$(a, b)$	$\sqrt{(b-x)(x-a)}$	$b > a$

The fixed point quadrature routines use the following workspace to store the nodes and weights, as well as additional variables for intermediate calculations:

type **gsl\_integration\_fixed\_workspace**

This workspace is used for fixed point quadrature rules and looks like this:

```
typedef struct
{
    size_t n;          /* number of nodes/weights */
    double *weights;  /* quadrature weights */
    double *x;        /* quadrature nodes */
    double *diag;     /* diagonal of Jacobi matrix */
    double *subdiag;  /* subdiagonal of Jacobi matrix */
    const gsl_integration_fixed_type * type;
} gsl_integration_fixed_workspace;
```

*gsl\_integration\_fixed\_workspace* \***gsl\_integration\_fixed\_alloc**(const *gsl\_integration\_fixed\_type* \*T, const size\_t n, const double a, const double b, const double alpha, const double beta)

This function allocates a workspace for computing integrals with interpolating quadratures using  $n$  quadrature nodes. The parameters *a*, *b*, *alpha*, and *beta* specify the integration interval and/or weighting function for the

various quadrature types. See the *table* above for constraints on these parameters. The size of the workspace is  $O(4n)$ .

type **gsl\_integration\_fixed\_type**

The type of quadrature used is specified by *T* which can be set to the following choices:

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_legendre**

This specifies Legendre quadrature integration. The parameters *alpha* and *beta* are ignored for this type.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_chebyshev**

This specifies Chebyshev type 1 quadrature integration. The parameters *alpha* and *beta* are ignored for this type.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_gegenbauer**

This specifies Gegenbauer quadrature integration. The parameter *beta* is ignored for this type.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_jacobi**

This specifies Jacobi quadrature integration.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_laguerre**

This specifies Laguerre quadrature integration. The parameter *beta* is ignored for this type.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_hermite**

This specifies Hermite quadrature integration. The parameter *beta* is ignored for this type.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_exponential**

This specifies exponential quadrature integration. The parameter *beta* is ignored for this type.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_rational**

This specifies rational quadrature integration.

*gsl\_integration\_fixed\_type* \***gsl\_integration\_fixed\_chebyshev2**

This specifies Chebyshev type 2 quadrature integration. The parameters *alpha* and *beta* are ignored for this type.

void **gsl\_integration\_fixed\_free**(*gsl\_integration\_fixed\_workspace* \*w)

This function frees the memory associated with the workspace *w*

size\_t **gsl\_integration\_fixed\_n**(const *gsl\_integration\_fixed\_workspace* \*w)

This function returns the number of quadrature nodes and weights.

double \***gsl\_integration\_fixed\_nodes**(const *gsl\_integration\_fixed\_workspace* \*w)

This function returns a pointer to an array of size *n* containing the quadrature nodes  $x_i$ .

double \***gsl\_integration\_fixed\_weights**(const *gsl\_integration\_fixed\_workspace* \*w)

This function returns a pointer to an array of size *n* containing the quadrature weights  $w_i$ .

int **gsl\_integration\_fixed**(const *gsl\_function* \*func, double \*result, const *gsl\_integration\_fixed\_workspace* \*w)

This function integrates the function  $f(x)$  provided in *func* using previously computed fixed quadrature rules. The integral is approximated as

$$\sum_{i=1}^n w_i f(x_i)$$

where  $w_i$  are the quadrature weights and  $x_i$  are the quadrature nodes computed previously by *gsl\_integration\_fixed\_alloc()*. The sum is stored in *result* on output.



## 17.15 Error codes

In addition to the standard error codes for invalid arguments the functions can return the following values,

<code>GSL_EMAXITER</code>	the maximum number of subdivisions was exceeded.
<code>GSL_EROUND</code>	cannot reach tolerance because of roundoff error, or roundoff error was detected in the extrapolation table.
<code>GSL_ESING</code>	a non-integrable singularity or other bad integrand behavior was found in the integration interval.
<code>GSL_EDIVERGE</code>	the integral is divergent, or too slowly convergent to be integrated numerically.
<code>GSL_EDOM</code>	error in the values of the input arguments

## 17.16 Examples

### 17.16.1 Adaptive integration example

The integrator QAGS will handle a large class of definite integrals. For example, consider the following integral, which has an algebraic-logarithmic singularity at the origin,

$$\int_0^1 x^{-1/2} \log(x) dx = -4$$

The program below computes this integral to a relative accuracy bound of 1e-7.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void * params) {
    double alpha = *(double *) params;
    double f = log(alpha*x) / sqrt(x);
    return f;
}

int
main (void)
{
    gsl_integration_workspace * w
        = gsl_integration_workspace_alloc (1000);

    double result, error;
    double expected = -4.0;
    double alpha = 1.0;

    gsl_function F;
    F.function = &f;
    F.params = &alpha;

    gsl_integration_qags (&F, 0, 1, 0, 1e-7, 1000,
                        w, &result, &error);

    printf ("result      = % .18f\n", result);
```

(continues on next page)



(continued from previous page)

```

printf ("exact result   = % .18f\n", expected);
printf ("estimated error = % .18f\n", error);
printf ("actual error    = % .18f\n", result - expected);
printf ("intervals      = %zu\n", w->size);

gsl_integration_workspace_free (w);

return 0;
}

```

The results below show that the desired accuracy is achieved after 8 subdivisions.

```

result          = -4.00000000000000085265
exact result    = -4.00000000000000000000
estimated error =  0.000000000000000135447
actual error    = -0.00000000000000085265
intervals       = 8

```

In fact, the extrapolation procedure used by QAGS produces an accuracy of almost twice as many digits. The error estimate returned by the extrapolation procedure is larger than the actual error, giving a margin of safety of one order of magnitude.

## 17.16.2 Fixed-point quadrature example

In this example, we use a fixed-point quadrature rule to integrate the integral

$$\int_{-\infty}^{\infty} e^{-x^2} (x^m + 1) dx = \begin{cases} \sqrt{\pi} + \Gamma\left(\frac{m+1}{2}\right), & m \text{ even} \\ \sqrt{\pi}, & m \text{ odd} \end{cases}$$

for integer  $m$ . Consulting our [table](#) of fixed point quadratures, we see that this integral can be evaluated with a Hermite quadrature rule, setting  $\alpha = 0, a = 0, b = 1$ . Since we are integrating a polynomial of degree  $m$ , we need to choose the number of nodes  $n \geq (m + 1)/2$  to achieve the best results.

First we will try integrating for  $m = 10, n = 5$ , which does not satisfy our criteria above:

```
$ ./integration2 10 5
```

The output is,

```

m              = 10
intervals      = 5
result         = 47.468529694563351029
exact result   = 54.115231635459025483
actual error   = -6.646701940895674454

```

So, we find a large error. Now we try integrating for  $m = 10, n = 6$  which does satisfy the criteria above:

```
$ ./integration2 10 6
```

The output is,

```

m              = 10
intervals      = 6

```

(continues on next page)

(continued from previous page)

```

result          = 54.115231635459096537
exact result    = 54.115231635459025483
actual error    = 0.0000000000000071054

```

The program is given below.

```

#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_sf_gamma.h>

double
f(double x, void * params)
{
    int m = *(int *) params;
    double f = gsl_pow_int(x, m) + 1.0;
    return f;
}

int
main (int argc, char *argv[])
{
    gsl_integration_fixed_workspace * w;
    const gsl_integration_fixed_type * T = gsl_integration_fixed_hermite;
    int m = 10;
    int n = 6;
    double expected, result;
    gsl_function F;

    if (argc > 1)
        m = atoi(argv[1]);

    if (argc > 2)
        n = atoi(argv[2]);

    w = gsl_integration_fixed_alloc(T, n, 0.0, 1.0, 0.0, 0.0);

    F.function = &f;
    F.params = &m;

    gsl_integration_fixed(&F, &result, w);

    if (m % 2 == 0)
        expected = M_SQRTPI + gsl_sf_gamma(0.5*(1.0 + m));
    else
        expected = M_SQRTPI;

    printf ("m                = %d\n", m);
    printf ("intervals            = %zu\n", gsl_integration_fixed_n(w));
    printf ("result                = % .18f\n", result);
    printf ("exact result          = % .18f\n", expected);
    printf ("actual error          = % .18f\n", result - expected);
}

```

(continues on next page)

(continued from previous page)

```
gsl_integration_fixed_free (w);  
  
return 0;  
}
```

## 17.17 References and Further Reading

The following book is the definitive reference for QUADPACK, and was written by the original authors. It provides descriptions of the algorithms, program listings, test programs and examples. It also includes useful advice on numerical integration and many references to the numerical integration literature used in developing QUADPACK.

- R. Piessens, E. de Doncker-Kapenga, C.W. Ueberhuber, D.K. Kahaner. QUADPACK A subroutine package for automatic integration Springer Verlag, 1983.

The CQUAD integration algorithm is described in the following paper:

- P. Gonnet, “Increasing the Reliability of Adaptive Quadrature Using Explicit Interpolants”, ACM Transactions on Mathematical Software, Volume 37 (2010), Issue 3, Article 26.

The fixed-point quadrature routines are based on IQPACK, described in the following papers:

- S. Elhay, J. Kautsky, Algorithm 655: IQPACK, FORTRAN Subroutines for the Weights of Interpolatory Quadrature, ACM Transactions on Mathematical Software, Volume 13, Number 4, December 1987, pages 399-415.
- J. Kautsky, S. Elhay, Calculation of the Weights of Interpolatory Quadratures, Numerische Mathematik, Volume 40, Number 3, October 1982, pages 407-422.



## RANDOM NUMBER GENERATION

The library provides a large collection of random number generators which can be accessed through a uniform interface. Environment variables allow you to select different generators and seeds at runtime, so that you can easily switch between generators without needing to recompile your program. Each instance of a generator keeps track of its own state, allowing the generators to be used in multi-threaded programs. Additional functions are available for transforming uniform random numbers into samples from continuous or discrete probability distributions such as the Gaussian, log-normal or Poisson distributions.

These functions are declared in the header file `gs1_rng.h`.

### 18.1 General comments on random numbers

In 1988, Park and Miller wrote a paper entitled “Random number generators: good ones are hard to find.” [Commun.: ACM, 31, 1192–1201]. Fortunately, some excellent random number generators are available, though poor ones are still in common use. You may be happy with the system-supplied random number generator on your computer, but you should be aware that as computers get faster, requirements on random number generators increase. Nowadays, a simulation that calls a random number generator millions of times can often finish before you can make it down the hall to the coffee machine and back.

A very nice review of random number generators was written by Pierre L’Ecuyer, as Chapter 4 of the book: *Handbook on Simulation*, Jerry Banks, ed. (Wiley, 1997). The chapter is available in postscript from L’Ecuyer’s ftp site (see references). Knuth’s volume on *Seminumerical Algorithms* (originally published in 1968) devotes 170 pages to random number generators, and has recently been updated in its 3rd edition (1997). It is brilliant, a classic. If you don’t own it, you should stop reading right now, run to the nearest bookstore, and buy it.

A good random number generator will satisfy both theoretical and statistical properties. Theoretical properties are often hard to obtain (they require real math!), but one prefers a random number generator with a long period, low serial correlation, and a tendency **not** to “fall mainly on the planes.” Statistical tests are performed with numerical simulations. Generally, a random number generator is used to estimate some quantity for which the theory of probability provides an exact answer. Comparison to this exact answer provides a measure of “randomness”.

### 18.2 The Random Number Generator Interface

It is important to remember that a random number generator is not a “real” function like sine or cosine. Unlike real functions, successive calls to a random number generator yield different return values. Of course that is just what you want for a random number generator, but to achieve this effect, the generator must keep track of some kind of “state” variable. Sometimes this state is just an integer (sometimes just the value of the previously generated random number), but often it is more complicated than that and may involve a whole array of numbers, possibly with some indices thrown in. To use the random number generators, you do not need to know the details of what comprises the state, and besides that varies from algorithm to algorithm.

type **gsl\_rng\_type**

type **gsl\_rng**

The random number generator library uses two special structs, *gsl\_rng\_type* which holds static information about each type of generator and *gsl\_rng* which describes an instance of a generator created from a given *gsl\_rng\_type*.

The functions described in this section are declared in the header file `gsl_rng.h`.

## 18.3 Random number generator initialization

*gsl\_rng* \***gsl\_rng\_alloc**(const *gsl\_rng\_type* \*T)

This function returns a pointer to a newly-created instance of a random number generator of type *T*. For example, the following code creates an instance of the Tausworthe generator:

```
gsl_rng * r = gsl_rng_alloc (gsl_rng_taus);
```

If there is insufficient memory to create the generator then the function returns a null pointer and the error handler is invoked with an error code of *GSL\_ENOMEM*.

The generator is automatically initialized with the default seed, *gsl\_rng\_default\_seed*. This is zero by default but can be changed either directly or by using the environment variable *GSL\_RNG\_SEED*.

The details of the available generator types are described later in this chapter.

void **gsl\_rng\_set**(const *gsl\_rng* \*r, unsigned long int s)

This function initializes (or “seeds”) the random number generator. If the generator is seeded with the same value of *s* on two different runs, the same stream of random numbers will be generated by successive calls to the routines below. If different values of  $s \geq 1$  are supplied, then the generated streams of random numbers should be completely different. If the seed *s* is zero then the standard seed from the original implementation is used instead. For example, the original Fortran source code for the *ranlux* generator used a seed of 314159265, and so choosing *s* equal to zero reproduces this when using *gsl\_rng\_ranlux*.

When using multiple seeds with the same generator, choose seed values greater than zero to avoid collisions with the default setting.

Note that the most generators only accept 32-bit seeds, with higher values being reduced modulo  $2^{32}$ . For generators with smaller ranges the maximum seed value will typically be lower.

void **gsl\_rng\_free**(*gsl\_rng* \*r)

This function frees all the memory associated with the generator *r*.

## 18.4 Sampling from a random number generator

The following functions return uniformly distributed random numbers, either as integers or double precision floating point numbers. Inline versions of these functions are used when *HAVE\_INLINE* is defined. To obtain non-uniform distributions, see *Random Number Distributions*.

unsigned long int **gsl\_rng\_get**(const *gsl\_rng* \*r)

This function returns a random integer from the generator *r*. The minimum and maximum values depend on the algorithm used, but all integers in the range [min, max] are equally likely. The values of min and max can be determined using the auxiliary functions *gsl\_rng\_max()* and *gsl\_rng\_min()*.

double **gsl\_rng\_uniform**(const *gsl\_rng* \*r)

This function returns a double precision floating point number uniformly distributed in the range [0,1). The range includes 0.0 but excludes 1.0. The value is typically obtained by dividing the result of *gsl\_rng\_get(r)*

by `gsl_rng_max(r) + 1.0` in double precision. Some generators compute this ratio internally so that they can provide floating point numbers with more than 32 bits of randomness (the maximum number of bits that can be portably represented in a single unsigned long int).

double **gsl\_rng\_uniform\_pos**(const *gsl\_rng* \*r)

This function returns a positive double precision floating point number uniformly distributed in the range (0,1), excluding both 0.0 and 1.0. The number is obtained by sampling the generator with the algorithm of `gsl_rng_uniform()` until a non-zero value is obtained. You can use this function if you need to avoid a singularity at 0.0.

unsigned long int **gsl\_rng\_uniform\_int**(const *gsl\_rng* \*r, unsigned long int n)

This function returns a random integer from 0 to  $n - 1$  inclusive by scaling down and/or discarding samples from the generator *r*. All integers in the range  $[0, n - 1]$  are produced with equal probability. For generators with a non-zero minimum value an offset is applied so that zero is returned with the correct probability.

Note that this function is designed for sampling from ranges smaller than the range of the underlying generator. The parameter *n* must be less than or equal to the range of the generator *r*. If *n* is larger than the range of the generator then the function calls the error handler with an error code of `GSL_EINVAL` and returns zero.

In particular, this function is not intended for generating the full range of unsigned integer values  $[0, 2^{32} - 1]$ . Instead choose a generator with the maximal integer range and zero minimum value, such as `gsl_rng_ranlxd1`, `gsl_rng_mt19937` or `gsl_rng_taus`, and sample it directly using `gsl_rng_get()`. The range of each generator can be found using the auxiliary functions described in the next section.

## 18.5 Auxiliary random number generator functions

The following functions provide information about an existing generator. You should use them in preference to hard-coding the generator parameters into your own code.

const char \***gsl\_rng\_name**(const *gsl\_rng* \*r)

This function returns a pointer to the name of the generator. For example:

```
printf ("r is a '%s' generator\n", gsl_rng_name (r));
```

would print something like:

```
r is a 'taus' generator
```

unsigned long int **gsl\_rng\_max**(const *gsl\_rng* \*r)

This function returns the largest value that `gsl_rng_get()` can return.

unsigned long int **gsl\_rng\_min**(const *gsl\_rng* \*r)

This function returns the smallest value that `gsl_rng_get()` can return. Usually this value is zero. There are some generators with algorithms that cannot return zero, and for these generators the minimum value is 1.

void \***gsl\_rng\_state**(const *gsl\_rng* \*r)

size\_t **gsl\_rng\_size**(const *gsl\_rng* \*r)

These functions return a pointer to the state of generator *r* and its size. You can use this information to access the state directly. For example, the following code will write the state of a generator to a stream:

```
void * state = gsl_rng_state (r);
size_t n = gsl_rng_size (r);
fwrite (state, n, 1, stream);
```

const *gsl\_rng\_type* \*\***gsl\_rng\_types\_setup**(void)

This function returns a pointer to an array of all the available generator types, terminated by a null pointer. The function should be called once at the start of the program, if needed. The following code fragment shows how to iterate over the array of generator types to print the names of the available algorithms:

```
const gsl_rng_type **t, **t0;

t0 = gsl_rng_types_setup ();

printf ("Available generators:\n");

for (t = t0; *t != 0; t++)
{
    printf ("%s\n", (*t)->name);
}
```

## 18.6 Random number environment variables

The library allows you to choose a default generator and seed from the environment variables *GSL\_RNG\_TYPE* and *GSL\_RNG\_SEED* and the function *gsl\_rng\_env\_setup*( ). This makes it easy try out different generators and seeds without having to recompile your program.

### **GSL\_RNG\_TYPE**

This environment variable specifies the default random number generator. It should be the name of a generator, such as *taus* or *mt19937*.

### **GSL\_RNG\_SEED**

This environment variable specifies the default seed for the random number generator

### *gsl\_rng\_type* \***gsl\_rng\_default**

This global library variable specifies the default random number generator, and can be initialized from *GSL\_RNG\_TYPE* using *gsl\_rng\_env\_setup*( ). It is defined as follows:

```
extern const gsl_rng_type *gsl_rng_default
```

### unsigned long int **gsl\_rng\_default\_seed**

This global library variable specifies the seed for the default random number generator, and can be initialized from *GSL\_RNG\_SEED* using *gsl\_rng\_env\_setup*( ). It is set to zero by default and is defined as follows:

```
extern unsigned long int gsl_rng_default_seed
```

const *gsl\_rng\_type* \***gsl\_rng\_env\_setup**(void)

This function reads the environment variables *GSL\_RNG\_TYPE* and *GSL\_RNG\_SEED* and uses their values to set the corresponding library variables *gsl\_rng\_default* and *gsl\_rng\_default\_seed*.

The value of *GSL\_RNG\_SEED* is converted to an unsigned long int using the C library function *strtoul*( ).

If you don't specify a generator for *GSL\_RNG\_TYPE* then *gsl\_rng\_mt19937* is used as the default. The initial value of *gsl\_rng\_default\_seed* is zero.

Here is a short program which shows how to create a global generator using the environment variables *GSL\_RNG\_TYPE* and *GSL\_RNG\_SEED*,

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
```

(continues on next page)



(continued from previous page)

```

gsl_rng * r; /* global generator */

int
main (void)
{
    const gsl_rng_type * T;

    gsl_rng_env_setup();

    T = gsl_rng_default;
    r = gsl_rng_alloc (T);

    printf ("generator type: %s\n", gsl_rng_name (r));
    printf ("seed = %lu\n", gsl_rng_default_seed);
    printf ("first value = %lu\n", gsl_rng_get (r));

    gsl_rng_free (r);
    return 0;
}

```

Running the program without any environment variables uses the initial defaults, an mt19937 generator with a seed of 0,

```

generator type: mt19937
seed = 0
first value = 4293858116

```

By setting the two variables on the command line we can change the default generator and the seed:

```

$ GSL_RNG_TYPE="taus" GSL_RNG_SEED=123 ./a.out
GSL_RNG_TYPE=taus
GSL_RNG_SEED=123
generator type: taus
seed = 123
first value = 2720986350

```

## 18.7 Copying random number generator state

The above methods do not expose the random number state which changes from call to call. It is often useful to be able to save and restore the state. To permit these practices, a few somewhat more advanced functions are supplied. These include:

`int gsl_rng_memcpy(gsl_rng *dest, const gsl_rng *src)`

This function copies the random number generator *src* into the pre-existing generator *dest*, making *dest* into an exact copy of *src*. The two generators must be of the same type.

`gsl_rng *gsl_rng_clone(const gsl_rng *r)`

This function returns a pointer to a newly created generator which is an exact copy of the generator *r*.

## 18.8 Reading and writing random number generator state

The library provides functions for reading and writing the random number state to a file as binary data.

int **gsl\_rng\_fwrite**(FILE \*stream, const *gsl\_rng* \*r)

This function writes the random number state of the random number generator *r* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl\_rng\_fread**(FILE \*stream, *gsl\_rng* \*r)

This function reads the random number state into the random number generator *r* from the open stream *stream* in binary format. The random number generator *r* must be preinitialized with the correct random number generator type since type information is not saved. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

## 18.9 Random number generator algorithms

The functions described above make no reference to the actual algorithm used. This is deliberate so that you can switch algorithms without having to change any of your application source code. The library provides a large number of generators of different types, including simulation quality generators, generators provided for compatibility with other libraries and historical generators from the past.

The following generators are recommended for use in simulation. They have extremely long periods, low correlation and pass most statistical tests. For the most reliable source of uncorrelated numbers, the second-generation RANLUX generators have the strongest proof of randomness.

*gsl\_rng\_type* \***gsl\_rng\_mt19937**

The MT19937 generator of Makoto Matsumoto and Takuji Nishimura is a variant of the twisted generalized feedback shift-register algorithm, and is known as the “Mersenne Twister” generator. It has a Mersenne prime period of  $2^{19937} - 1$  (about  $10^{6000}$ ) and is equi-distributed in 623 dimensions. It has passed the DIEHARD statistical tests. It uses 624 words of state per generator and is comparable in speed to the other generators. The original generator used a default seed of 4357 and choosing *s* equal to zero in `gsl_rng_set()` reproduces this. Later versions switched to 5489 as the default seed, you can choose this explicitly via `gsl_rng_set()` instead if you require it.

For more information see,

- Makoto Matsumoto and Takuji Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”. ACM Transactions on Modeling and Computer Simulation, Vol.: 8, No.: 1 (Jan. 1998), Pages 3–30

The generator `gsl_rng_mt19937` uses the second revision of the seeding procedure published by the two authors above in 2002. The original seeding procedures could cause spurious artifacts for some seed values. They are still available through the alternative generators `gsl_rng_mt19937_1999` and `gsl_rng_mt19937_1998`.

*gsl\_rng\_type* \***gsl\_rng\_ranlxs0**

*gsl\_rng\_type* \***gsl\_rng\_ranlxs1**

*gsl\_rng\_type* \***gsl\_rng\_ranlxs2**

The generator `ranlxs0` is a second-generation version of the RANLUX algorithm of Luscher, which produces “luxury random numbers”. This generator provides single precision output (24 bits) at three luxury levels `ranlxs0`, `ranlxs1` and `ranlxs2`, in increasing order of strength. It uses double-precision floating point arithmetic internally and can be significantly faster than the integer version of `ranlux`, particularly on 64-bit architectures. The period of the generator is about  $10^{171}$ . The algorithm has mathematically proven properties and can

provide truly decorrelated numbers at a known level of randomness. The higher luxury levels provide increased decorrelation between samples as an additional safety margin.

Note that the range of allowed seeds for this generator is  $[0, 2^{31} - 1]$ . Higher seed values are wrapped modulo  $2^{31}$ .

*gsl\_rng\_type* \***gsl\_rng\_ranlxd1**

*gsl\_rng\_type* \***gsl\_rng\_ranlxd2**

These generators produce double precision output (48 bits) from the RANLXS generator. The library provides two luxury levels `ranlxd1` and `ranlxd2`, in increasing order of strength.

*gsl\_rng\_type* \***gsl\_rng\_ranlux**

*gsl\_rng\_type* \***gsl\_rng\_ranlux389**

The `ranlux` generator is an implementation of the original algorithm developed by Luscher. It uses a lagged-fibonacci-with-skipping algorithm to produce “luxury random numbers”. It is a 24-bit generator, originally designed for single-precision IEEE floating point numbers. This implementation is based on integer arithmetic, while the second-generation versions RANLXS and RANLXD described above provide floating-point implementations which will be faster on many platforms. The period of the generator is about  $10^{171}$ . The algorithm has mathematically proven properties and it can provide truly decorrelated numbers at a known level of randomness. The default level of decorrelation recommended by Luscher is provided by *gsl\_rng\_ranlux*, while *gsl\_rng\_ranlux389* gives the highest level of randomness, with all 24 bits decorrelated. Both types of generator use 24 words of state per generator.

For more information see,

- M. Luscher, “A portable high-quality random number generator for lattice field theory calculations”, *Computer Physics Communications*, 79 (1994) 100–110.
- F. James, “RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Luscher”, *Computer Physics Communications*, 79 (1994) 111–114

*gsl\_rng\_type* \***gsl\_rng\_cmrng**

This is a combined multiple recursive generator by L’Ecuyer. Its sequence is,

$$z_n = (x_n - y_n) \pmod{m_1}$$

where the two underlying generators  $x_n$  and  $y_n$  are,

$$\begin{aligned} x_n &= (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \pmod{m_1} \\ y_n &= (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \pmod{m_2} \end{aligned}$$

with coefficients  $a_1 = 0$ ,  $a_2 = 63308$ ,  $a_3 = -183326$ ,  $b_1 = 86098$ ,  $b_2 = 0$ ,  $b_3 = -539608$ , and moduli  $m_1 = 2^{31} - 1 = 2147483647$  and  $m_2 = 2145483479$ .

The period of this generator is  $\text{lcm}(m_1^3 - 1, m_2^3 - 1)$ , which is approximately  $2^{185}$  (about  $10^{56}$ ). It uses 6 words of state per generator. For more information see,

- P. L’Ecuyer, “Combined Multiple Recursive Random Number Generators”, *Operations Research*, 44, 5 (1996), 816–822.

*gsl\_rng\_type* \***gsl\_rng\_mrg**

This is a fifth-order multiple recursive generator by L’Ecuyer, Blouin and Coutre. Its sequence is,

$$x_n = (a_1 x_{n-1} + a_5 x_{n-5}) \pmod{m}$$

with  $a_1 = 107374182$ ,  $a_2 = a_3 = a_4 = 0$ ,  $a_5 = 104480$  and  $m = 2^{31} - 1$ .

The period of this generator is about  $10^{46}$ . It uses 5 words of state per generator. More information can be found in the following paper,

- P. L'Ecuyer, F. Blouin, and R. Coutre, "A search for good multiple recursive random number generators", ACM Transactions on Modeling and Computer Simulation 3, 87–98 (1993).

`gsl_rng_type *gsl_rng_taus`

`gsl_rng_type *gsl_rng_taus2`

This is a maximally equidistributed combined Tausworthe generator by L'Ecuyer. The sequence is,

$$x_n = (s_n^1 \oplus s_n^2 \oplus s_n^3)$$

where,

$$s_{n+1}^1 = (((s_n^1 \& 4294967294) \ll 12) \oplus (((s_n^1 \ll 13) \oplus s_n^1) \gg 19))$$

$$s_{n+1}^2 = (((s_n^2 \& 4294967288) \ll 4) \oplus (((s_n^2 \ll 2) \oplus s_n^2) \gg 25))$$

$$s_{n+1}^3 = (((s_n^3 \& 4294967280) \ll 17) \oplus (((s_n^3 \ll 3) \oplus s_n^3) \gg 11))$$

computed modulo  $2^{32}$ . In the formulas above  $\oplus$  denotes *exclusive-or*. Note that the algorithm relies on the properties of 32-bit unsigned integers and has been implemented using a bitmask of `0xFFFFFFFF` to make it work on 64 bit machines.

The period of this generator is  $2^{88}$  (about  $10^{26}$ ). It uses 3 words of state per generator. For more information see,

- P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators", Mathematics of Computation, 65, 213 (1996), 203–213.

The generator `gsl_rng_taus2` uses the same algorithm as `gsl_rng_taus` but with an improved seeding procedure described in the paper,

- P. L'Ecuyer, "Tables of Maximally Equidistributed Combined LFSR Generators", Mathematics of Computation, 68, 225 (1999), 261–269

The generator `gsl_rng_taus2` should now be used in preference to `gsl_rng_taus`.

`gsl_rng_type *gsl_rng_gfsr4`

The `gfsr4` generator is like a lagged-fibonacci generator, and produces each number as an xor'd sum of four previous values.

$$r_n = r_{n-A} \oplus r_{n-B} \oplus r_{n-C} \oplus r_{n-D}$$

Ziff (ref below) notes that "it is now widely known" that two-tap registers (such as R250, which is described below) have serious flaws, the most obvious one being the three-point correlation that comes from the definition of the generator. Nice mathematical properties can be derived for GFSR's, and numerics bears out the claim that 4-tap GFSR's with appropriately chosen offsets are as random as can be measured, using the author's test.

This implementation uses the values suggested the example on p392 of Ziff's article:  $A = 471$ ,  $B = 1586$ ,  $C = 6988$ ,  $D = 9689$ .

If the offsets are appropriately chosen (such as the one ones in this implementation), then the sequence is said to be maximal; that means that the period is  $2^D - 1$ , where  $D$  is the longest lag. (It is one less than  $2^D$  because it is not permitted to have all zeros in the `ra[]` array.) For this implementation with  $D = 9689$  that works out to about  $10^{2917}$ .

Note that the implementation of this generator using a 32-bit integer amounts to 32 parallel implementations of one-bit generators. One consequence of this is that the period of this 32-bit generator is the same as for the one-bit generator. Moreover, this independence means that all 32-bit patterns are equally likely, and in particular that 0 is an allowed random value. (We are grateful to Heiko Bauke for clarifying for us these properties of GFSR random number generators.)

For more information see,

- Robert M. Ziff, "Four-tap shift-register-sequence random-number generators", Computers in Physics, 12(4), Jul/Aug 1998, pp 385–392.

## 18.10 Unix random number generators

The standard Unix random number generators `rand`, `random` and `rand48` are provided as part of GSL. Although these generators are widely available individually often they aren't all available on the same platform. This makes it difficult to write portable code using them and so we have included the complete set of Unix generators in GSL for convenience. Note that these generators don't produce high-quality randomness and aren't suitable for work requiring accurate statistics. However, if you won't be measuring statistical quantities and just want to introduce some variation into your program then these generators are quite acceptable.

*gsl\_rng\_type* \***gsl\_rng\_rand**

This is the BSD `rand` generator. Its sequence is

$$x_{n+1} = (ax_n + c) \pmod{m}$$

with  $a = 1103515245$ ,  $c = 12345$  and  $m = 2^{31}$ . The seed specifies the initial value,  $x_1$ . The period of this generator is  $2^{31}$ , and it uses 1 word of storage per generator.

*gsl\_rng\_type* \***gsl\_rng\_random\_bsd**

*gsl\_rng\_type* \***gsl\_rng\_random\_libc5**

*gsl\_rng\_type* \***gsl\_rng\_random\_glibc2**

These generators implement the `random` family of functions, a set of linear feedback shift register generators originally used in BSD Unix. There are several versions of `random` in use today: the original BSD version (e.g. on SunOS4), a `libc5` version (found on older GNU/Linux systems) and a `glibc2` version. Each version uses a different seeding procedure, and thus produces different sequences.

The original BSD routines accepted a variable length buffer for the generator state, with longer buffers providing higher-quality randomness. The `random` function implemented algorithms for buffer lengths of 8, 32, 64, 128 and 256 bytes, and the algorithm with the largest length that would fit into the user-supplied buffer was used. To support these algorithms additional generators are available with the following names:

```
gsl_rng_random8_bsd
gsl_rng_random32_bsd
gsl_rng_random64_bsd
gsl_rng_random128_bsd
gsl_rng_random256_bsd
```

where the numeric suffix indicates the buffer length. The original BSD `random` function used a 128-byte default buffer and so *gsl\_rng\_random\_bsd* has been made equivalent to `gsl_rng_random128_bsd`. Corresponding versions of the `libc5` and `glibc2` generators are also available, with the names `gsl_rng_random8_libc5`, `gsl_rng_random8_glibc2`, etc.

*gsl\_rng\_type* \***gsl\_rng\_rand48**

This is the Unix `rand48` generator. Its sequence is

$$x_{n+1} = (ax_n + c) \pmod{m}$$

defined on 48-bit unsigned integers with  $a = 25214903917$ ,  $c = 11$  and  $m = 2^{48}$ . The seed specifies the upper 32 bits of the initial value,  $x_1$ , with the lower 16 bits set to `0x330E`. The function *gsl\_rng\_get*(`&`) returns the upper 32 bits from each term of the sequence. This does not have a direct parallel in the original `rand48` functions, but forcing the result to type `long int` reproduces the output of `mrnd48`. The function *gsl\_rng\_uniform*(`&`) uses the full 48 bits of internal state to return the double precision number  $x_n/m$ , which is equivalent to the function `drand48`. Note that some versions of the GNU C Library contained a bug in `mrnd48` function which caused it to produce different results (only the lower 16-bits of the return value were set).

## 18.11 Other random number generators

The generators in this section are provided for compatibility with existing libraries. If you are converting an existing program to use GSL then you can select these generators to check your new implementation against the original one, using the same random number generator. After verifying that your new program reproduces the original results you can then switch to a higher-quality generator.

Note that most of the generators in this section are based on single linear congruence relations, which are the least sophisticated type of generator. In particular, linear congruences have poor properties when used with a non-prime modulus, as several of these routines do (e.g. with a power of two modulus,  $2^{31}$  or  $2^{32}$ ). This leads to periodicity in the least significant bits of each number, with only the higher bits having any randomness. Thus if you want to produce a random bitstream it is best to avoid using the least significant bits.

### *gsl\_rng\_type* \***gsl\_rng\_ranf**

This is the CRAY random number generator RANF. Its sequence is

$$x_{n+1} = (ax_n) \pmod m$$

defined on 48-bit unsigned integers with  $a = 44485709377909$  and  $m = 2^{48}$ . The seed specifies the lower 32 bits of the initial value,  $x_1$ , with the lowest bit set to prevent the seed taking an even value. The upper 16 bits of  $x_1$  are set to 0. A consequence of this procedure is that the pairs of seeds 2 and 3, 4 and 5, etc.: produce the same sequences.

The generator compatible with the CRAY MATHLIB routine RANF. It produces double precision floating point numbers which should be identical to those from the original RANF.

There is a subtlety in the implementation of the seeding. The initial state is reversed through one step, by multiplying by the modular inverse of  $a \pmod m$ . This is done for compatibility with the original CRAY implementation.

Note that you can only seed the generator with integers up to  $2^{32}$ , while the original CRAY implementation uses non-portable wide integers which can cover all  $2^{48}$  states of the generator.

The function *gsl\_rng\_get()* returns the upper 32 bits from each term of the sequence. The function *gsl\_rng\_uniform()* uses the full 48 bits to return the double precision number  $x_n/m$ .

The period of this generator is  $2^{46}$ .

### *gsl\_rng\_type* \***gsl\_rng\_ranmar**

This is the RANMAR lagged-fibonacci generator of Marsaglia, Zaman and Tsang. It is a 24-bit generator, originally designed for single-precision IEEE floating point numbers. It was included in the CERNLIB high-energy physics library.

### *gsl\_rng\_type* \***gsl\_rng\_r250**

This is the shift-register generator of Kirkpatrick and Stoll. The sequence is based on the recurrence

$$x_n = x_{n-103} \oplus x_{n-250}$$

where  $\oplus$  denotes *exclusive-or*, defined on 32-bit words. The period of this generator is about  $2^{250}$  and it uses 250 words of state per generator.

For more information see,

- S. Kirkpatrick and E. Stoll, "A very fast shift-register sequence random number generator", Journal of Computational Physics, 40, 517–526 (1981)

### *gsl\_rng\_type* \***gsl\_rng\_tt800**

This is an earlier version of the twisted generalized feedback shift-register generator, and has been superseded by the development of MT19937. However, it is still an acceptable generator in its own right. It has a period of  $2^{800}$  and uses 33 words of storage per generator.

For more information see,

- Makoto Matsumoto and Yoshiharu Kurita, “Twisted GFSR Generators II”, ACM Transactions on Modelling and Computer Simulation, Vol.: 4, No.: 3, 1994, pages 254–266.

*gsl\_rng\_type* \***gsl\_rng\_vax**

This is the VAX generator MTH\$RANDOM. Its sequence is,

$$x_{n+1} = (ax_n + c) \pmod{m}$$

with  $a = 69069$ ,  $c = 1$  and  $m = 2^{32}$ . The seed specifies the initial value,  $x_1$ . The period of this generator is  $2^{32}$  and it uses 1 word of storage per generator.

*gsl\_rng\_type* \***gsl\_rng\_transputer**

This is the random number generator from the INMOS Transputer Development system. Its sequence is,

$$x_{n+1} = (ax_n) \pmod{m}$$

with  $a = 1664525$  and  $m = 2^{32}$ . The seed specifies the initial value,  $x_1$ .

*gsl\_rng\_type* \***gsl\_rng\_randu**

This is the IBM RANDU generator. Its sequence is

$$x_{n+1} = (ax_n) \pmod{m}$$

with  $a = 65539$  and  $m = 2^{31}$ . The seed specifies the initial value,  $x_1$ . The period of this generator was only  $2^{29}$ . It has become a textbook example of a poor generator.

*gsl\_rng\_type* \***gsl\_rng\_minstd**

This is Park and Miller’s “minimal standard” MINSTD generator, a simple linear congruence which takes care to avoid the major pitfalls of such algorithms. Its sequence is,

$$x_{n+1} = (ax_n) \pmod{m}$$

with  $a = 16807$  and  $m = 2^{31} - 1 = 2147483647$ . The seed specifies the initial value,  $x_1$ . The period of this generator is about  $2^{31}$ .

This generator was used in the IMSL Library (subroutine RNUN) and in MATLAB (the RAND function) in the past. It is also sometimes known by the acronym “GGL” (I’m not sure what that stands for).

For more information see,

- Park and Miller, “Random Number Generators: Good ones are hard to find”, Communications of the ACM, October 1988, Volume 31, No 10, pages 1192–1201.

*gsl\_rng\_type* \***gsl\_rng\_uni***gsl\_rng\_type* \***gsl\_rng\_uni32**

This is a reimplement of the 16-bit SLATEC random number generator RUNIF. A generalization of the generator to 32 bits is provided by *gsl\_rng\_uni32*. The original source code is available from NETLIB.

*gsl\_rng\_type* \***gsl\_rng\_slatec**

This is the SLATEC random number generator RAND. It is ancient. The original source code is available from NETLIB.

*gsl\_rng\_type* \***gsl\_rng\_zuf**

This is the ZUFALL lagged Fibonacci series generator of Peterson. Its sequence is,

$$t = u_{n-273} + u_{n-607}$$

$$u_n = t - \text{floor}(t)$$

The original source code is available from NETLIB. For more information see,



- W. Petersen, “Lagged Fibonacci Random Number Generators for the NEC SX-3”, International Journal of High Speed Computing (1994).

*gsl\_rng\_type* \***gsl\_rng\_knuthran2**

This is a second-order multiple recursive generator described by Knuth in *Seminumerical Algorithms*, 3rd Ed., page 108. Its sequence is,

$$x_n = (a_1x_{n-1} + a_2x_{n-2}) \pmod m$$

with  $a_1 = 271828183$ ,  $a_2 = 314159269$ , and  $m = 2^{31} - 1$ .

*gsl\_rng\_type* \***gsl\_rng\_knuthran2002**

*gsl\_rng\_type* \***gsl\_rng\_knuthran**

This is a second-order multiple recursive generator described by Knuth in *Seminumerical Algorithms*, 3rd Ed., Section 3.6. Knuth provides its C code. The updated routine *gsl\_rng\_knuthran2002* is from the revised 9th printing and corrects some weaknesses in the earlier version, which is implemented as *gsl\_rng\_knuthran*.

*gsl\_rng\_type* \***gsl\_rng\_borosh13**

*gsl\_rng\_type* \***gsl\_rng\_fishman18**

*gsl\_rng\_type* \***gsl\_rng\_fishman20**

*gsl\_rng\_type* \***gsl\_rng\_lecuyer21**

*gsl\_rng\_type* \***gsl\_rng\_waterman14**

These multiplicative generators are taken from Knuth’s *Seminumerical Algorithms*, 3rd Ed., pages 106–108. Their sequence is,

$$x_{n+1} = (ax_n) \pmod m$$

where the seed specifies the initial value,  $x_1$ . The parameters  $a$  and  $m$  are as follows, Borosh-Niederreiter:  $a = 1812433253$ ,  $m = 2^{32}$ , Fishman18:  $a = 62089911$ ,  $m = 2^{31} - 1$ , Fishman20:  $a = 48271$ ,  $m = 2^{31} - 1$ , L’Ecuyer:  $a = 40692$ ,  $m = 2^{31} - 249$ , Waterman:  $a = 1566083941$ ,  $m = 2^{32}$ .

*gsl\_rng\_type* \***gsl\_rng\_fishman2x**

This is the L’Ecuyer–Fishman random number generator. It is taken from Knuth’s *Seminumerical Algorithms*, 3rd Ed., page 108. Its sequence is,

$$z_{n+1} = (x_n - y_n) \pmod m$$

with  $m = 2^{31} - 1$ .  $x_n$  and  $y_n$  are given by the *fishman20* and *lecuyer21* algorithms. The seed specifies the initial value,  $x_1$ .

*gsl\_rng\_type* \***gsl\_rng\_coveyou**

This is the Coveyou random number generator. It is taken from Knuth’s *Seminumerical Algorithms*, 3rd Ed., Section 3.2.2. Its sequence is,

$$x_{n+1} = (x_n(x_n + 1)) \pmod m$$

with  $m = 2^{32}$ . The seed specifies the initial value,  $x_1$ .

## 18.12 Performance

The following table shows the relative performance of a selection the available random number generators. The fastest simulation quality generators are *taus*, *gfsr4* and *mt19937*. The generators which offer the best mathematically-proven quality are those based on the RANLUX algorithm:



```

1754 k ints/sec,      870 k doubles/sec, taus
1613 k ints/sec,      855 k doubles/sec, gfsr4
1370 k ints/sec,      769 k doubles/sec, mt19937
 565 k ints/sec,      571 k doubles/sec, ranlxs0
 400 k ints/sec,      405 k doubles/sec, ranlxs1
 490 k ints/sec,      389 k doubles/sec, mrg
 407 k ints/sec,      297 k doubles/sec, ranlux
 243 k ints/sec,      254 k doubles/sec, ranlxd1
 251 k ints/sec,      253 k doubles/sec, ranlxs2
 238 k ints/sec,      215 k doubles/sec, cmrg
 247 k ints/sec,      198 k doubles/sec, ranlux389
 141 k ints/sec,      140 k doubles/sec, ranlxd2

```

## 18.13 Examples

The following program demonstrates the use of a random number generator to produce uniform random numbers in the range [0.0, 1.0),

```

#include <stdio.h>
#include <gsl/gsl_rng.h>

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;

  int i, n = 10;

  gsl_rng_env_setup();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  for (i = 0; i < n; i++)
  {
    double u = gsl_rng_uniform (r);
    printf ("%0.5f\n", u);
  }

  gsl_rng_free (r);

  return 0;
}

```

Here is the output of the program,

```

0.99974
0.16291
0.28262
0.94720

```

(continues on next page)

(continued from previous page)

```
0.23166
0.48497
0.95748
0.74431
0.54004
0.73995
```

The numbers depend on the seed used by the generator. The default seed can be changed with the `GSL_RNG_SEED` environment variable to produce a different stream of numbers. The generator itself can be changed using the environment variable `GSL_RNG_TYPE`. Here is the output of the program using a seed value of 123 and the multiple-recursive generator `mrg`:

```
$ GSL_RNG_SEED=123 GSL_RNG_TYPE=mrg ./a.out
```

```
0.33050
0.86631
0.32982
0.67620
0.53391
0.06457
0.16847
0.70229
0.04371
0.86374
```

## 18.14 References and Further Reading

The subject of random number generation and testing is reviewed extensively in Knuth's *Seminumerical Algorithms*.

- Donald E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms* (Vol 2, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896842.

Further information is available in the review paper written by Pierre L'Ecuyer,

- P. L'Ecuyer, "Random Number Generation", Chapter 4 of the *Handbook on Simulation*, Jerry Banks Ed., Wiley, 1998, 93–137.
- <http://www.iro.umontreal.ca/~lecuyer/papers.html> in the file `handsim.ps`.

The source code for the DIEHARD random number generator tests is also available online,

- DIEHARD source code, G. Marsaglia, <http://stat.fsu.edu/pub/diehard/>

A comprehensive set of random number generator tests is available from NIST,

- NIST Special Publication 800-22, "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications".
- <http://csrc.nist.gov/rng/>

## 18.15 Acknowledgements

Thanks to Makoto Matsumoto, Takuji Nishimura and Yoshiharu Kurita for making the source code to their generators (MT19937, MM&TN; TT800, MM&YK) available under the GNU General Public License. Thanks to Martin Luscher for providing notes and source code for the RANLXS and RANLXD generators.



## QUASI-RANDOM SEQUENCES

This chapter describes functions for generating quasi-random sequences in arbitrary dimensions. A quasi-random sequence progressively covers a  $d$ -dimensional space with a set of points that are uniformly distributed. Quasi-random sequences are also known as low-discrepancy sequences. The quasi-random sequence generators use an interface that is similar to the interface for random number generators, except that seeding is not required—each generator produces a single sequence.

The functions described in this section are declared in the header file `gsl_qrng.h`.

### 19.1 Quasi-random number generator initialization

type **gsl\_qrng**

This is a workspace for computing quasi-random sequences.

*gsl\_qrng* \***gsl\_qrng\_alloc**(const *gsl\_qrng\_type* \*T, unsigned int d)

This function returns a pointer to a newly-created instance of a quasi-random sequence generator of type  $T$  and dimension  $d$ . If there is insufficient memory to create the generator then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

void **gsl\_qrng\_free**(*gsl\_qrng* \*q)

This function frees all the memory associated with the generator  $q$ .

void **gsl\_qrng\_init**(*gsl\_qrng* \*q)

This function reinitializes the generator  $q$  to its starting point. Note that quasi-random sequences do not use a seed and always produce the same set of values.

### 19.2 Sampling from a quasi-random number generator

int **gsl\_qrng\_get**(const *gsl\_qrng* \*q, double x[])

This function stores the next point from the sequence generator  $q$  in the array  $x$ . The space available for  $x$  must match the dimension of the generator. The point  $x$  will lie in the range  $0 < x_i < 1$  for each  $x_i$ . An inline version of this function is used when `HAVE_INLINE` is defined.

## 19.3 Auxiliary quasi-random number generator functions

const char \***gsl\_qrng\_name**(const *gsl\_qrng* \*q)

This function returns a pointer to the name of the generator.

size\_t **gsl\_qrng\_size**(const *gsl\_qrng* \*q)

void \***gsl\_qrng\_state**(const *gsl\_qrng* \*q)

These functions return a pointer to the state of generator *r* and its size. You can use this information to access the state directly. For example, the following code will write the state of a generator to a stream:

```
void * state = gsl_qrng_state (q);
size_t n = gsl_qrng_size (q);
fwrite (state, n, 1, stream);
```

## 19.4 Saving and restoring quasi-random number generator state

int **gsl\_qrng\_memcpy**(*gsl\_qrng* \*dest, const *gsl\_qrng* \*src)

This function copies the quasi-random sequence generator *src* into the pre-existing generator *dest*, making *dest* into an exact copy of *src*. The two generators must be of the same type.

*gsl\_qrng* \***gsl\_qrng\_clone**(const *gsl\_qrng* \*q)

This function returns a pointer to a newly created generator which is an exact copy of the generator *q*.

## 19.5 Quasi-random number generator algorithms

The following quasi-random sequence algorithms are available,

type **gsl\_qrng\_type**

*gsl\_qrng\_type* \***gsl\_qrng\_niederreiter\_2**

This generator uses the algorithm described in Bratley, Fox, Niederreiter, ACM Trans. Model. Comp. Sim. 2, 195 (1992). It is valid up to 12 dimensions.

*gsl\_qrng\_type* \***gsl\_qrng\_sobol**

This generator uses the Sobol sequence described in Antonov, Saleev, USSR Comput. Maths. Math. Phys. 19, 252 (1980). It is valid up to 40 dimensions.

*gsl\_qrng\_type* \***gsl\_qrng\_halton**

*gsl\_qrng\_type* \***gsl\_qrng\_reversehalton**

These generators use the Halton and reverse Halton sequences described in J.H. Halton, Numerische Mathematik, 2, 84-90 (1960) and B. Vandewoestyne and R. Cools Computational and Applied Mathematics, 189, 1&2, 341-361 (2006). They are valid up to 1229 dimensions.

## 19.6 Examples

The following program prints the first 1024 points of the 2-dimensional Sobol sequence.

```
#include <stdio.h>
#include <gsl/gsl_qrng.h>

int
main (void)
{
  int i;
  gsl_qrng * q = gsl_qrng_alloc (gsl_qrng_sobol, 2);

  for (i = 0; i < 1024; i++)
  {
    double v[2];
    gsl_qrng_get (q, v);
    printf ("%0.5f %0.5f\n", v[0], v[1]);
  }

  gsl_qrng_free (q);
  return 0;
}
```

Here is the output from the program:

```
$ ./a.out
0.50000 0.50000
0.75000 0.25000
0.25000 0.75000
0.37500 0.37500
0.87500 0.87500
0.62500 0.12500
0.12500 0.62500
....
```

It can be seen that successive points progressively fill-in the spaces between previous points.

Fig. 19.1 shows the distribution in the x-y plane of the first 1024 points from the Sobol sequence,

## 19.7 References

The implementations of the quasi-random sequence routines are based on the algorithms described in the following paper,

- P. Bratley and B.L. Fox and H. Niederreiter, “Algorithm 738: Programs to Generate Niederreiter’s Low-discrepancy Sequences”, ACM Transactions on Mathematical Software, Vol.: 20, No.: 4, December, 1994, p.: 494–495.

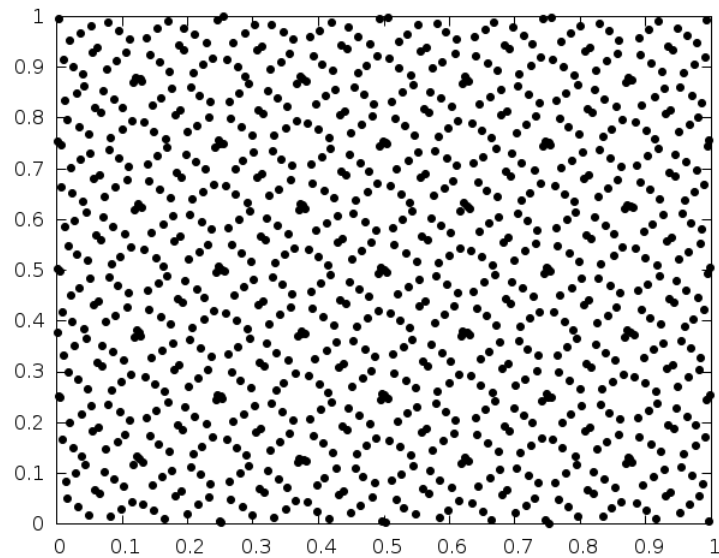


Fig. 19.1: Distribution of the first 1024 points from the quasi-random Sobol sequence



## RANDOM NUMBER DISTRIBUTIONS

This chapter describes functions for generating random variates and computing their probability distributions. Samples from the distributions described in this chapter can be obtained using any of the random number generators in the library as an underlying source of randomness.

In the simplest cases a non-uniform distribution can be obtained analytically from the uniform distribution of a random number generator by applying an appropriate transformation. This method uses one call to the random number generator. More complicated distributions are created by the *acceptance-rejection* method, which compares the desired distribution against a distribution which is similar and known analytically. This usually requires several samples from the generator.

The library also provides cumulative distribution functions and inverse cumulative distribution functions, sometimes referred to as quantile functions. The cumulative distribution functions and their inverses are computed separately for the upper and lower tails of the distribution, allowing full accuracy to be retained for small results.

The functions for random variates and probability density functions described in this section are declared in `gsl_randist.h`. The corresponding cumulative distribution functions are declared in `gsl_cdf.h`.

Note that the discrete random variate functions always return a value of type `unsigned int`, and on most platforms this has a maximum value of

$$2^{32} - 1 \approx 4.29 \times 10^9$$

They should only be called with a safe range of parameters (where there is a negligible probability of a variate exceeding this limit) to prevent incorrect results due to overflow.

### 20.1 Introduction

Continuous random number distributions are defined by a probability density function,  $p(x)$ , such that the probability of  $x$  occurring in the infinitesimal range  $x$  to  $x + dx$  is  $p(x)dx$ .

The cumulative distribution function for the lower tail  $P(x)$  is defined by the integral,

$$P(x) = \int_{-\infty}^x dx' p(x')$$

and gives the probability of a variate taking a value less than  $x$ .

The cumulative distribution function for the upper tail  $Q(x)$  is defined by the integral,

$$Q(x) = \int_x^{+\infty} dx' p(x')$$

and gives the probability of a variate taking a value greater than  $x$ .

The upper and lower cumulative distribution functions are related by  $P(x) + Q(x) = 1$  and satisfy  $0 \leq P(x) \leq 1$ ,  $0 \leq Q(x) \leq 1$ .

The inverse cumulative distributions,  $x = P^{-1}(P)$  and  $x = Q^{-1}(Q)$  give the values of  $x$  which correspond to a specific value of  $P$  or  $Q$ . They can be used to find confidence limits from probability values.

For discrete distributions the probability of sampling the integer value  $k$  is given by  $p(k)$ , where  $\sum_k p(k) = 1$ . The cumulative distribution for the lower tail  $P(k)$  of a discrete distribution is defined as,

$$P(k) = \sum_{i \leq k} p(i)$$

where the sum is over the allowed range of the distribution less than or equal to  $k$ .

The cumulative distribution for the upper tail of a discrete distribution  $Q(k)$  is defined as

$$Q(k) = \sum_{i > k} p(i)$$

giving the sum of probabilities for all values greater than  $k$ . These two definitions satisfy the identity  $P(k) + Q(k) = 1$ .

If the range of the distribution is 1 to  $n$  inclusive then  $P(n) = 1$ ,  $Q(n) = 0$  while  $P(1) = p(1)$ ,  $Q(1) = 1 - p(1)$ .

## 20.2 The Gaussian Distribution

double **gsl\_ran\_gaussian**(const *gsl\_rng* \*r, double sigma)

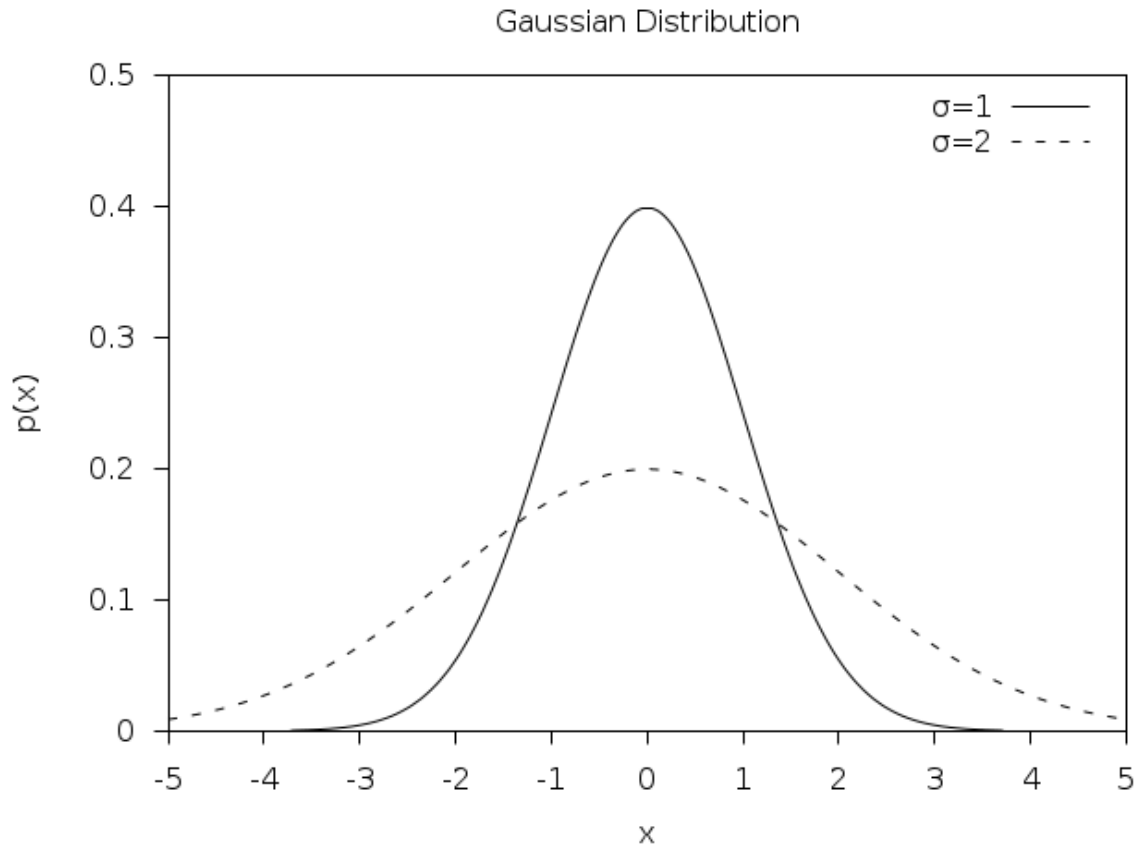
This function returns a Gaussian random variate, with mean zero and standard deviation *sigma*. The probability distribution for Gaussian random variates is,

$$p(x)dx = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$$

for  $x$  in the range  $-\infty$  to  $+\infty$ . Use the transformation  $z = \mu + x$  on the numbers returned by *gsl\_ran\_gaussian()* to obtain a Gaussian distribution with mean  $\mu$ . This function uses the Box-Muller algorithm which requires two calls to the random number generator *r*.

double **gsl\_ran\_gaussian\_pdf**(double x, double sigma)

This function computes the probability density  $p(x)$  at  $x$  for a Gaussian distribution with standard deviation *sigma*, using the formula given above.



double **gsl\_ran\_gaussian\_ziggurat**(const *gsl\_rng* \*r, double sigma)

double **gsl\_ran\_gaussian\_ratio\_method**(const *gsl\_rng* \*r, double sigma)

This function computes a Gaussian random variate using the alternative Marsaglia-Tsang ziggurat and Kinderman-Monahan-Leva ratio methods. The Ziggurat algorithm is the fastest available algorithm in most cases.

double **gsl\_ran\_ugaussian**(const *gsl\_rng* \*r)

double **gsl\_ran\_ugaussian\_pdf**(double x)

double **gsl\_ran\_ugaussian\_ratio\_method**(const *gsl\_rng* \*r)

These functions compute results for the unit Gaussian distribution. They are equivalent to the functions above with a standard deviation of one,  $\sigma = 1$ .

double **gsl\_cdf\_gaussian\_P**(double x, double sigma)

double **gsl\_cdf\_gaussian\_Q**(double x, double sigma)

double **gsl\_cdf\_gaussian\_Pinv**(double P, double sigma)

double **gsl\_cdf\_gaussian\_Qinv**(double Q, double sigma)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Gaussian distribution with standard deviation *sigma*.

double **gsl\_cdf\_ugaussian\_P**(double x)

double **gsl\_cdf\_ugaussian\_Q**(double x)

double **gsl\_cdf\_ugaussian\_Pinv**(double P)

double **gsl\_cdf\_ugaussian\_Qinv**(double Q)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the unit Gaussian distribution.

## 20.3 The Gaussian Tail Distribution

double **gsl\_ran\_gaussian\_tail**(const *gsl\_rng* \*r, double a, double sigma)

This function provides random variates from the upper tail of a Gaussian distribution with standard deviation *sigma*. The values returned are larger than the lower limit *a*, which must be positive. The method is based on Marsaglia's famous rectangle-wedge-tail algorithm (Ann. Math. Stat. 32, 894–899 (1961)), with this aspect explained in Knuth, v2, 3rd ed, p139,586 (exercise 11).

The probability distribution for Gaussian tail random variates is,

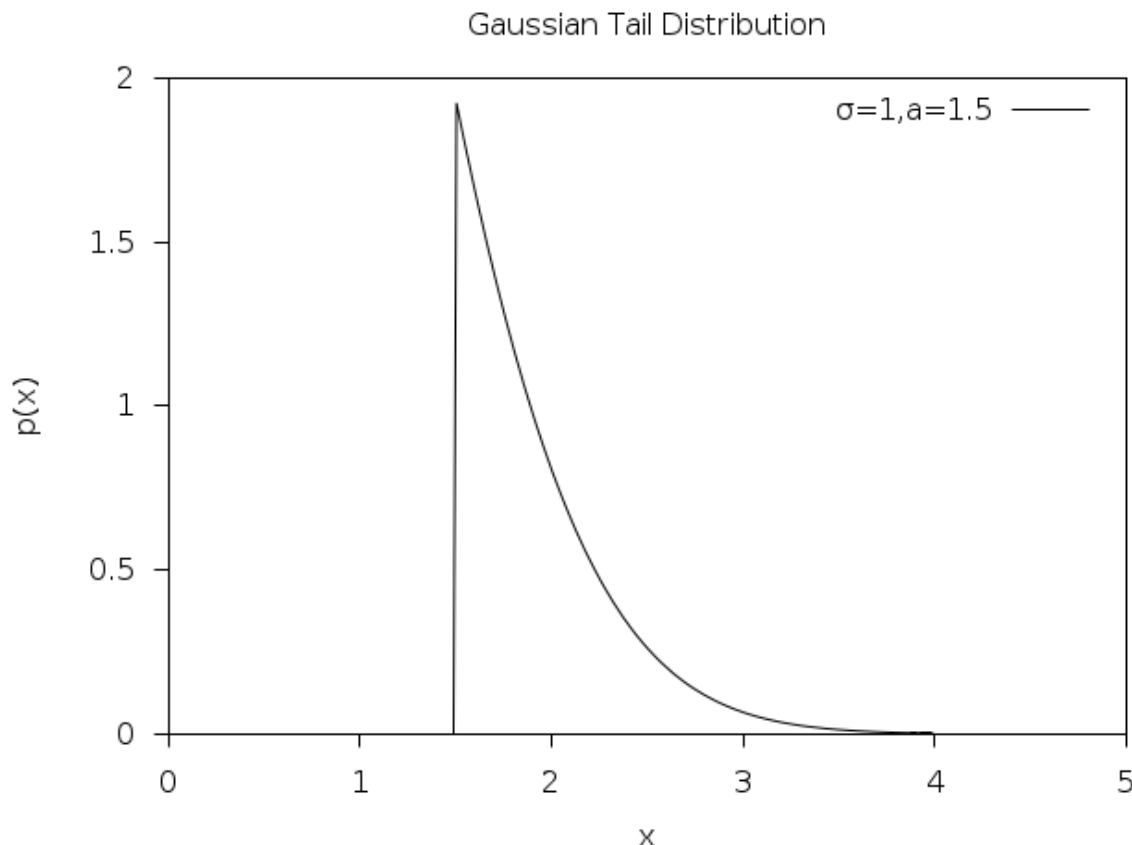
$$p(x)dx = \frac{1}{N(a; \sigma)\sqrt{2\pi\sigma^2}} \exp(-x^2/2\sigma^2)dx$$

for  $x > a$  where  $N(a; \sigma)$  is the normalization constant,

$$N(a; \sigma) = \frac{1}{2} \operatorname{erfc}\left(\frac{a}{\sqrt{2}\sigma}\right).$$

double **gsl\_ran\_gaussian\_tail\_pdf**(double x, double a, double sigma)

This function computes the probability density  $p(x)$  at  $x$  for a Gaussian tail distribution with standard deviation *sigma* and lower limit *a*, using the formula given above.



double **gsl\_ran\_ugaussian\_tail**(const *gsl\_rng* \*r, double a)

double **gsl\_ran\_ugaussian\_tail\_pdf**(double x, double a)

These functions compute results for the tail of a unit Gaussian distribution. They are equivalent to the functions above with a standard deviation of one,  $\sigma = 1$ .



## 20.4 The Bivariate Gaussian Distribution

void **gsl\_ran\_bivariate\_gaussian**(const *gsl\_rng* \*r, double sigma\_x, double sigma\_y, double rho, double \*x, double \*y)

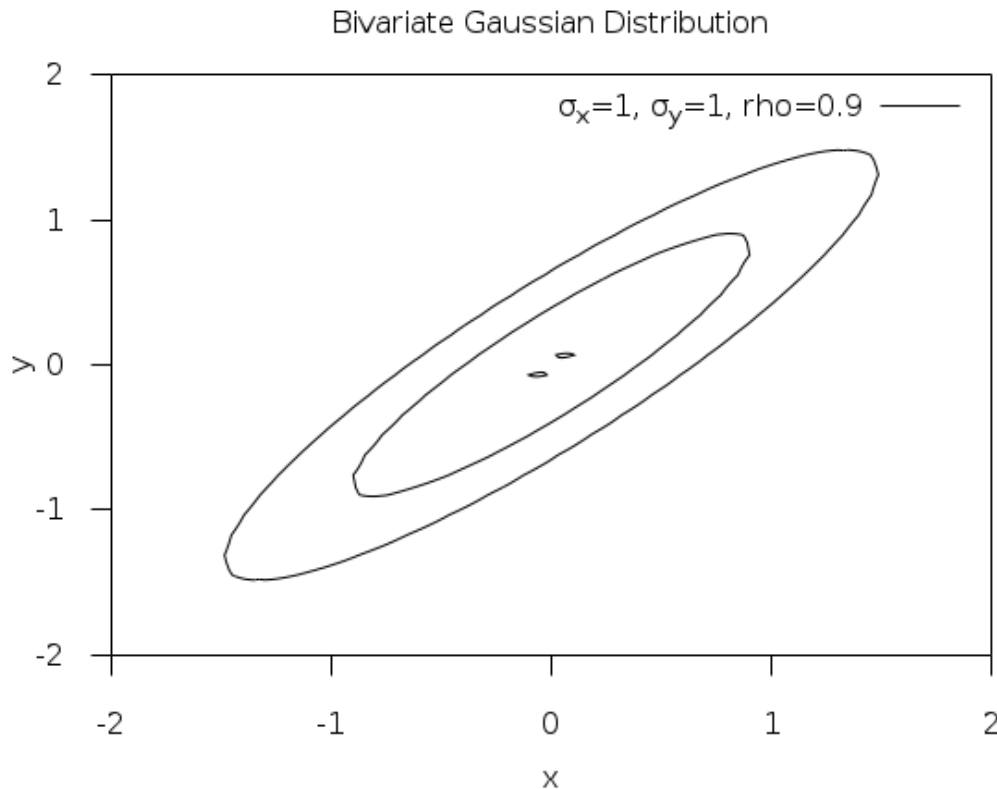
This function generates a pair of correlated Gaussian variates, with mean zero, correlation coefficient *rho* and standard deviations *sigma\_x* and *sigma\_y* in the *x* and *y* directions. The probability distribution for bivariate Gaussian random variates is,

$$p(x, y)dxdy = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left(-\frac{(x^2/\sigma_x^2 + y^2/\sigma_y^2 - 2\rho xy/(\sigma_x\sigma_y))}{2(1-\rho^2)}\right) dxdy$$

for *x, y* in the range  $-\infty$  to  $+\infty$ . The correlation coefficient *rho* should lie between 1 and  $-1$ .

double **gsl\_ran\_bivariate\_gaussian\_pdf**(double x, double y, double sigma\_x, double sigma\_y, double rho)

This function computes the probability density  $p(x, y)$  at  $(x, y)$  for a bivariate Gaussian distribution with standard deviations *sigma\_x*, *sigma\_y* and correlation coefficient *rho*, using the formula given above.



## 20.5 The Multivariate Gaussian Distribution

int **gsl\_ran\_multivariate\_gaussian**(const *gsl\_rng* \*r, const *gsl\_vector* \*mu, const *gsl\_matrix* \*L, *gsl\_vector* \*result)

This function generates a random vector satisfying the  $k$ -dimensional multivariate Gaussian distribution with mean  $\mu$  and variance-covariance matrix  $\Sigma$ . On input, the  $k$ -vector  $\mu$  is given in *mu*, and the Cholesky factor of the  $k$ -by- $k$  matrix  $\Sigma = LL^T$  is given in the lower triangle of *L*, as output from *gsl\_linalg\_cholesky\_decomp()*. The random vector is stored in *result* on output. The probability distribution for multivariate Gaussian random variates is

$$p(x_1, \dots, x_k) dx_1 \dots dx_k = \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right) dx_1 \dots dx_k$$

int **gsl\_ran\_multivariate\_gaussian\_pdf**(const *gsl\_vector* \*x, const *gsl\_vector* \*mu, const *gsl\_matrix* \*L, double \*result, *gsl\_vector* \*work)

int **gsl\_ran\_multivariate\_gaussian\_log\_pdf**(const *gsl\_vector* \*x, const *gsl\_vector* \*mu, const *gsl\_matrix* \*L, double \*result, *gsl\_vector* \*work)

These functions compute  $p(x)$  or  $\log p(x)$  at the point *x*, using mean vector *mu* and variance-covariance matrix specified by its Cholesky factor *L* using the formula above. Additional workspace of length  $k$  is required in *work*.

int **gsl\_ran\_multivariate\_gaussian\_mean**(const *gsl\_matrix* \*X, *gsl\_vector* \*mu\_hat)

Given a set of  $n$  samples  $X_j$  from a  $k$ -dimensional multivariate Gaussian distribution, this function computes the maximum likelihood estimate of the mean of the distribution, given by

$$\hat{\mu} = \frac{1}{n} \sum_{j=1}^n X_j$$

The samples  $X_1, X_2, \dots, X_n$  are given in the  $n$ -by- $k$  matrix *X*, and the maximum likelihood estimate of the mean is stored in *mu\_hat* on output.

int **gsl\_ran\_multivariate\_gaussian\_vcov**(const *gsl\_matrix* \*X, *gsl\_matrix* \*sigma\_hat)

Given a set of  $n$  samples  $X_j$  from a  $k$ -dimensional multivariate Gaussian distribution, this function computes the maximum likelihood estimate of the variance-covariance matrix of the distribution, given by

$$\hat{\Sigma} = \frac{1}{n} \sum_{j=1}^n (X_j - \hat{\mu})(X_j - \hat{\mu})^T$$

The samples  $X_1, X_2, \dots, X_n$  are given in the  $n$ -by- $k$  matrix *X* and the maximum likelihood estimate of the variance-covariance matrix is stored in *sigma\_hat* on output.



## 20.6 The Exponential Distribution

double **gsl\_ran\_exponential**(const *gsl\_rng* \*r, double mu)

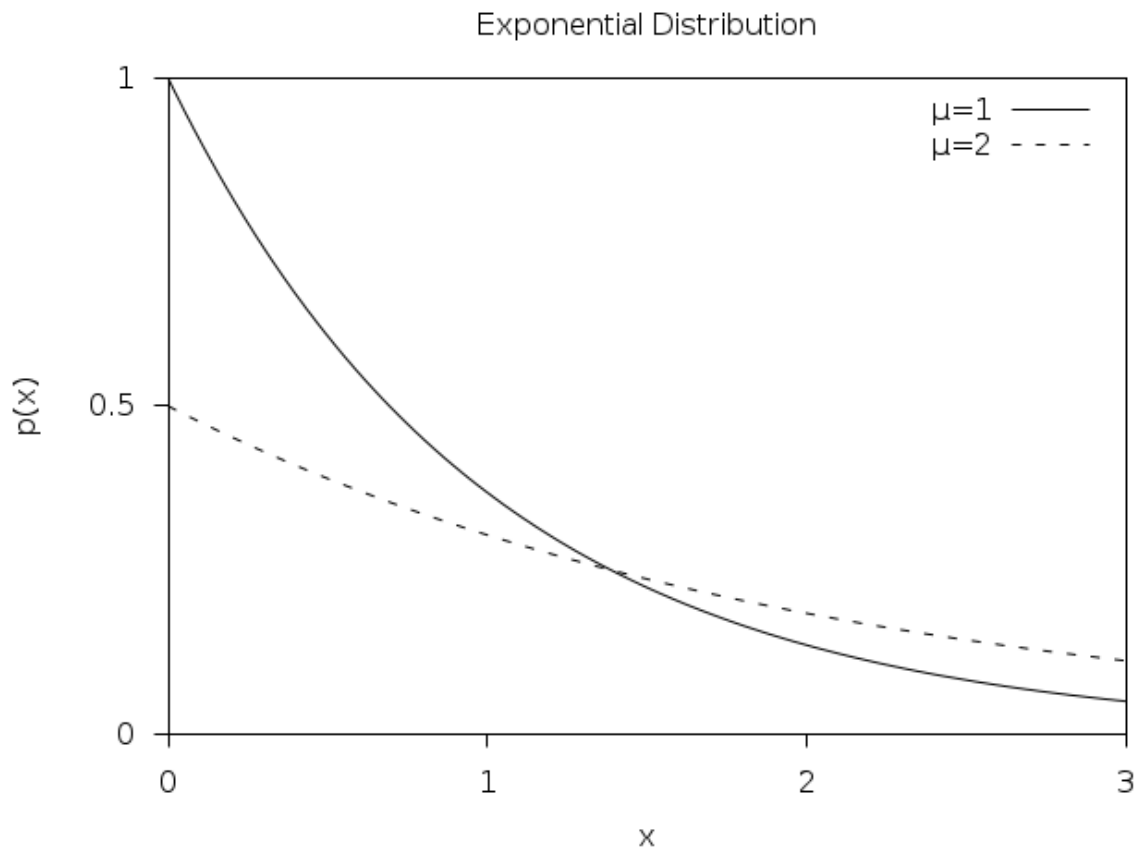
This function returns a random variate from the exponential distribution with mean *mu*. The distribution is,

$$p(x)dx = \frac{1}{\mu} \exp(-x/\mu)dx$$

for  $x \geq 0$ .

double **gsl\_ran\_exponential\_pdf**(double x, double mu)

This function computes the probability density  $p(x)$  at  $x$  for an exponential distribution with mean *mu*, using the formula given above.



double **gsl\_cdf\_exponential\_P**(double x, double mu)

double **gsl\_cdf\_exponential\_Q**(double x, double mu)

double **gsl\_cdf\_exponential\_Pinv**(double P, double mu)

double **gsl\_cdf\_exponential\_Qinv**(double Q, double mu)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the exponential distribution with mean *mu*.

## 20.7 The Laplace Distribution

double **gsl\_ran\_laplace**(const *gsl\_rng* \*r, double a)

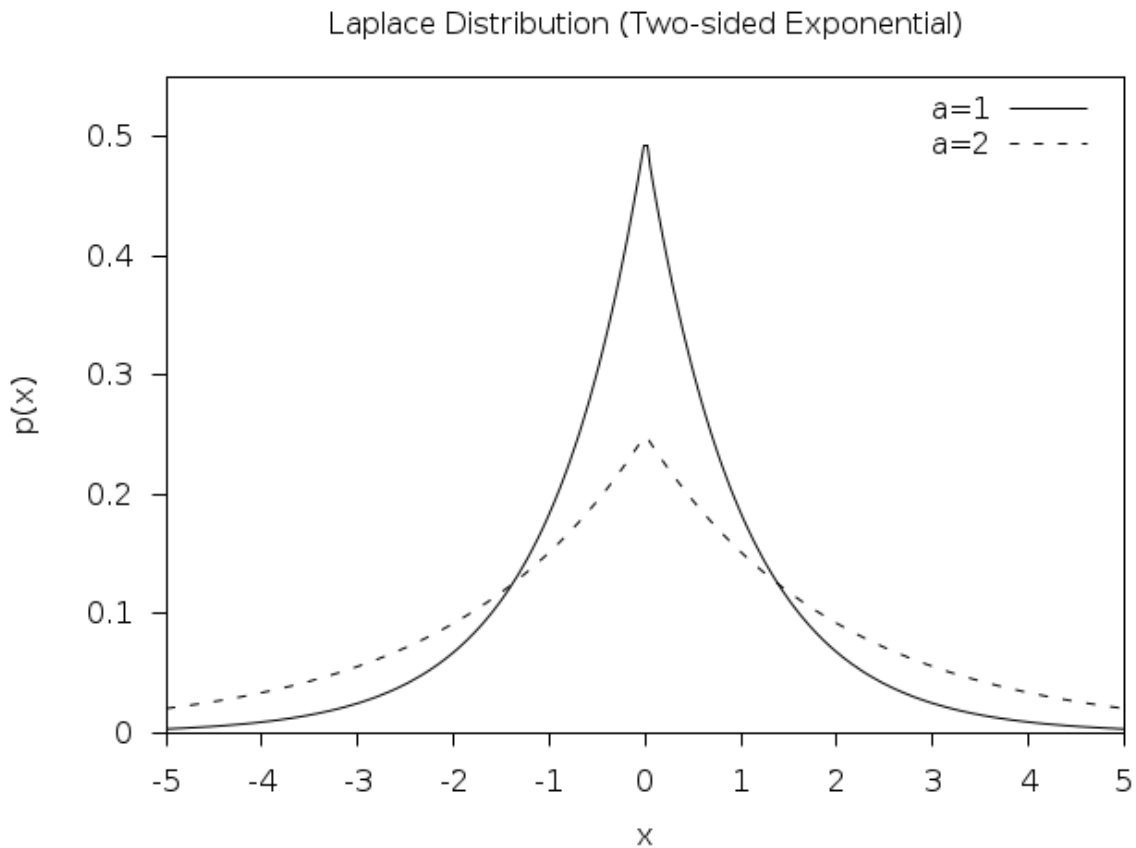
This function returns a random variate from the Laplace distribution with width *a*. The distribution is,

$$p(x)dx = \frac{1}{2a} \exp(-|x/a|)dx$$

for  $-\infty < x < \infty$ .

double **gsl\_ran\_laplace\_pdf**(double x, double a)

This function computes the probability density  $p(x)$  at *x* for a Laplace distribution with width *a*, using the formula given above.



double **gsl\_cdf\_laplace\_P**(double x, double a)

double **gsl\_cdf\_laplace\_Q**(double x, double a)

double **gsl\_cdf\_laplace\_Pinv**(double P, double a)

double **gsl\_cdf\_laplace\_Qinv**(double Q, double a)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Laplace distribution with width *a*.

## 20.8 The Exponential Power Distribution

double **gsl\_ran\_exppow**(const *gsl\_rng* \*r, double a, double b)

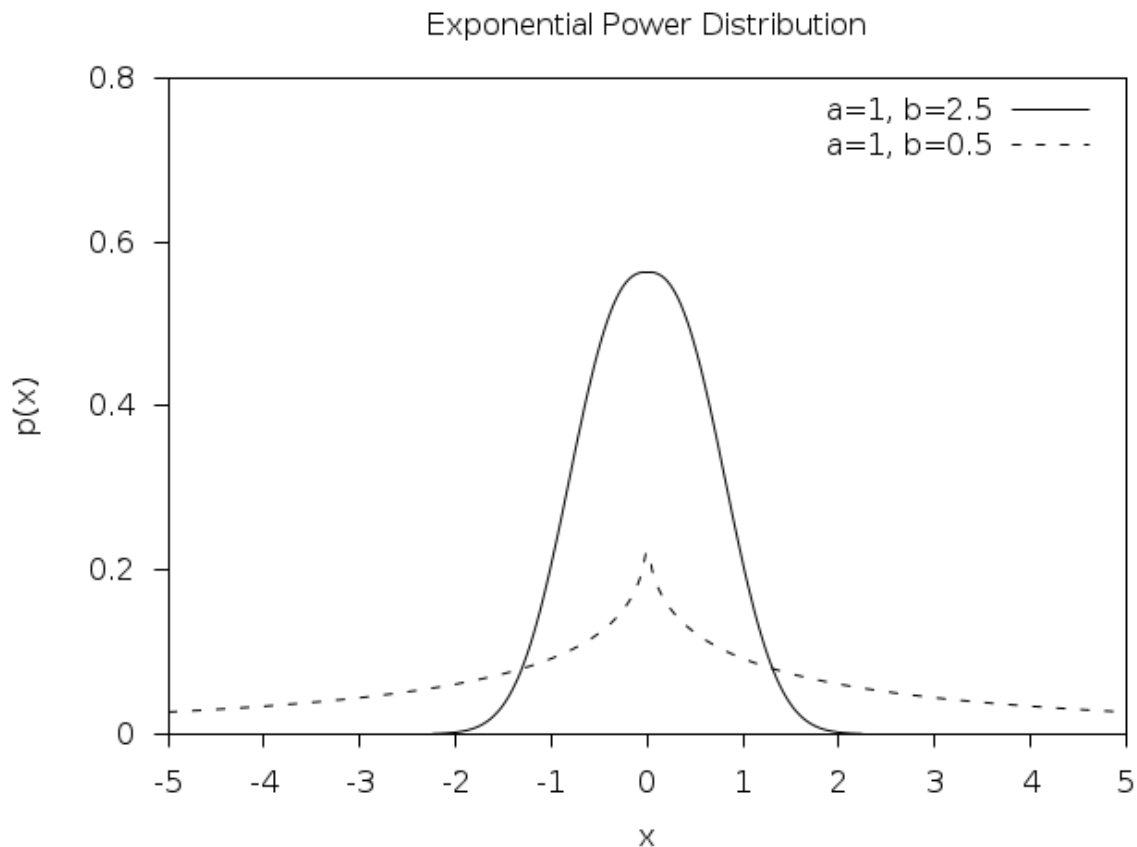
This function returns a random variate from the exponential power distribution with scale parameter  $a$  and exponent  $b$ . The distribution is,

$$p(x)dx = \frac{1}{2a\Gamma(1 + 1/b)} \exp(-|x/a|^b)dx$$

for  $x \geq 0$ . For  $b = 1$  this reduces to the Laplace distribution. For  $b = 2$  it has the same form as a Gaussian distribution, but with  $a = \sqrt{2}\sigma$ .

double **gsl\_ran\_exppow\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for an exponential power distribution with scale parameter  $a$  and exponent  $b$ , using the formula given above.



double **gsl\_cdf\_exppow\_P**(double x, double a, double b)

double **gsl\_cdf\_exppow\_Q**(double x, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  for the exponential power distribution with parameters  $a$  and  $b$ .

## 20.9 The Cauchy Distribution

double **gsl\_ran\_cauchy**(const *gsl\_rng* \*r, double a)

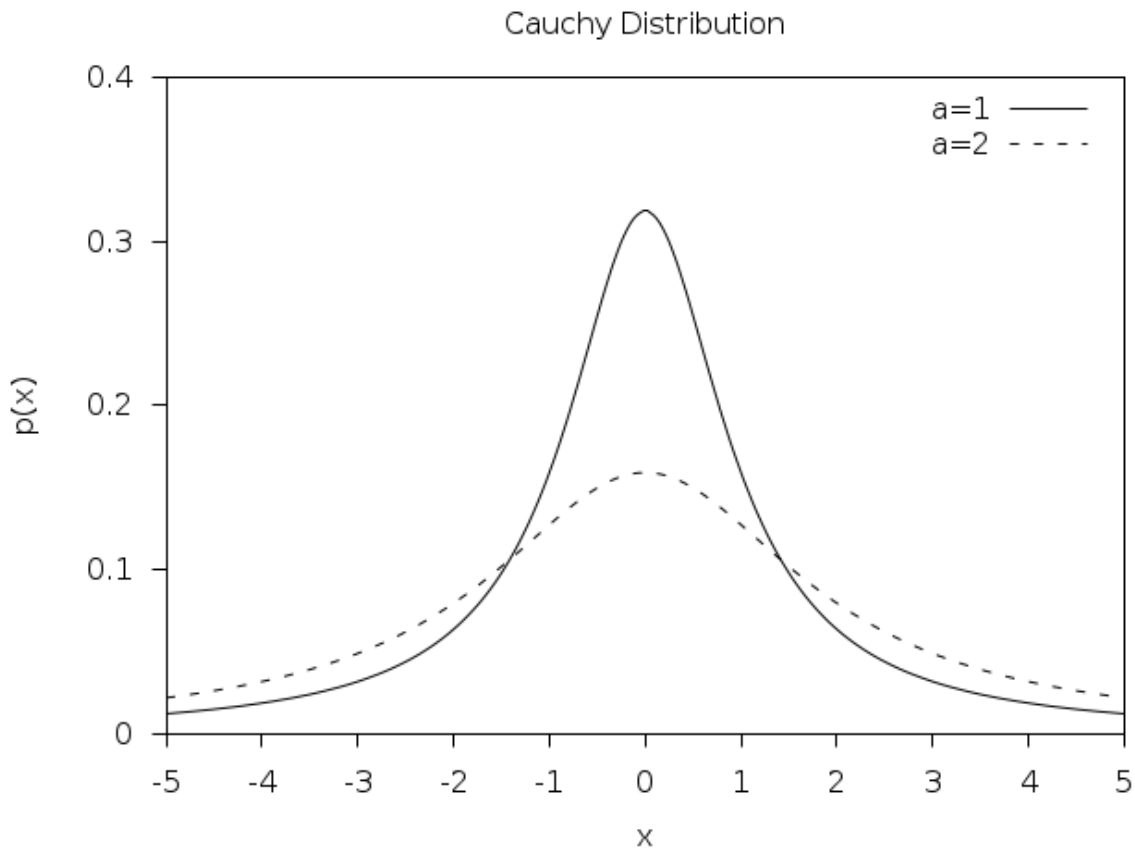
This function returns a random variate from the Cauchy distribution with scale parameter  $a$ . The probability distribution for Cauchy random variates is,

$$p(x)dx = \frac{1}{a\pi(1 + (x/a)^2)}dx$$

for  $x$  in the range  $-\infty$  to  $+\infty$ . The Cauchy distribution is also known as the Lorentz distribution.

double **gsl\_ran\_cauchy\_pdf**(double x, double a)

This function computes the probability density  $p(x)$  at  $x$  for a Cauchy distribution with scale parameter  $a$ , using the formula given above.



double **gsl\_cdf\_cauchy\_P**(double x, double a)

double **gsl\_cdf\_cauchy\_Q**(double x, double a)

double **gsl\_cdf\_cauchy\_Pinv**(double P, double a)

double **gsl\_cdf\_cauchy\_Qinv**(double Q, double a)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Cauchy distribution with scale parameter  $a$ .

## 20.10 The Rayleigh Distribution

double **gsl\_ran\_rayleigh**(const *gsl\_rng* \*r, double sigma)

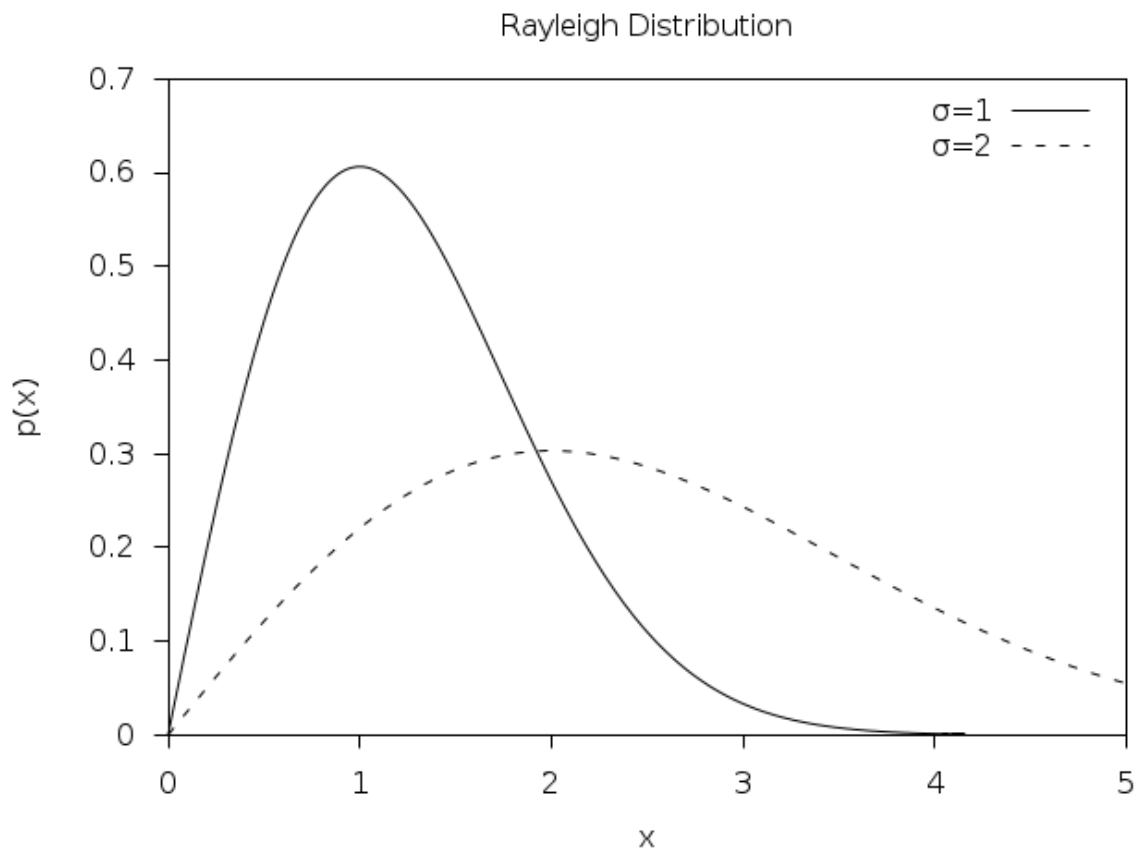
This function returns a random variate from the Rayleigh distribution with scale parameter *sigma*. The distribution is,

$$p(x)dx = \frac{x}{\sigma^2} \exp(-x^2/(2\sigma^2))dx$$

for  $x > 0$ .

double **gsl\_ran\_rayleigh\_pdf**(double x, double sigma)

This function computes the probability density  $p(x)$  at  $x$  for a Rayleigh distribution with scale parameter *sigma*, using the formula given above.



double **gsl\_cdf\_rayleigh\_P**(double x, double sigma)

double **gsl\_cdf\_rayleigh\_Q**(double x, double sigma)

double **gsl\_cdf\_rayleigh\_Pinv**(double P, double sigma)

double **gsl\_cdf\_rayleigh\_Qinv**(double Q, double sigma)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Rayleigh distribution with scale parameter *sigma*.

## 20.11 The Rayleigh Tail Distribution

double **gsl\_ran\_rayleigh\_tail**(const *gsl\_rng* \*r, double a, double sigma)

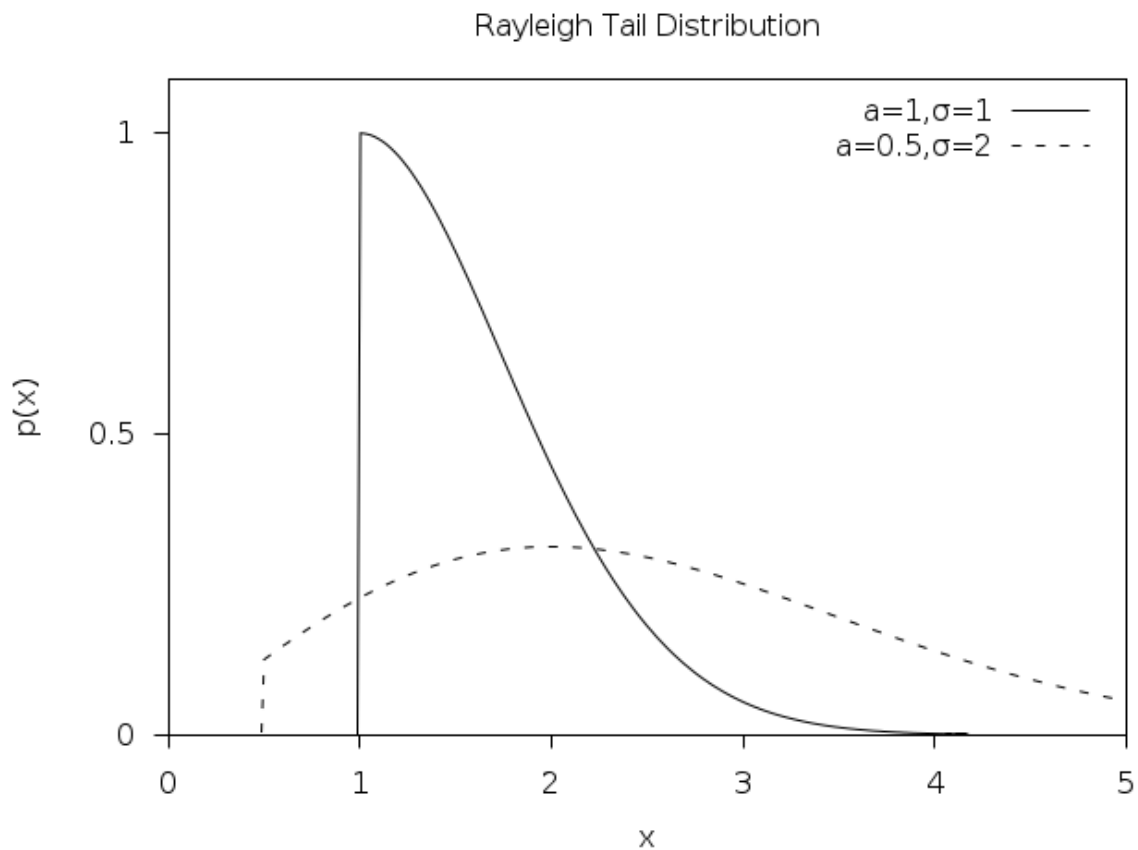
This function returns a random variate from the tail of the Rayleigh distribution with scale parameter *sigma* and a lower limit of *a*. The distribution is,

$$p(x)dx = \frac{x}{\sigma^2} \exp(-(a^2 - x^2)/(2\sigma^2))dx$$

for  $x > a$ .

double **gsl\_ran\_rayleigh\_tail\_pdf**(double x, double a, double sigma)

This function computes the probability density  $p(x)$  at  $x$  for a Rayleigh tail distribution with scale parameter *sigma* and lower limit *a*, using the formula given above.



## 20.12 The Landau Distribution

double **gsl\_ran\_landau**(const *gsl\_rng* \*r)

This function returns a random variate from the Landau distribution. The probability distribution for Landau random variates is defined analytically by the complex integral,

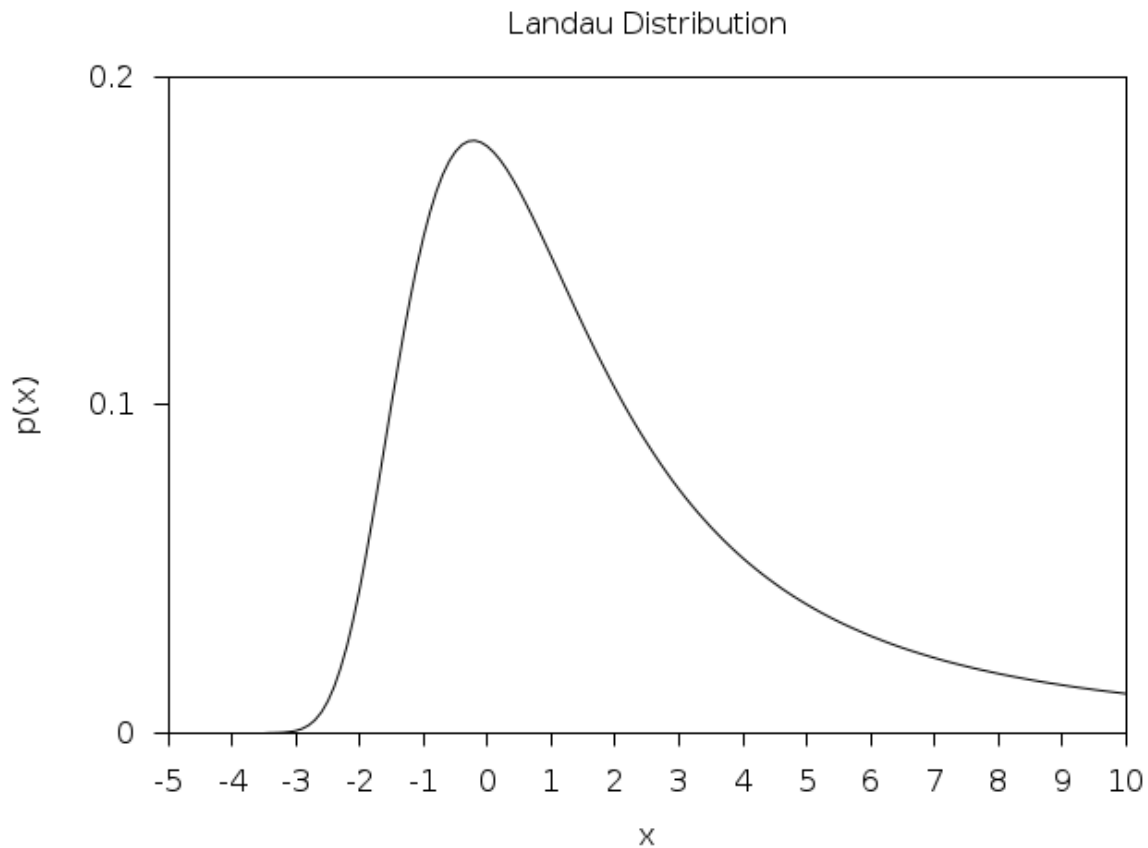
$$p(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} ds \exp(s \log(s) + xs)$$

For numerical purposes it is more convenient to use the following equivalent form of the integral,

$$p(x) = (1/\pi) \int_0^\infty dt \exp(-t \log(t) - xt) \sin(\pi t).$$

double **gsl\_ran\_landau\_pdf**(double x)

This function computes the probability density  $p(x)$  at  $x$  for the Landau distribution using an approximation to the formula given above.



## 20.13 The Levy alpha-Stable Distributions

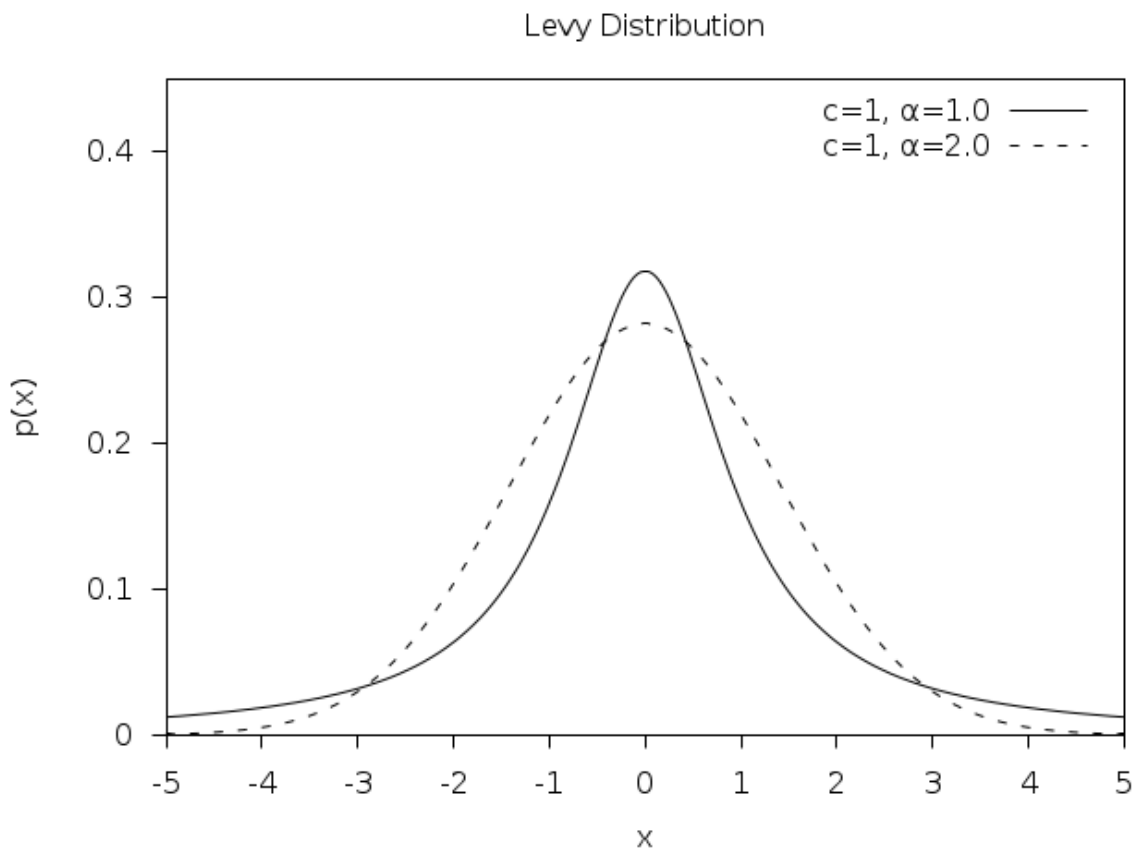
double `gsl_ran_levy`(const `gsl_rng` \*r, double c, double alpha)

This function returns a random variate from the Levy symmetric stable distribution with scale `c` and exponent `alpha`. The symmetric stable probability distribution is defined by a Fourier transform,

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dt \exp(-itx - |ct|^\alpha)$$

There is no explicit solution for the form of  $p(x)$  and the library does not define a corresponding pdf function. For  $\alpha = 1$  the distribution reduces to the Cauchy distribution. For  $\alpha = 2$  it is a Gaussian distribution with  $\sigma = \sqrt{2}c$ . For  $\alpha < 1$  the tails of the distribution become extremely wide.

The algorithm only works for  $0 < \alpha \leq 2$ .





## 20.14 The Levy skew alpha-Stable Distribution

double `gsl_ran_levy_skew`(const `gsl_rng` \*r, double c, double alpha, double beta)

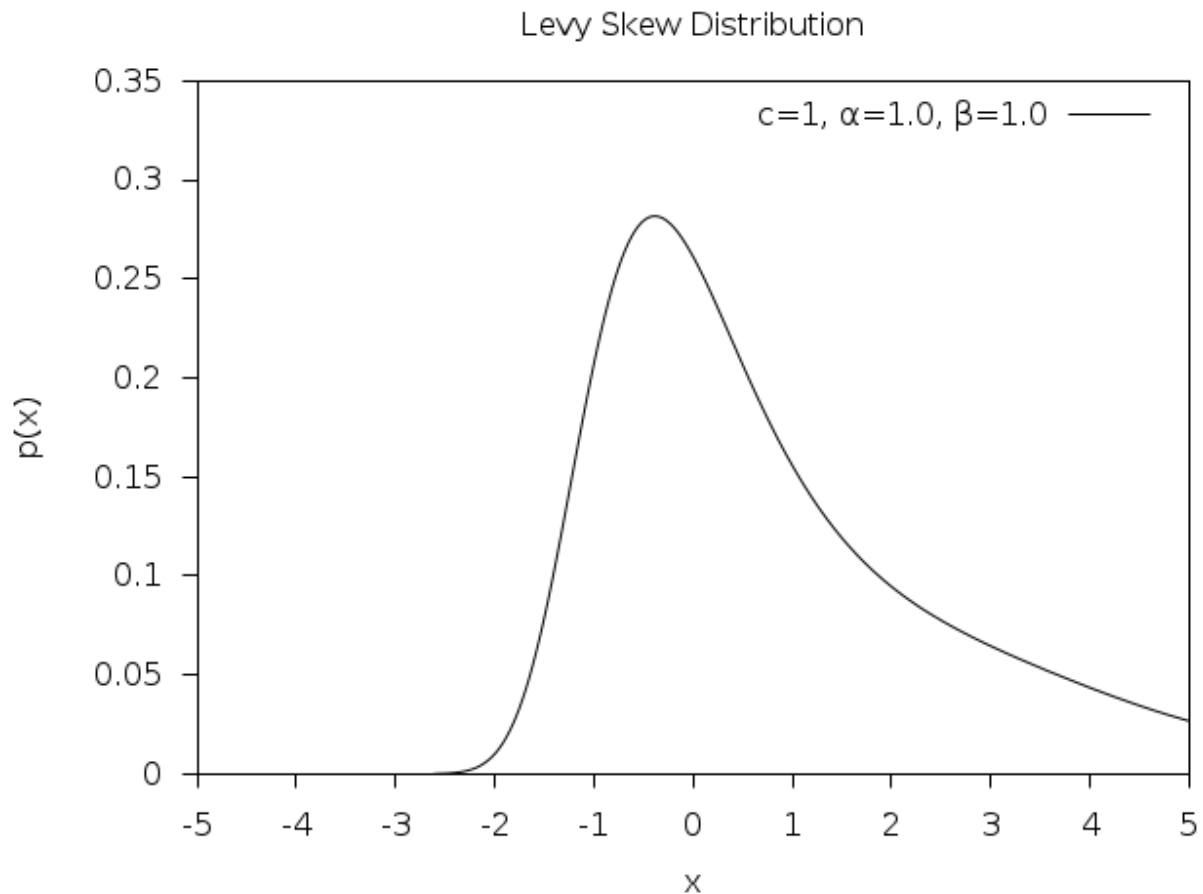
This function returns a random variate from the Levy skew stable distribution with scale `c`, exponent `alpha` and skewness parameter `beta`. The skewness parameter must lie in the range  $[-1, 1]$ . The Levy skew stable probability distribution is defined by a Fourier transform,

$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dt \exp(-itx - |ct|^\alpha (1 - i\beta \operatorname{sgn}(t) \tan(\pi\alpha/2)))$$

When  $\alpha = 1$  the term  $\tan(\pi\alpha/2)$  is replaced by  $-(2/\pi) \log |t|$ . There is no explicit solution for the form of  $p(x)$  and the library does not define a corresponding pdf function. For  $\alpha = 2$  the distribution reduces to a Gaussian distribution with  $\sigma = \sqrt{2}c$  and the skewness parameter has no effect. For  $\alpha < 1$  the tails of the distribution become extremely wide. The symmetric distribution corresponds to  $\beta = 0$ .

The algorithm only works for  $0 < \alpha \leq 2$ .

The Levy alpha-stable distributions have the property that if  $N$  alpha-stable variates are drawn from the distribution  $p(c, \alpha, \beta)$  then the sum  $Y = X_1 + X_2 + \dots + X_N$  will also be distributed as an alpha-stable variate,  $p(N^{1/\alpha}c, \alpha, \beta)$ .



## 20.15 The Gamma Distribution

double **gsl\_ran\_gamma**(const *gsl\_rng* \*r, double a, double b)

This function returns a random variate from the gamma distribution. The distribution function is,

$$p(x)dx = \frac{1}{\Gamma(a)b^a} x^{a-1} e^{-x/b} dx$$

for  $x > 0$ .

The gamma distribution with an integer parameter  $a$  is known as the Erlang distribution.

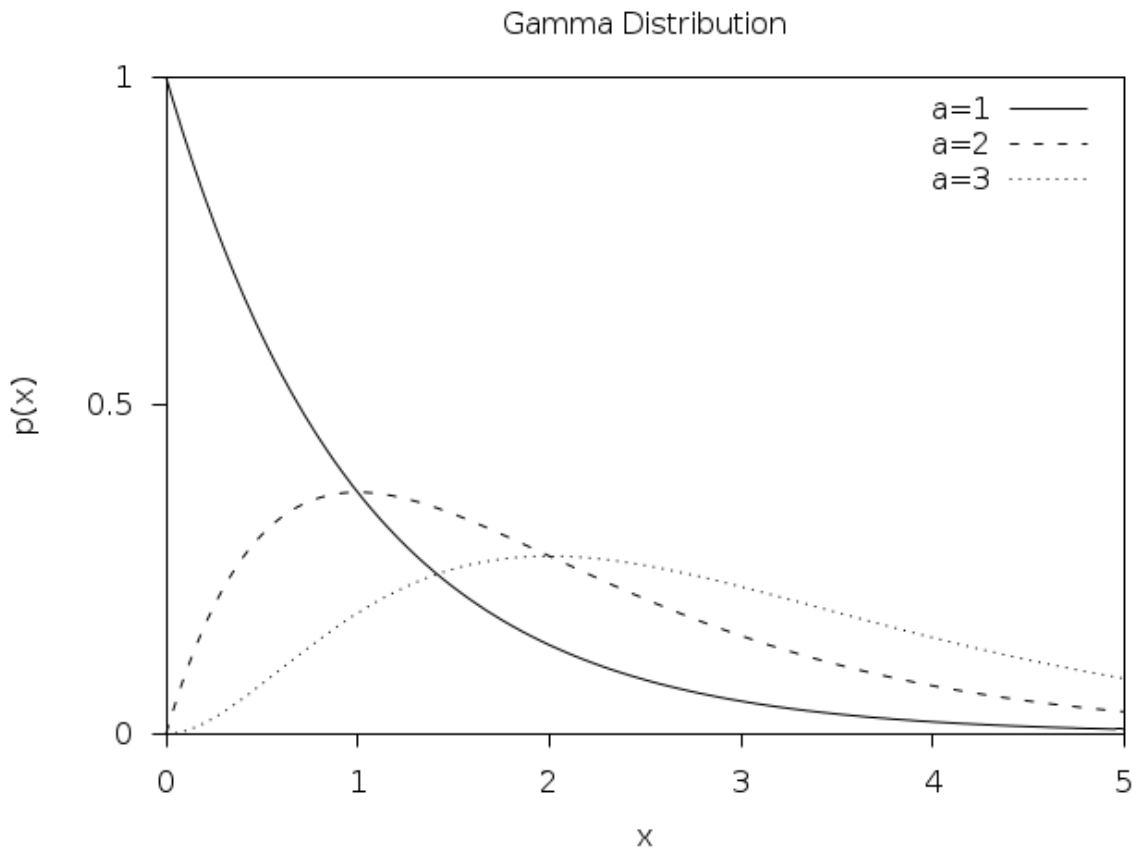
The variates are computed using the Marsaglia-Tsang fast gamma method. This function for this method was previously called `gsl_ran_gamma_mt()` and can still be accessed using this name.

double **gsl\_ran\_gamma\_knuth**(const *gsl\_rng* \*r, double a, double b)

This function returns a gamma variate using the algorithms from Knuth (vol 2).

double **gsl\_ran\_gamma\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for a gamma distribution with parameters  $a$  and  $b$ , using the formula given above.



double **gsl\_cdf\_gamma\_P**(double x, double a, double b)

double **gsl\_cdf\_gamma\_Q**(double x, double a, double b)

double **gsl\_cdf\_gamma\_Pinv**(double P, double a, double b)

double **gsl\_cdf\_gamma\_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the gamma distribution with parameters  $a$  and  $b$ .

## 20.16 The Flat (Uniform) Distribution

double **gsl\_ran\_flat**(const *gsl\_rng* \*r, double a, double b)

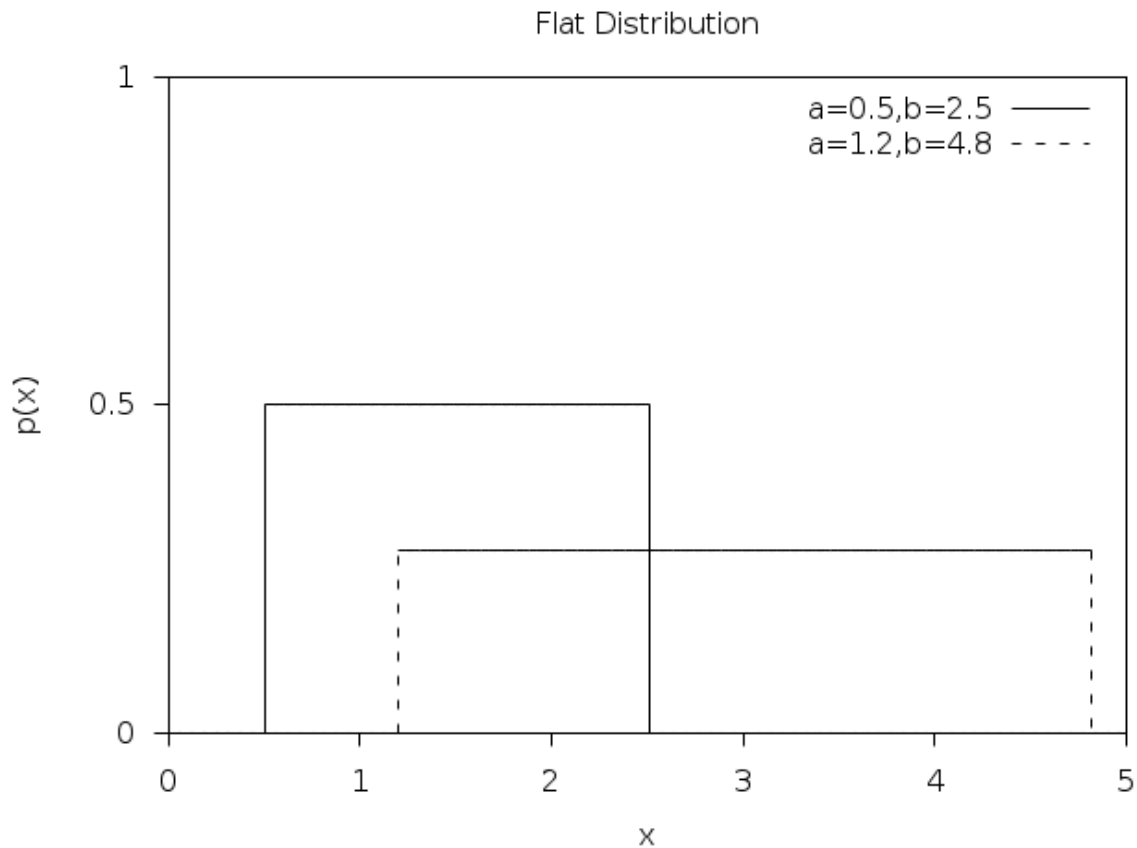
This function returns a random variate from the flat (uniform) distribution from  $a$  to  $b$ . The distribution is,

$$p(x)dx = \frac{1}{(b-a)}dx$$

if  $a \leq x < b$  and 0 otherwise.

double **gsl\_ran\_flat\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for a uniform distribution from  $a$  to  $b$ , using the formula given above.



double **gsl\_cdf\_flat\_P**(double x, double a, double b)

double **gsl\_cdf\_flat\_Q**(double x, double a, double b)

double **gsl\_cdf\_flat\_Pinv**(double P, double a, double b)

double **gsl\_cdf\_flat\_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for a uniform distribution from  $a$  to  $b$ .

## 20.17 The Lognormal Distribution

double **gsl\_ran\_lognormal**(const *gsl\_rng* \*r, double zeta, double sigma)

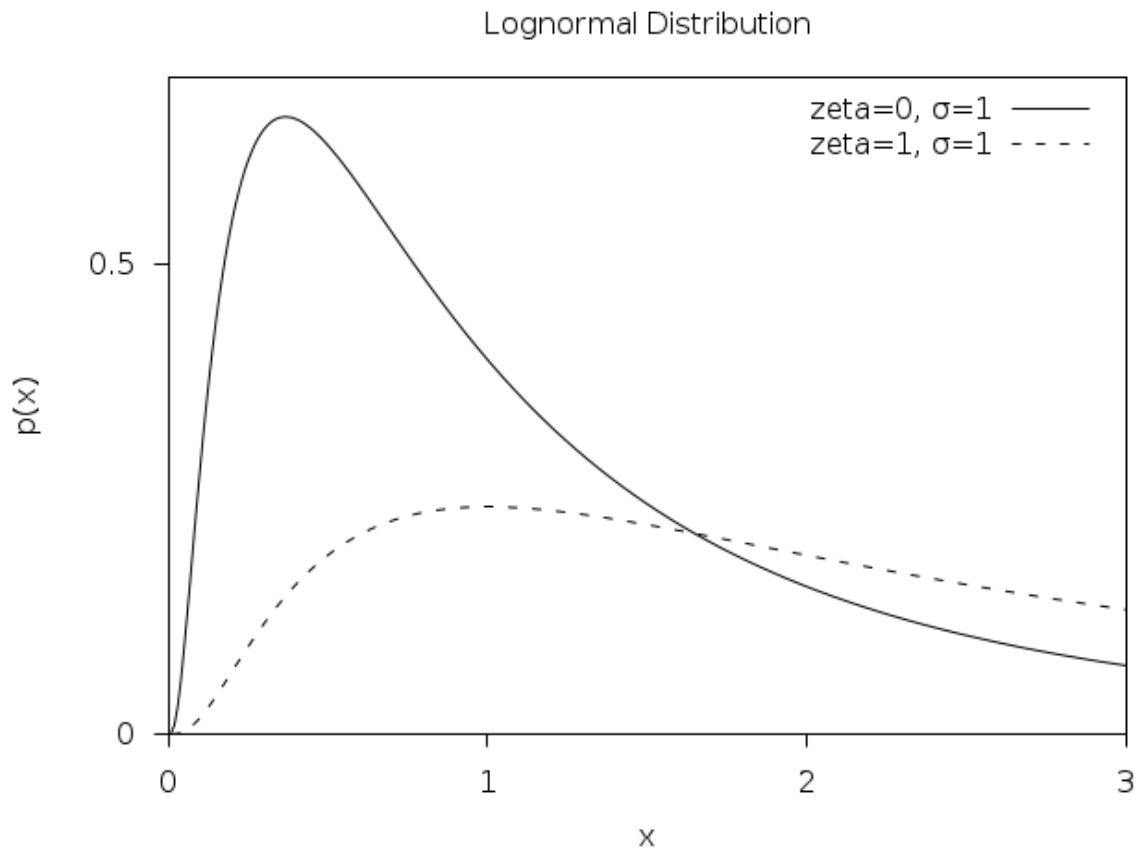
This function returns a random variate from the lognormal distribution. The distribution function is,

$$p(x)dx = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp(-(\ln(x) - \zeta)^2/2\sigma^2)dx$$

for  $x > 0$ .

double **gsl\_ran\_lognormal\_pdf**(double x, double zeta, double sigma)

This function computes the probability density  $p(x)$  at  $x$  for a lognormal distribution with parameters *zeta* and *sigma*, using the formula given above.



double **gsl\_cdf\_lognormal\_P**(double x, double zeta, double sigma)

double **gsl\_cdf\_lognormal\_Q**(double x, double zeta, double sigma)

double **gsl\_cdf\_lognormal\_Pinv**(double P, double zeta, double sigma)

double **gsl\_cdf\_lognormal\_Qinv**(double Q, double zeta, double sigma)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the lognormal distribution with parameters *zeta* and *sigma*.

## 20.18 The Chi-squared Distribution

The chi-squared distribution arises in statistics. If  $Y_i$  are  $n$  independent Gaussian random variates with unit variance then the sum-of-squares,

$$X_i = \sum_i Y_i^2$$

has a chi-squared distribution with  $n$  degrees of freedom.

double **gsl\_ran\_chisq**(const *gsl\_rng* \*r, double nu)

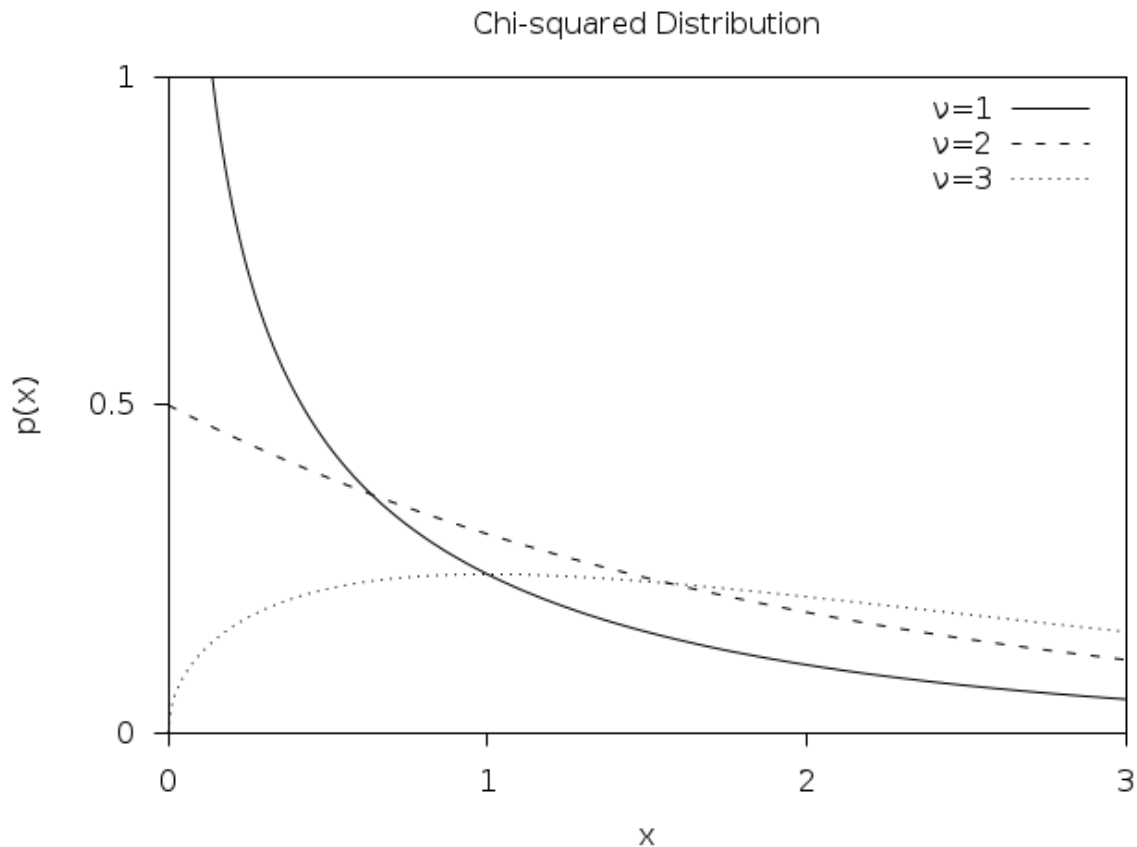
This function returns a random variate from the chi-squared distribution with  $nu$  degrees of freedom. The distribution function is,

$$p(x)dx = \frac{1}{2\Gamma(\nu/2)} (x/2)^{\nu/2-1} \exp(-x/2) dx$$

for  $x \geq 0$ .

double **gsl\_ran\_chisq\_pdf**(double x, double nu)

This function computes the probability density  $p(x)$  at  $x$  for a chi-squared distribution with  $nu$  degrees of freedom, using the formula given above.



double **gsl\_cdf\_chisq\_P**(double x, double nu)

double **gsl\_cdf\_chisq\_Q**(double x, double nu)

double **gsl\_cdf\_chisq\_Pinv**(double P, double nu)

double **gsl\_cdf\_chisq\_Qinv**(double Q, double nu)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the chi-squared distribution with  $nu$  degrees of freedom.

## 20.19 The F-distribution

The F-distribution arises in statistics. If  $Y_1$  and  $Y_2$  are chi-squared deviates with  $\nu_1$  and  $\nu_2$  degrees of freedom then the ratio,

$$X = \frac{(Y_1/\nu_1)}{(Y_2/\nu_2)}$$

has an F-distribution  $F(x; \nu_1, \nu_2)$ .

double **gsl\_ran\_fdist**(const *gsl\_rng* \*r, double nu1, double nu2)

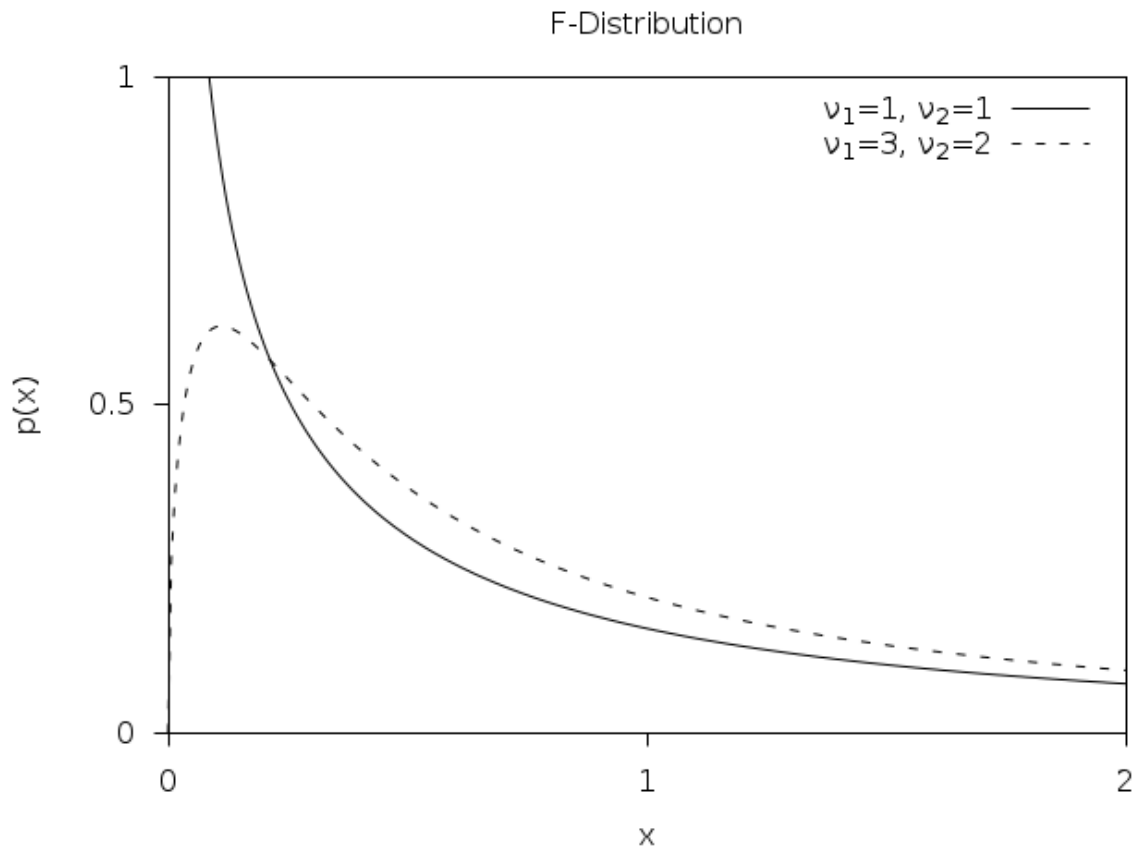
This function returns a random variate from the F-distribution with degrees of freedom *nu1* and *nu2*. The distribution function is,

$$p(x)dx = \frac{\Gamma((\nu_1 + \nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} x^{\nu_1/2-1} (\nu_2 + \nu_1 x)^{-\nu_1/2-\nu_2/2}$$

for  $x \geq 0$ .

double **gsl\_ran\_fdist\_pdf**(double x, double nu1, double nu2)

This function computes the probability density  $p(x)$  at  $x$  for an F-distribution with *nu1* and *nu2* degrees of freedom, using the formula given above.



double **gsl\_cdf\_fdist\_P**(double x, double nu1, double nu2)

double **gsl\_cdf\_fdist\_Q**(double x, double nu1, double nu2)



double **gsl\_cdf\_fdist\_Pinv**(double P, double nu1, double nu2)

double **gsl\_cdf\_fdist\_Qinv**(double Q, double nu1, double nu2)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the F-distribution with *nu1* and *nu2* degrees of freedom.

## 20.20 The t-distribution

The t-distribution arises in statistics. If  $Y_1$  has a normal distribution and  $Y_2$  has a chi-squared distribution with  $\nu$  degrees of freedom then the ratio,

$$X = \frac{Y_1}{\sqrt{Y_2/\nu}}$$

has a t-distribution  $t(x; \nu)$  with  $\nu$  degrees of freedom.

double **gsl\_ran\_tdist**(const *gsl\_rng* \*r, double nu)

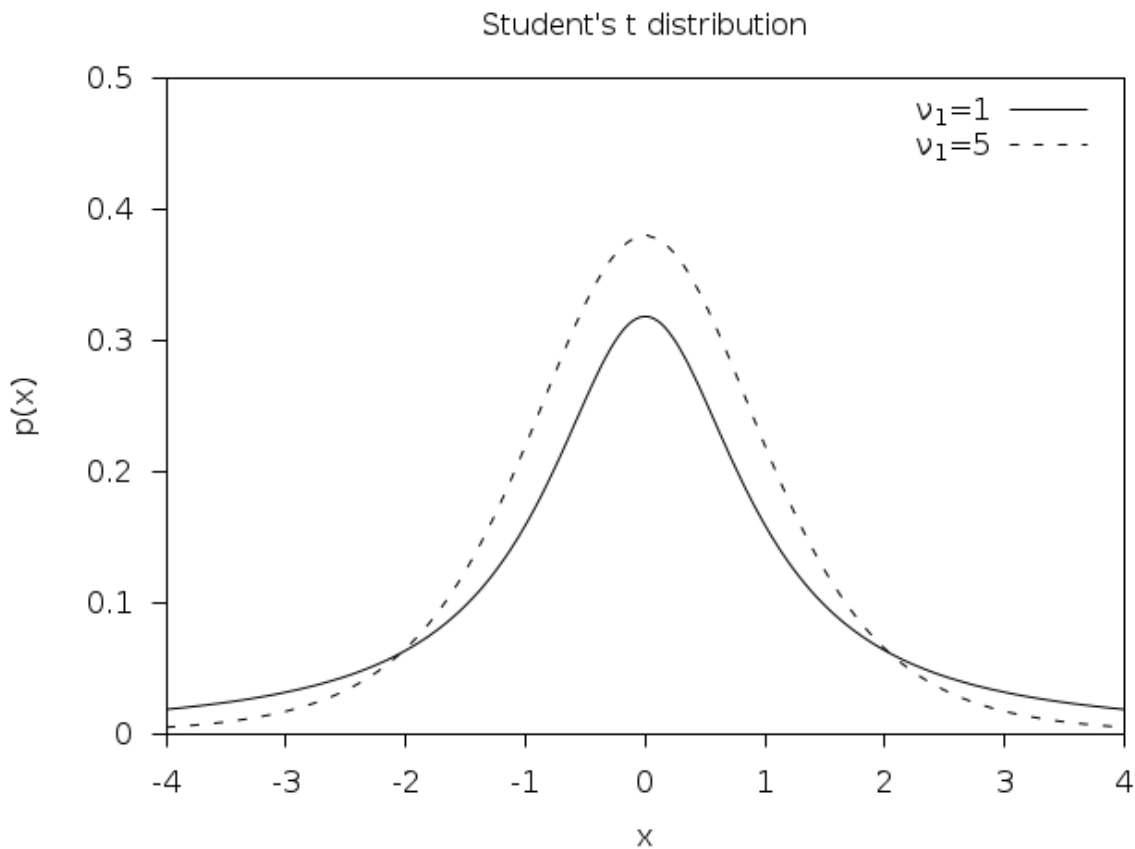
This function returns a random variate from the t-distribution. The distribution function is,

$$p(x)dx = \frac{\Gamma((\nu + 1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}(1 + x^2/\nu)^{-(\nu+1)/2}dx$$

for  $-\infty < x < +\infty$ .

double **gsl\_ran\_tdist\_pdf**(double x, double nu)

This function computes the probability density  $p(x)$  at  $x$  for a t-distribution with  $nu$  degrees of freedom, using the formula given above.



double **gsl\_cdf\_tdist\_P**(double x, double nu)

double **gsl\_cdf\_tdist\_Q**(double x, double nu)

double **gsl\_cdf\_tdist\_Pinv**(double P, double nu)

double **gsl\_cdf\_tdist\_Qinv**(double Q, double nu)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the t-distribution with  $nu$  degrees of freedom.

## 20.21 The Beta Distribution

double **gsl\_ran\_beta**(const *gsl\_rng* \*r, double a, double b)

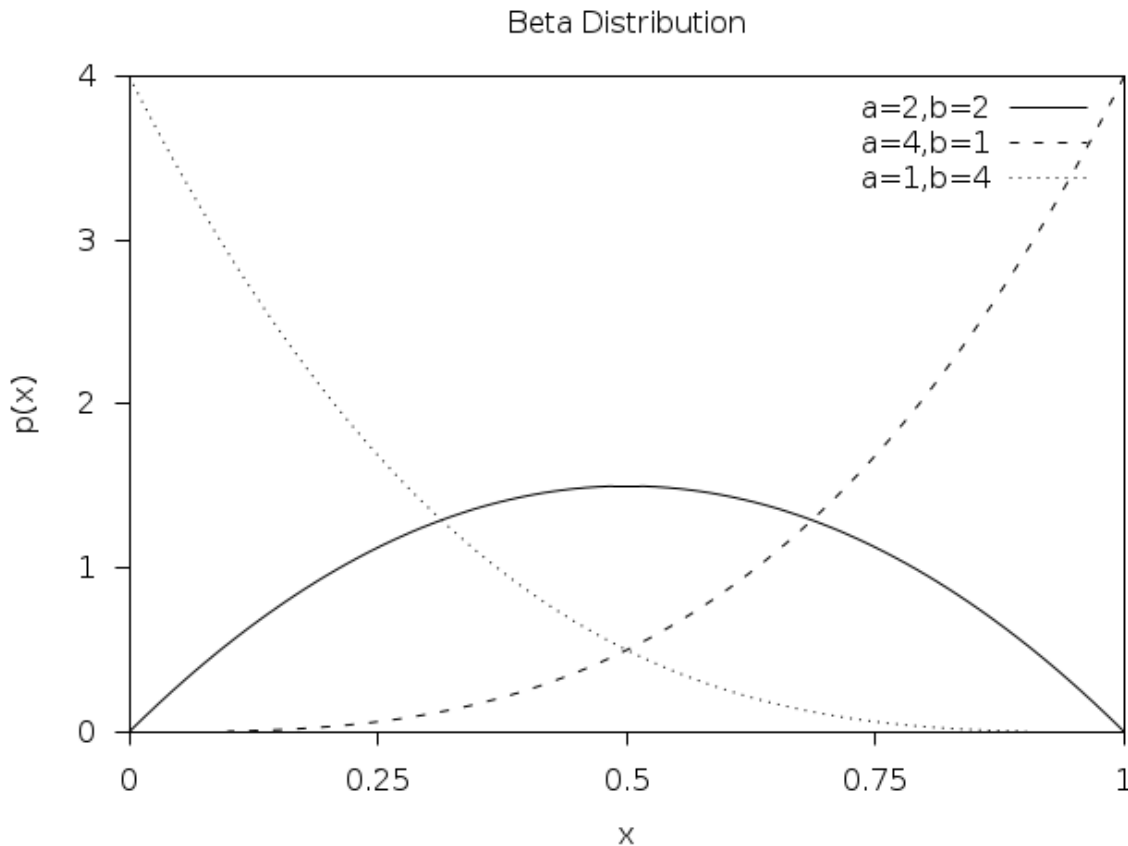
This function returns a random variate from the beta distribution. The distribution function is,

$$p(x)dx = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1} dx$$

for  $0 \leq x \leq 1$ .

double **gsl\_ran\_beta\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for a beta distribution with parameters  $a$  and  $b$ , using the formula given above.



double **gsl\_cdf\_beta\_P**(double x, double a, double b)

double **gsl\_cdf\_beta\_Q**(double x, double a, double b)

double **gsl\_cdf\_beta\_Pinv**(double P, double a, double b)

double **gsl\_cdf\_beta\_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the beta distribution with parameters  $a$  and  $b$ .

## 20.22 The Logistic Distribution

double **gsl\_ran\_logistic**(const *gsl\_rng* \*r, double a)

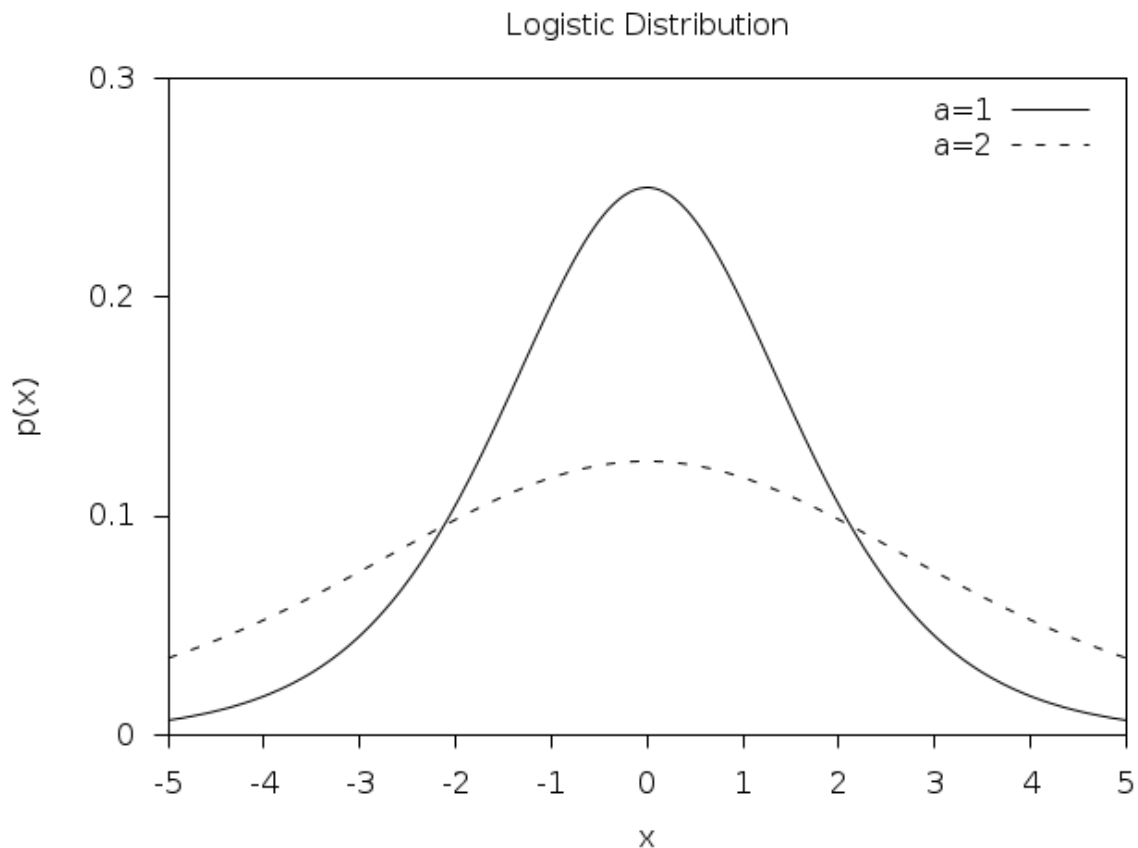
This function returns a random variate from the logistic distribution. The distribution function is,

$$p(x)dx = \frac{\exp(-x/a)}{a(1 + \exp(-x/a))^2} dx$$

for  $-\infty < x < +\infty$ .

double **gsl\_ran\_logistic\_pdf**(double x, double a)

This function computes the probability density  $p(x)$  at  $x$  for a logistic distribution with scale parameter  $a$ , using the formula given above.



double **gsl\_cdf\_logistic\_P**(double x, double a)

double **gsl\_cdf\_logistic\_Q**(double x, double a)

double **gsl\_cdf\_logistic\_Pinv**(double P, double a)

double **gsl\_cdf\_logistic\_Qinv**(double Q, double a)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the logistic distribution with scale parameter  $a$ .

## 20.23 The Pareto Distribution

double **gsl\_ran\_pareto**(const *gsl\_rng* \*r, double a, double b)

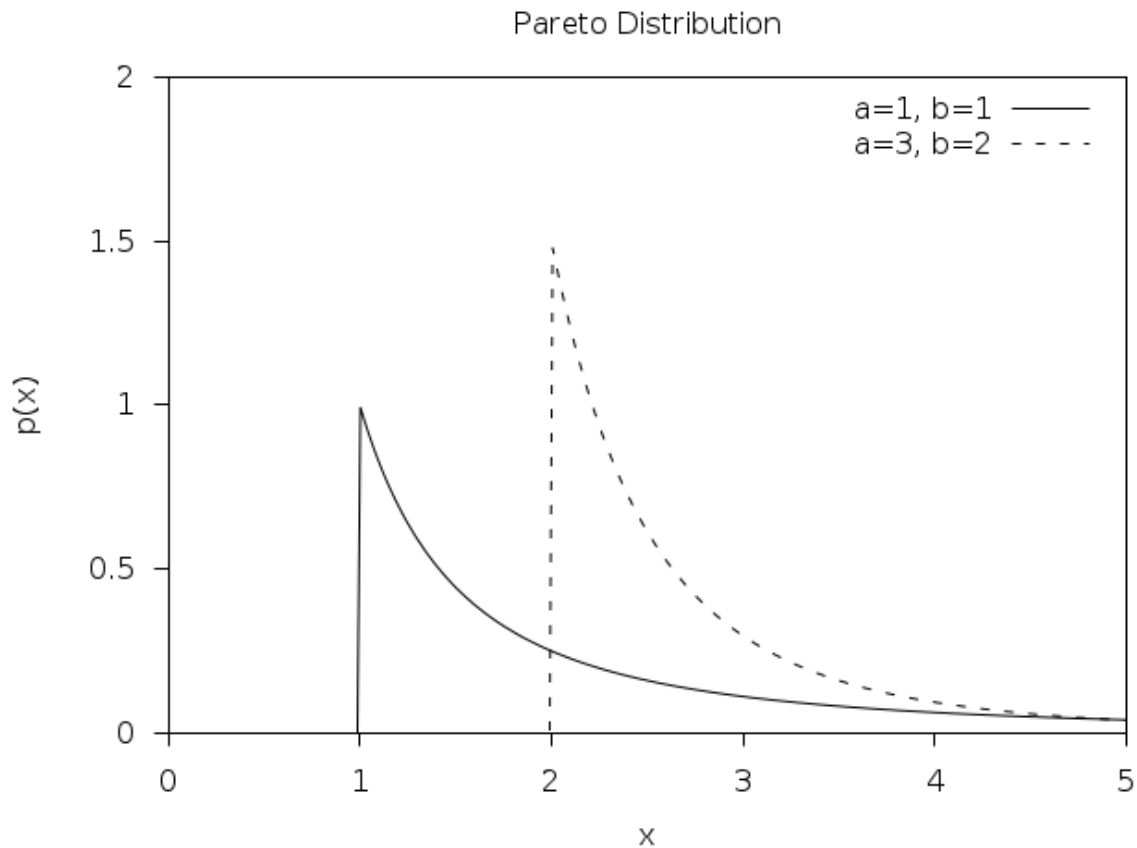
This function returns a random variate from the Pareto distribution of order  $a$ . The distribution function is,

$$p(x)dx = (a/b)/(x/b)^{a+1}dx$$

for  $x \geq b$ .

double **gsl\_ran\_pareto\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for a Pareto distribution with exponent  $a$  and scale  $b$ , using the formula given above.



double **gsl\_cdf\_pareto\_P**(double x, double a, double b)

double **gsl\_cdf\_pareto\_Q**(double x, double a, double b)

double **gsl\_cdf\_pareto\_Pinv**(double P, double a, double b)

double **gsl\_cdf\_pareto\_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Pareto distribution with exponent  $a$  and scale  $b$ .

## 20.24 Spherical Vector Distributions

The spherical distributions generate random vectors, located on a spherical surface. They can be used as random directions, for example in the steps of a random walk.

void **gsl\_ran\_dir\_2d**(const *gsl\_rng* \*r, double \*x, double \*y)

void **gsl\_ran\_dir\_2d\_trig\_method**(const *gsl\_rng* \*r, double \*x, double \*y)

This function returns a random direction vector  $v = (x, y)$  in two dimensions. The vector is normalized such that  $|v|^2 = x^2 + y^2 = 1$ . The obvious way to do this is to take a uniform random number between 0 and  $2\pi$  and let  $x$  and  $y$  be the sine and cosine respectively. Two trig functions would have been expensive in the old days, but with modern hardware implementations, this is sometimes the fastest way to go. This is the case for the Pentium (but not the case for the Sun Sparcstation). One can avoid the trig evaluations by choosing  $x$  and  $y$  in the interior of a unit circle (choose them at random from the interior of the enclosing square, and then reject those that are outside the unit circle), and then dividing by  $\sqrt{x^2 + y^2}$ . A much cleverer approach, attributed to von Neumann (See Knuth, v2, 3rd ed, p140, exercise 23), requires neither trig nor a square root. In this approach,  $u$  and  $v$  are chosen at random from the interior of a unit circle, and then  $x = (u^2 - v^2)/(u^2 + v^2)$  and  $y = 2uv/(u^2 + v^2)$ .

void **gsl\_ran\_dir\_3d**(const *gsl\_rng* \*r, double \*x, double \*y, double \*z)

This function returns a random direction vector  $v = (x, y, z)$  in three dimensions. The vector is normalized such that  $|v|^2 = x^2 + y^2 + z^2 = 1$ . The method employed is due to Robert E. Knop (CACM 13, 326 (1970)), and explained in Knuth, v2, 3rd ed, p136. It uses the surprising fact that the distribution projected along any axis is actually uniform (this is only true for 3 dimensions).

void **gsl\_ran\_dir\_nd**(const *gsl\_rng* \*r, size\_t n, double \*x)

This function returns a random direction vector  $v = (x_1, x_2, \dots, x_n)$  in  $n$  dimensions. The vector is normalized such that  $|v|^2 = x_1^2 + x_2^2 + \dots + x_n^2 = 1$ . The method uses the fact that a multivariate Gaussian distribution is spherically symmetric. Each component is generated to have a Gaussian distribution, and then the components are normalized. The method is described by Knuth, v2, 3rd ed, p135–136, and attributed to G. W. Brown, Modern Mathematics for the Engineer (1956).

## 20.25 The Weibull Distribution

double **gsl\_ran\_weibull**(const *gsl\_rng* \*r, double a, double b)

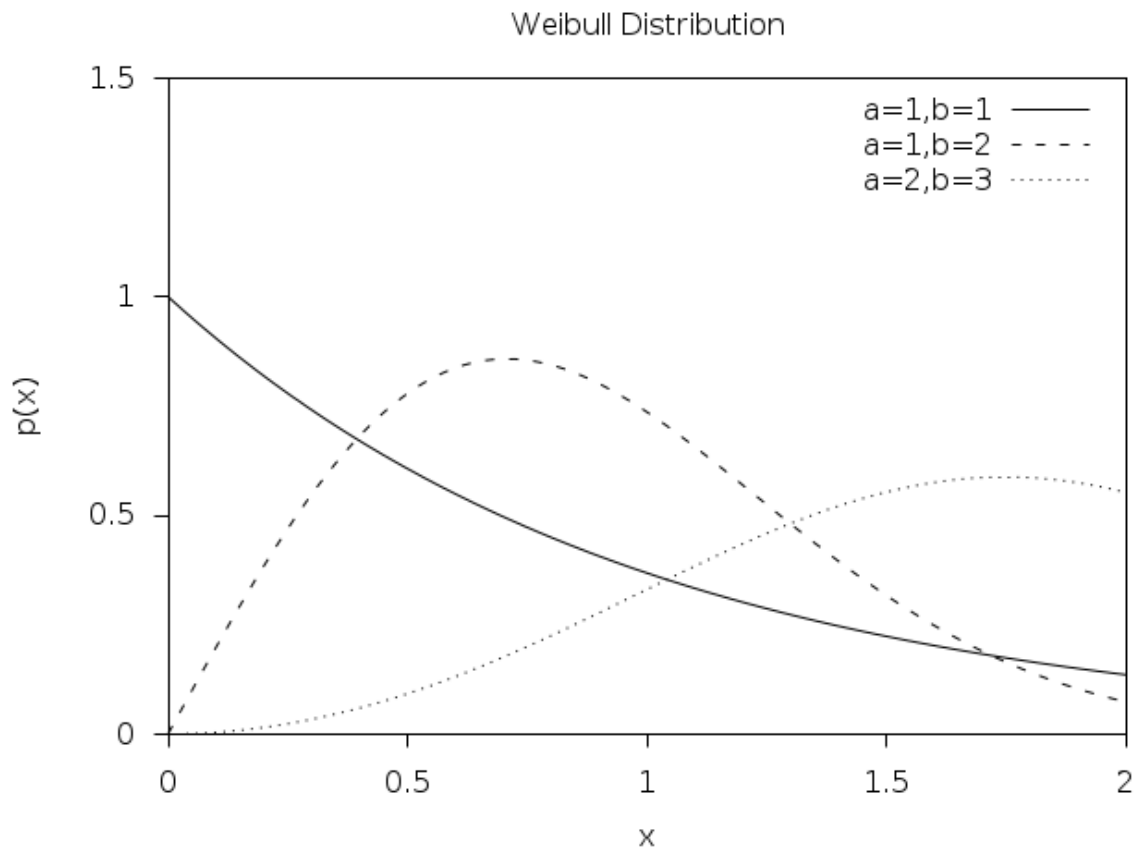
This function returns a random variate from the Weibull distribution. The distribution function is,

$$p(x)dx = \frac{b}{a^b} x^{b-1} \exp(-(x/a)^b) dx$$

for  $x \geq 0$ .

double **gsl\_ran\_weibull\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for a Weibull distribution with scale  $a$  and exponent  $b$ , using the formula given above.



double **gsl\_cdf\_weibull\_P**(double x, double a, double b)

double **gsl\_cdf\_weibull\_Q**(double x, double a, double b)

double **gsl\_cdf\_weibull\_Pinv**(double P, double a, double b)

double **gsl\_cdf\_weibull\_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Weibull distribution with scale  $a$  and exponent  $b$ .



## 20.26 The Type-1 Gumbel Distribution

double **gsl\_ran\_gumbel1**(const *gsl\_rng* \*r, double a, double b)

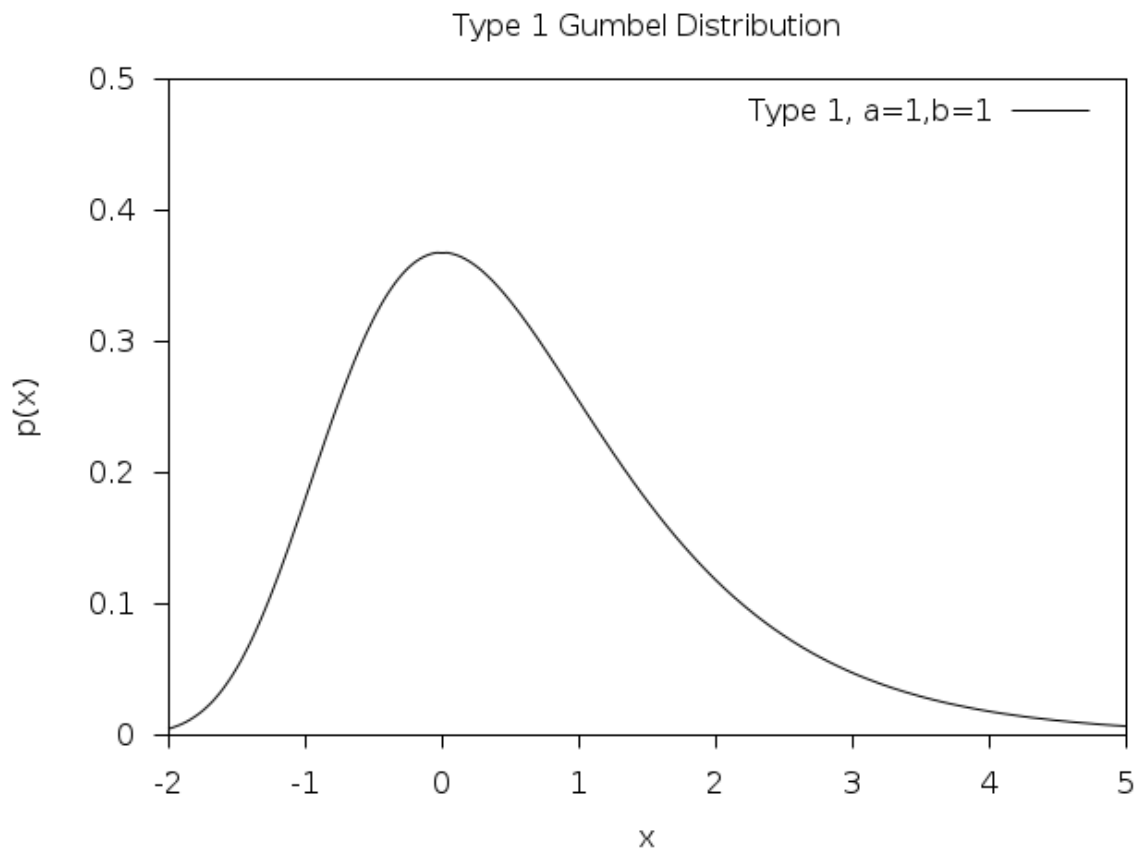
This function returns a random variate from the Type-1 Gumbel distribution. The Type-1 Gumbel distribution function is,

$$p(x)dx = ab \exp(-(b \exp(-ax) + ax))dx$$

for  $-\infty < x < \infty$ .

double **gsl\_ran\_gumbel1\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for a Type-1 Gumbel distribution with parameters  $a$  and  $b$ , using the formula given above.



double **gsl\_cdf\_gumbel1\_P**(double x, double a, double b)

double **gsl\_cdf\_gumbel1\_Q**(double x, double a, double b)

double **gsl\_cdf\_gumbel1\_Pinv**(double P, double a, double b)

double **gsl\_cdf\_gumbel1\_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Type-1 Gumbel distribution with parameters  $a$  and  $b$ .

## 20.27 The Type-2 Gumbel Distribution

double **gsl\_ran\_gumbel2**(const *gsl\_rng* \*r, double a, double b)

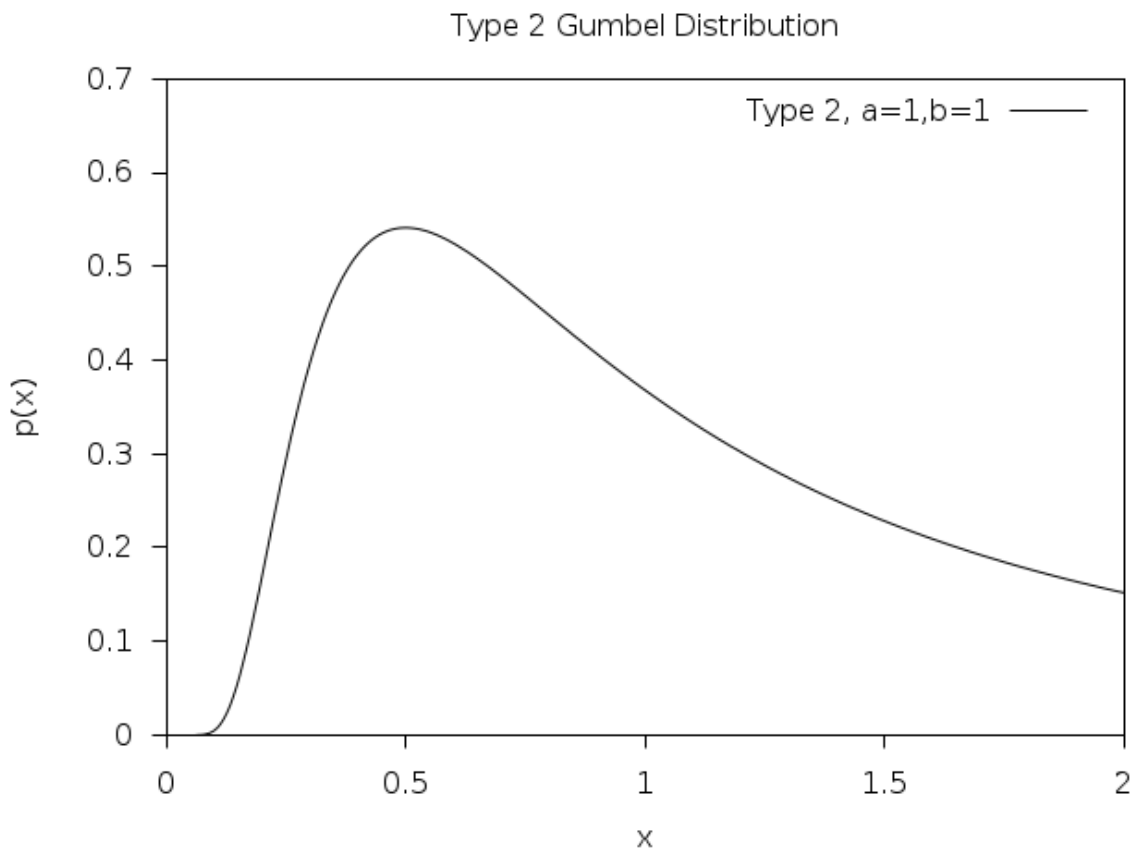
This function returns a random variate from the Type-2 Gumbel distribution. The Type-2 Gumbel distribution function is,

$$p(x)dx = abx^{-a-1} \exp(-bx^{-a})dx$$

for  $0 < x < \infty$ .

double **gsl\_ran\_gumbel2\_pdf**(double x, double a, double b)

This function computes the probability density  $p(x)$  at  $x$  for a Type-2 Gumbel distribution with parameters  $a$  and  $b$ , using the formula given above.



double **gsl\_cdf\_gumbel2\_P**(double x, double a, double b)

double **gsl\_cdf\_gumbel2\_Q**(double x, double a, double b)

double **gsl\_cdf\_gumbel2\_Pinv**(double P, double a, double b)

double **gsl\_cdf\_gumbel2\_Qinv**(double Q, double a, double b)

These functions compute the cumulative distribution functions  $P(x)$ ,  $Q(x)$  and their inverses for the Type-2 Gumbel distribution with parameters  $a$  and  $b$ .

## 20.28 The Dirichlet Distribution

void **gsl\_ran\_dirichlet**(const *gsl\_rng* \*r, size\_t K, const double alpha[], double theta[])

This function returns an array of  $K$  random variates from a Dirichlet distribution of order  $K-1$ . The distribution function is

$$p(\theta_1, \dots, \theta_K) d\theta_1 \cdots d\theta_K = \frac{1}{Z} \prod_{i=1}^K \theta_i^{\alpha_i-1} \delta(1 - \sum_{i=1}^K \theta_i) d\theta_1 \cdots d\theta_K$$

for  $\theta_i \geq 0$  and  $\alpha_i > 0$ . The delta function ensures that  $\sum \theta_i = 1$ . The normalization factor  $Z$  is

$$Z = \frac{\prod_{i=1}^K \Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^K \alpha_i)}$$

The random variates are generated by sampling  $K$  values from gamma distributions with parameters  $a = \alpha_i$ ,  $b = 1$ , and renormalizing. See A.M. Law, W.D. Kelton, *Simulation Modeling and Analysis* (1991).

double **gsl\_ran\_dirichlet\_pdf**(size\_t K, const double alpha[], const double theta[])

This function computes the probability density  $p(\theta_1, \dots, \theta_K)$  at `theta[K]` for a Dirichlet distribution with parameters `alpha[K]`, using the formula given above.

double **gsl\_ran\_dirichlet\_lnpdf**(size\_t K, const double alpha[], const double theta[])

This function computes the logarithm of the probability density  $p(\theta_1, \dots, \theta_K)$  for a Dirichlet distribution with parameters `alpha[K]`.

## 20.29 General Discrete Distributions

Given  $K$  discrete events with different probabilities  $P[k]$ , produce a random value  $k$  consistent with its probability.

The obvious way to do this is to preprocess the probability list by generating a cumulative probability array with  $K + 1$  elements:

$$\begin{aligned}C[0] &= 0 \\C[k + 1] &= C[k] + P[k]\end{aligned}$$

Note that this construction produces  $C[K] = 1$ . Now choose a uniform deviate  $u$  between 0 and 1, and find the value of  $k$  such that  $C[k] \leq u < C[k + 1]$ . Although this in principle requires of order  $\log K$  steps per random number generation, they are fast steps, and if you use something like  $\lfloor uK \rfloor$  as a starting point, you can often do pretty well.

But faster methods have been devised. Again, the idea is to preprocess the probability list, and save the result in some form of lookup table; then the individual calls for a random discrete event can go rapidly. An approach invented by G. Marsaglia (Generating discrete random variables in a computer, *Comm ACM* 6, 37–38 (1963)) is very clever, and readers interested in examples of good algorithm design are directed to this short and well-written paper. Unfortunately, for large  $K$ , Marsaglia's lookup table can be quite large.

A much better approach is due to Alastair J. Walker (An efficient method for generating discrete random variables with general distributions, *ACM Trans on Mathematical Software* 3, 253–256 (1977); see also Knuth, v2, 3rd ed, p120–121,139). This requires two lookup tables, one floating point and one integer, but both only of size  $K$ . After preprocessing, the random numbers are generated in  $O(1)$  time, even for large  $K$ . The preprocessing suggested by Walker requires  $O(K^2)$  effort, but that is not actually necessary, and the implementation provided here only takes  $O(K)$  effort. In general, more preprocessing leads to faster generation of the individual random numbers, but a diminishing return is reached pretty early. Knuth points out that the optimal preprocessing is combinatorially difficult for large  $K$ .

This method can be used to speed up some of the discrete random number generators below, such as the binomial distribution. To use it for something like the Poisson Distribution, a modification would have to be made, since it only takes a finite set of  $K$  outcomes.

type **gsl\_ran\_discrete\_t**

This structure contains the lookup table for the discrete random number generator.

*gsl\_ran\_discrete\_t* \***gsl\_ran\_discrete\_preproc**(size\_t K, const double \*P)

This function returns a pointer to a structure that contains the lookup table for the discrete random number generator. The array  $P$  contains the probabilities of the discrete events; these array elements must all be positive, but they needn't add up to one (so you can think of them more generally as “weights”)—the preprocessor will normalize appropriately. This return value is used as an argument for the *gsl\_ran\_discrete()* function below.

size\_t **gsl\_ran\_discrete**(const *gsl\_rng* \*r, const *gsl\_ran\_discrete\_t* \*g)

After the preprocessor, above, has been called, you use this function to get the discrete random numbers.

double **gsl\_ran\_discrete\_pdf**(size\_t k, const *gsl\_ran\_discrete\_t* \*g)

Returns the probability  $P[k]$  of observing the variable  $k$ . Since  $P[k]$  is not stored as part of the lookup table, it must be recomputed; this computation takes  $O(K)$ , so if  $K$  is large and you care about the original array  $P[k]$  used to create the lookup table, then you should just keep this original array  $P[k]$  around.

void **gsl\_ran\_discrete\_free**(*gsl\_ran\_discrete\_t* \*g)

De-allocates the lookup table pointed to by  $g$ .

## 20.30 The Poisson Distribution

unsigned int **gsl\_ran\_poisson**(const *gsl\_rng* \*r, double mu)

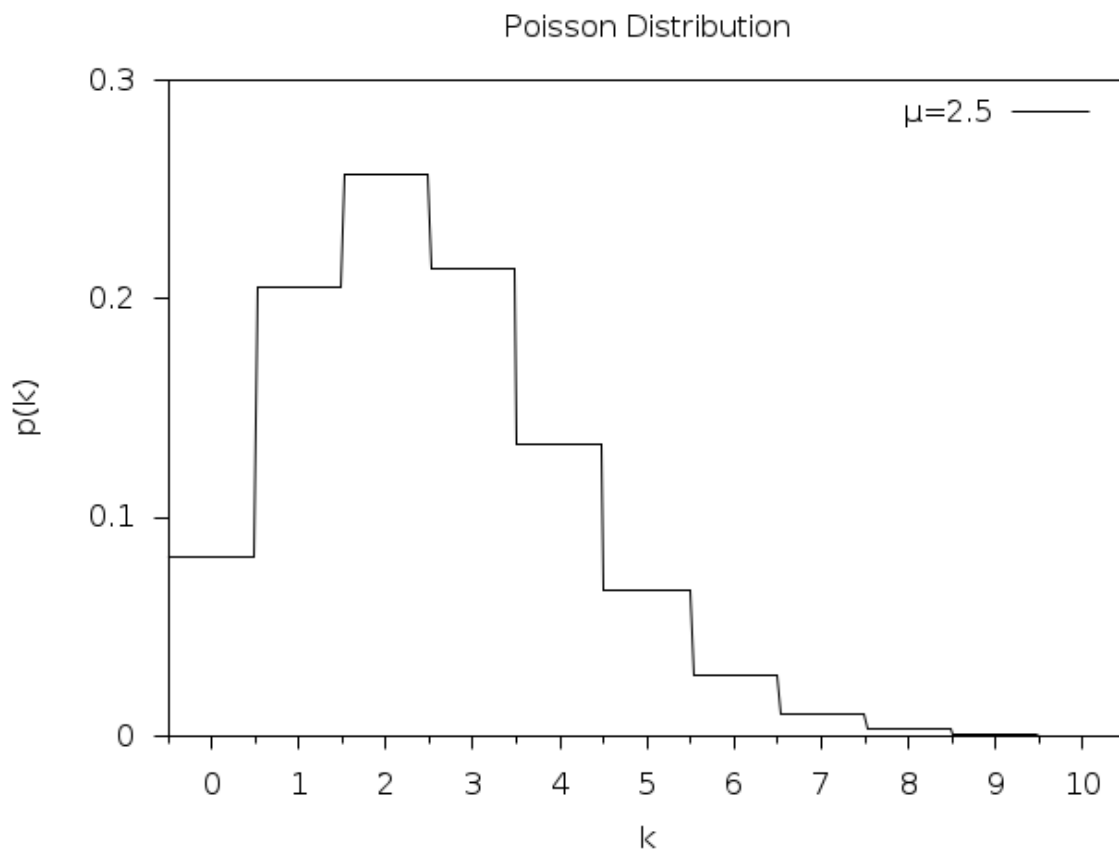
This function returns a random integer from the Poisson distribution with mean  $\mu$ . The probability distribution for Poisson variates is,

$$p(k) = \frac{\mu^k}{k!} \exp(-\mu)$$

for  $k \geq 0$ .

double **gsl\_ran\_poisson\_pdf**(unsigned int k, double mu)

This function computes the probability  $p(k)$  of obtaining  $k$  from a Poisson distribution with mean  $\mu$ , using the formula given above.



double **gsl\_cdf\_poisson\_P**(unsigned int k, double mu)

double **gsl\_cdf\_poisson\_Q**(unsigned int k, double mu)

These functions compute the cumulative distribution functions  $P(k)$ ,  $Q(k)$  for the Poisson distribution with parameter  $\mu$ .

## 20.31 The Bernoulli Distribution

unsigned int **gsl\_ran\_bernoulli**(const *gsl\_rng* \*r, double p)

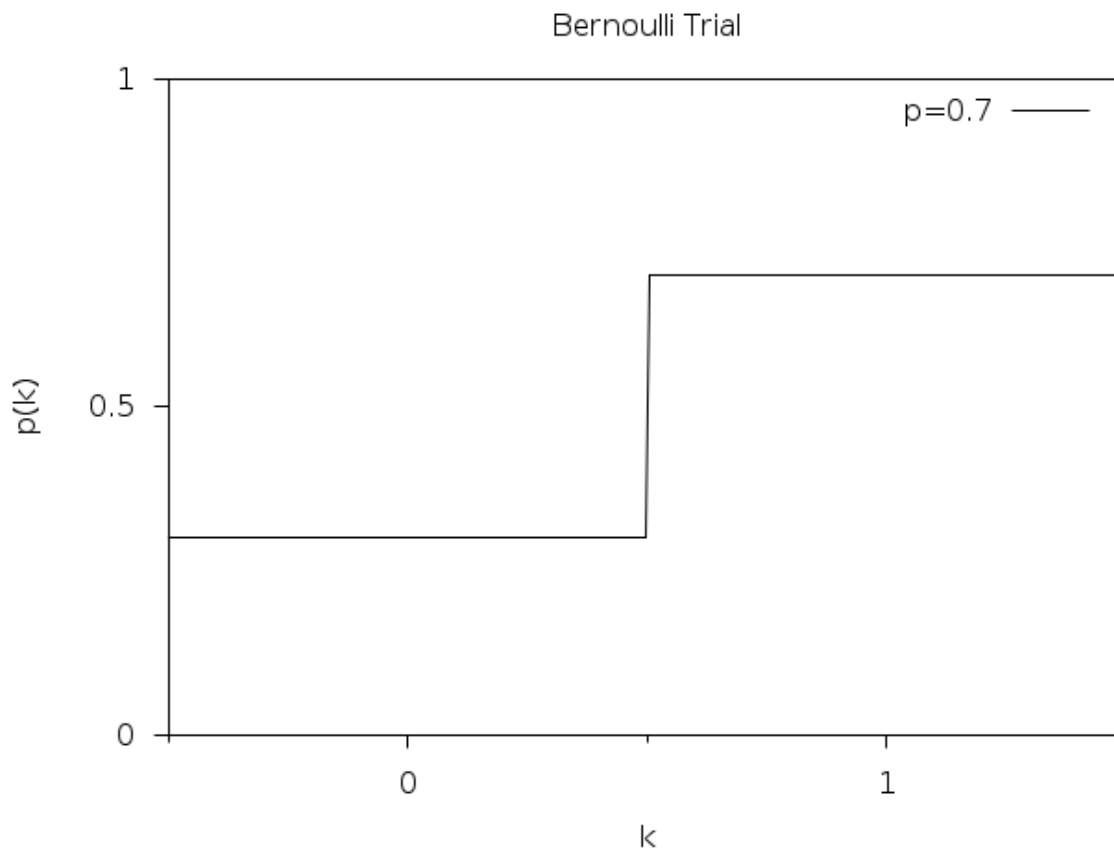
This function returns either 0 or 1, the result of a Bernoulli trial with probability  $p$ . The probability distribution for a Bernoulli trial is,

$$p(0) = 1 - p$$

$$p(1) = p$$

double **gsl\_ran\_bernoulli\_pdf**(unsigned int k, double p)

This function computes the probability  $p(k)$  of obtaining  $k$  from a Bernoulli distribution with probability parameter  $p$ , using the formula given above.



## 20.32 The Binomial Distribution

unsigned int **gsl\_ran\_binomial**(const *gsl\_rng* \*r, double p, unsigned int n)

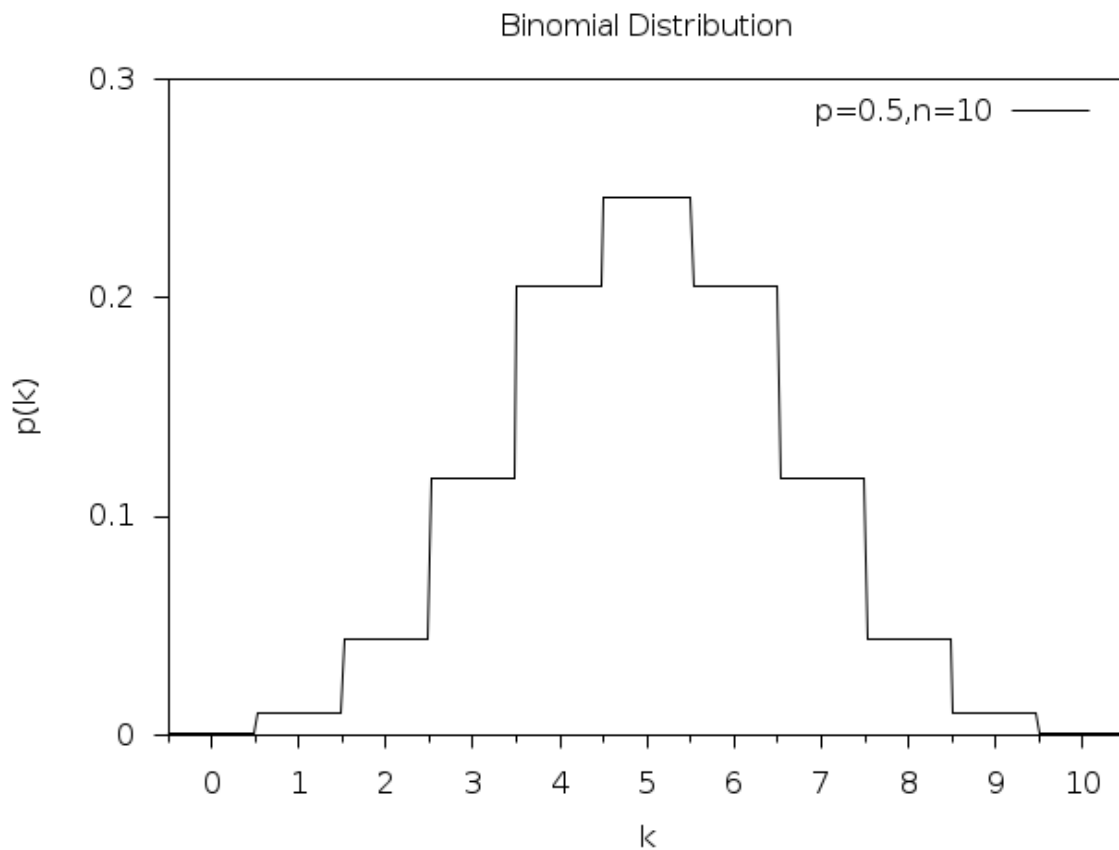
This function returns a random integer from the binomial distribution, the number of successes in  $n$  independent trials with probability  $p$ . The probability distribution for binomial variates is,

$$p(k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

for  $0 \leq k \leq n$ .

double **gsl\_ran\_binomial\_pdf**(unsigned int k, double p, unsigned int n)

This function computes the probability  $p(k)$  of obtaining  $k$  from a binomial distribution with parameters  $p$  and  $n$ , using the formula given above.



double **gsl\_cdf\_binomial\_P**(unsigned int k, double p, unsigned int n)

double **gsl\_cdf\_binomial\_Q**(unsigned int k, double p, unsigned int n)

These functions compute the cumulative distribution functions  $P(k)$ ,  $Q(k)$  for the binomial distribution with parameters  $p$  and  $n$ .

## 20.33 The Multinomial Distribution

void **gsl\_ran\_multinomial**(const *gsl\_rng* \*r, size\_t K, unsigned int N, const double p[], unsigned int n[])

This function computes a random sample  $\mathbf{n}$  from the multinomial distribution formed by  $N$  trials from an underlying distribution  $\mathbf{p}[K]$ . The distribution function for  $\mathbf{n}$  is,

$$P(n_1, n_2, \dots, n_K) = \frac{N!}{n_1! n_2! \dots n_K!} p_1^{n_1} p_2^{n_2} \dots p_K^{n_K}$$

where  $(n_1, n_2, \dots, n_K)$  are nonnegative integers with  $\sum_{k=1}^K n_k = N$ , and  $(p_1, p_2, \dots, p_K)$  is a probability distribution with  $\sum p_i = 1$ . If the array  $\mathbf{p}[K]$  is not normalized then its entries will be treated as weights and normalized appropriately. The arrays  $\mathbf{n}$  and  $\mathbf{p}$  must both be of length  $K$ .

Random variates are generated using the conditional binomial method (see C.S. Davis, *The computer generation of multinomial random variates*, Comp. Stat. Data Anal. 16 (1993) 205–217 for details).

double **gsl\_ran\_multinomial\_pdf**(size\_t K, const double p[], const unsigned int n[])

This function computes the probability  $P(n_1, n_2, \dots, n_K)$  of sampling  $\mathbf{n}[K]$  from a multinomial distribution with parameters  $\mathbf{p}[K]$ , using the formula given above.

double **gsl\_ran\_multinomial\_lnpdf**(size\_t K, const double p[], const unsigned int n[])

This function returns the logarithm of the probability for the multinomial distribution  $P(n_1, n_2, \dots, n_K)$  with parameters  $\mathbf{p}[K]$ .



## 20.34 The Negative Binomial Distribution

unsigned int **gsl\_ran\_negative\_binomial**(const *gsl\_rng* \*r, double p, double n)

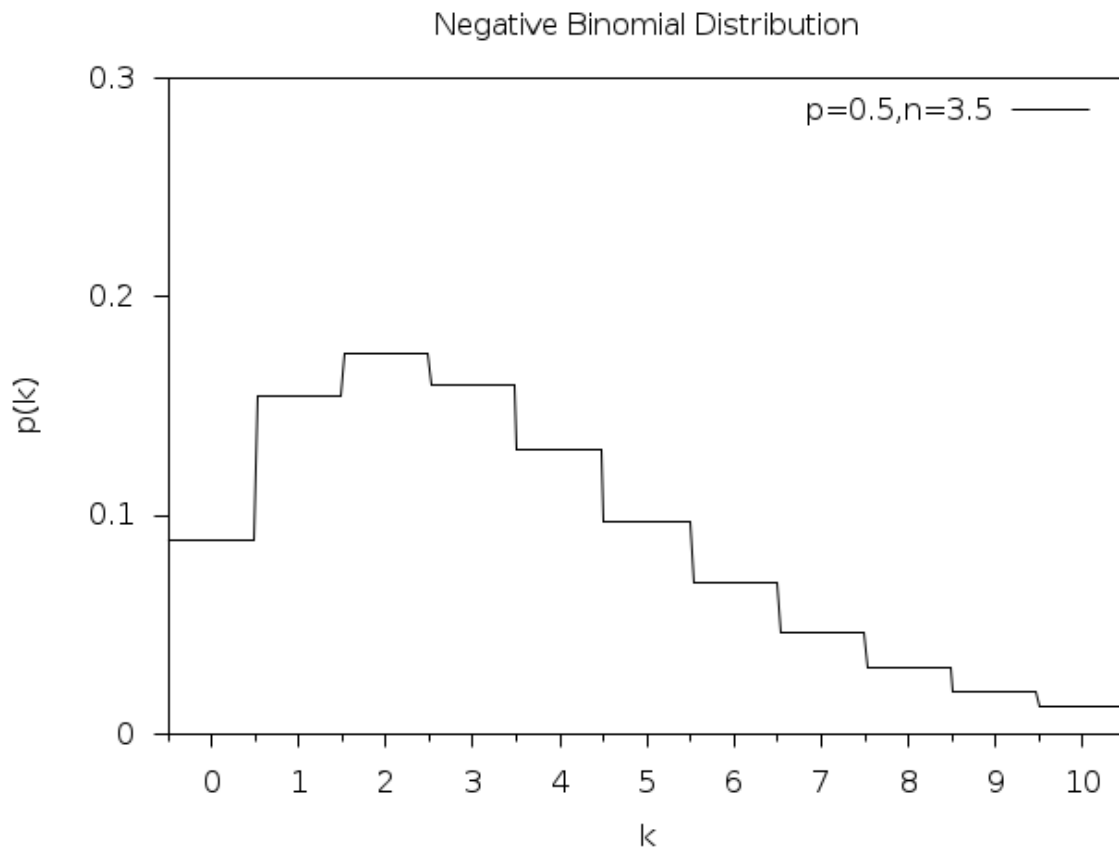
This function returns a random integer from the negative binomial distribution, the number of failures occurring before  $n$  successes in independent trials with probability  $p$  of success. The probability distribution for negative binomial variates is,

$$p(k) = \frac{\Gamma(n+k)}{\Gamma(k+1)\Gamma(n)} p^n (1-p)^k$$

Note that  $n$  is not required to be an integer.

double **gsl\_ran\_negative\_binomial\_pdf**(unsigned int k, double p, double n)

This function computes the probability  $p(k)$  of obtaining  $k$  from a negative binomial distribution with parameters  $p$  and  $n$ , using the formula given above.



double **gsl\_cdf\_negative\_binomial\_P**(unsigned int k, double p, double n)

double **gsl\_cdf\_negative\_binomial\_Q**(unsigned int k, double p, double n)

These functions compute the cumulative distribution functions  $P(k)$ ,  $Q(k)$  for the negative binomial distribution with parameters  $p$  and  $n$ .

## 20.35 The Pascal Distribution

unsigned int **gsl\_ran\_pascal**(const *gsl\_rng* \*r, double p, unsigned int n)

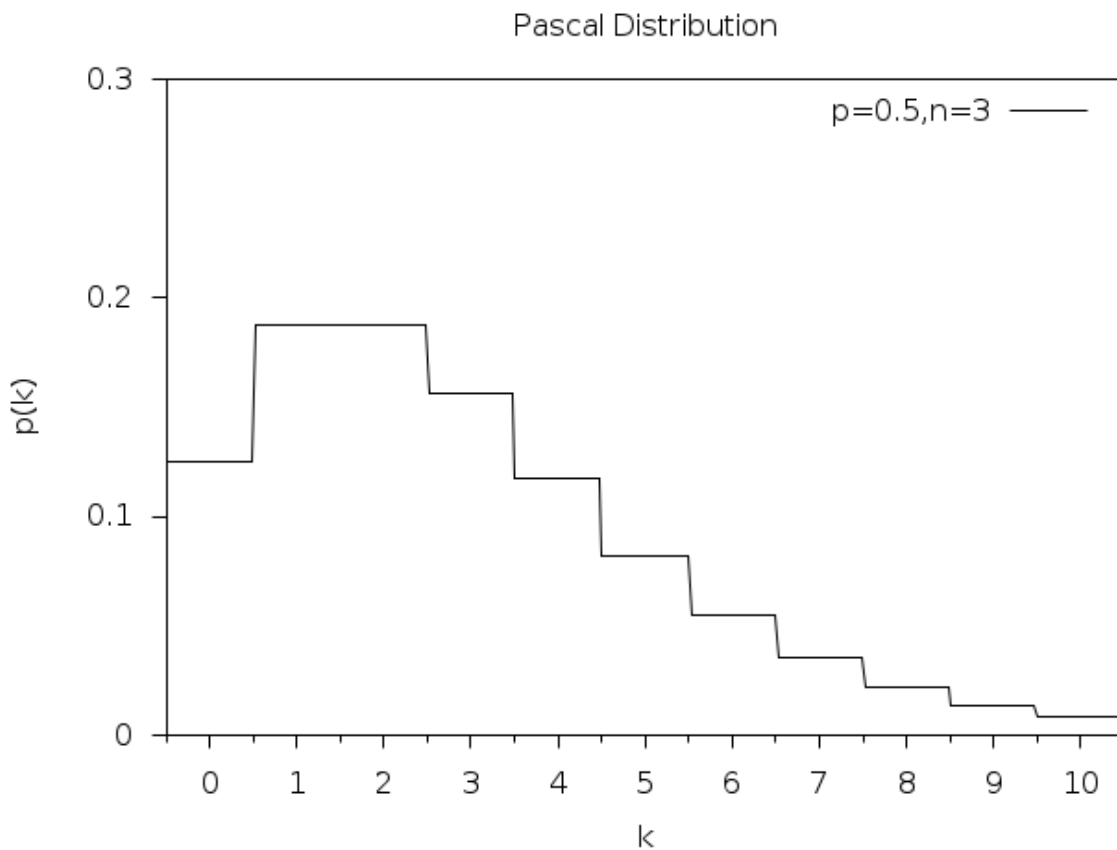
This function returns a random integer from the Pascal distribution. The Pascal distribution is simply a negative binomial distribution with an integer value of  $n$ .

$$p(k) = \frac{(n+k-1)!}{k!(n-1)!} p^n (1-p)^k$$

for  $k \geq 0$ .

double **gsl\_ran\_pascal\_pdf**(unsigned int k, double p, unsigned int n)

This function computes the probability  $p(k)$  of obtaining  $k$  from a Pascal distribution with parameters  $p$  and  $n$ , using the formula given above.



double **gsl\_cdf\_pascal\_P**(unsigned int k, double p, unsigned int n)

double **gsl\_cdf\_pascal\_Q**(unsigned int k, double p, unsigned int n)

These functions compute the cumulative distribution functions  $P(k)$ ,  $Q(k)$  for the Pascal distribution with parameters  $p$  and  $n$ .

## 20.36 The Geometric Distribution

unsigned int **gsl\_ran\_geometric**(const *gsl\_rng* \*r, double p)

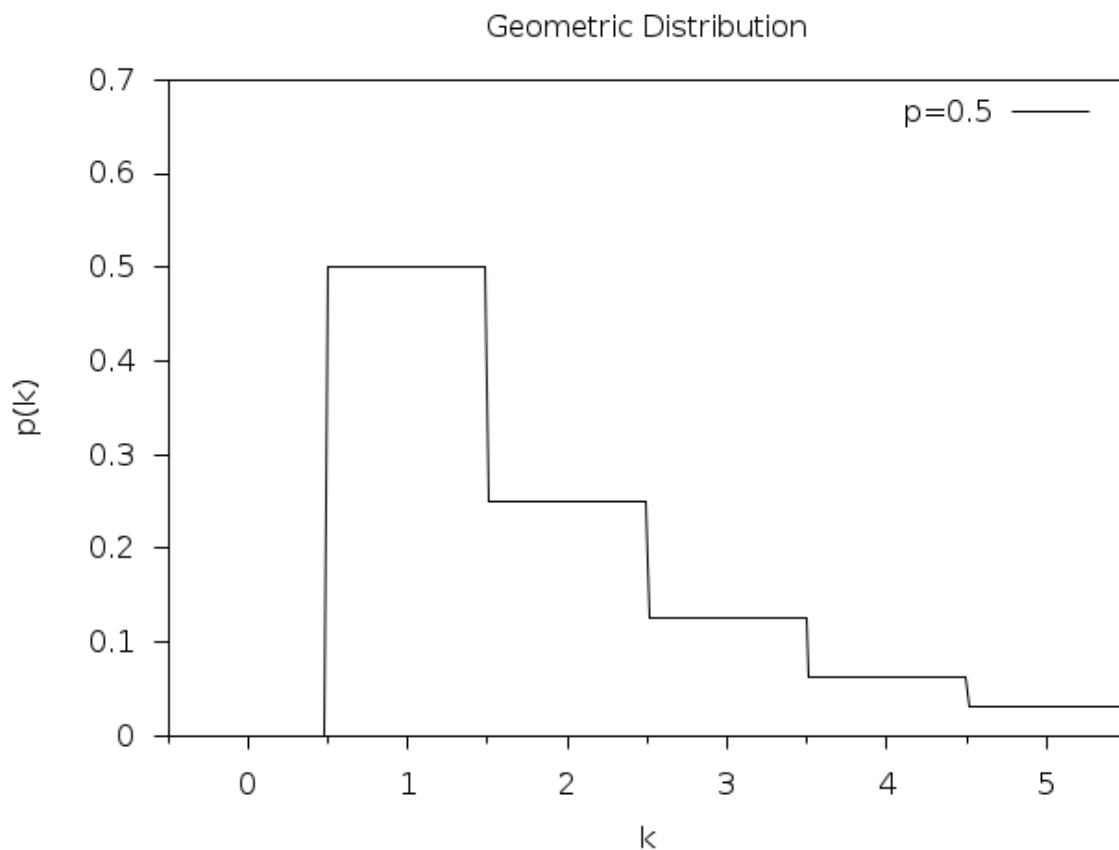
This function returns a random integer from the geometric distribution, the number of independent trials with probability  $p$  until the first success. The probability distribution for geometric variates is,

$$p(k) = p(1 - p)^{k-1}$$

for  $k \geq 1$ . Note that the distribution begins with  $k = 1$  with this definition. There is another convention in which the exponent  $k - 1$  is replaced by  $k$ .

double **gsl\_ran\_geometric\_pdf**(unsigned int k, double p)

This function computes the probability  $p(k)$  of obtaining  $k$  from a geometric distribution with probability parameter  $p$ , using the formula given above.



double **gsl\_cdf\_geometric\_P**(unsigned int k, double p)

double **gsl\_cdf\_geometric\_Q**(unsigned int k, double p)

These functions compute the cumulative distribution functions  $P(k)$ ,  $Q(k)$  for the geometric distribution with parameter  $p$ .

## 20.37 The Hypergeometric Distribution

unsigned int **gsl\_ran\_hypergeometric**(const *gsl\_rng* \*r, unsigned int n1, unsigned int n2, unsigned int t)

This function returns a random integer from the hypergeometric distribution. The probability distribution for hypergeometric random variates is,

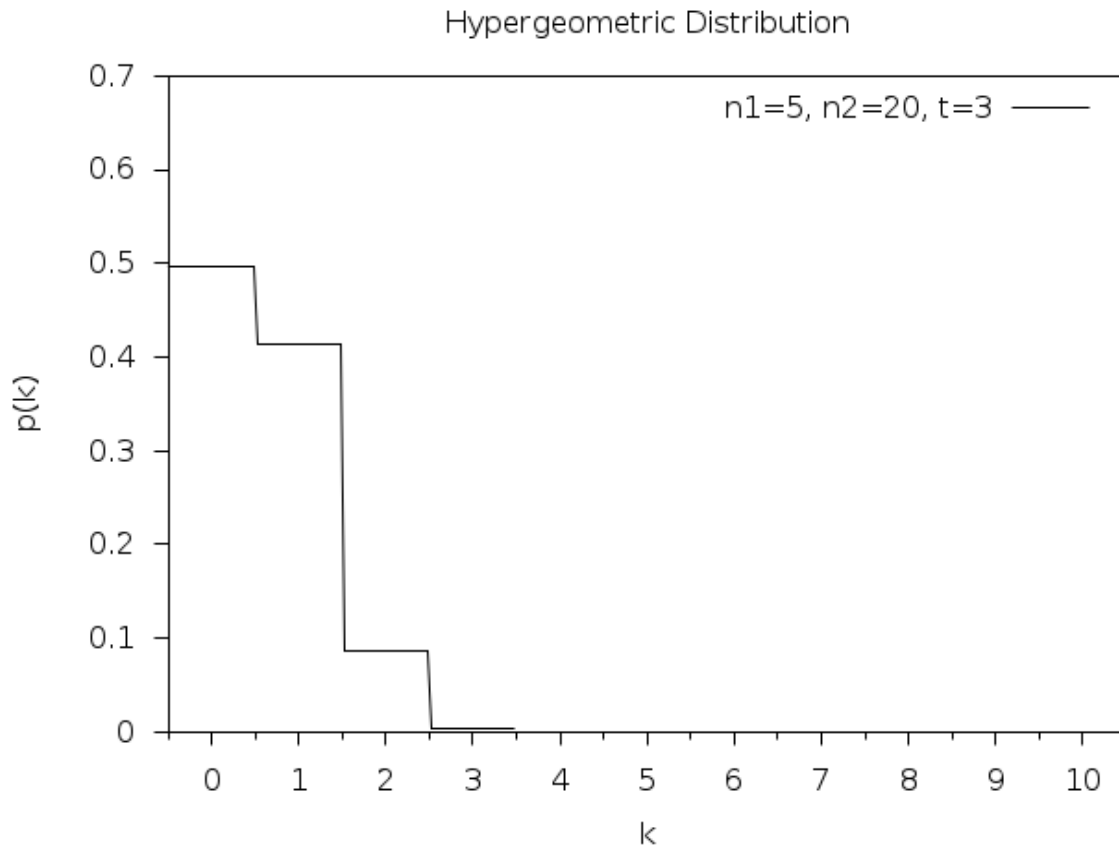
$$p(k) = C(n_1, k)C(n_2, t - k)/C(n_1 + n_2, t)$$

where  $C(a, b) = a!/(b!(a - b)!)$  and  $t \leq n_1 + n_2$ . The domain of  $k$  is  $\max(0, t - n_2), \dots, \min(t, n_1)$

If a population contains  $n_1$  elements of “type 1” and  $n_2$  elements of “type 2” then the hypergeometric distribution gives the probability of obtaining  $k$  elements of “type 1” in  $t$  samples from the population without replacement.

double **gsl\_ran\_hypergeometric\_pdf**(unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

This function computes the probability  $p(k)$  of obtaining  $k$  from a hypergeometric distribution with parameters  $n_1, n_2, t$ , using the formula given above.



double **gsl\_cdf\_hypergeometric\_P**(unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

double **gsl\_cdf\_hypergeometric\_Q**(unsigned int k, unsigned int n1, unsigned int n2, unsigned int t)

These functions compute the cumulative distribution functions  $P(k)$ ,  $Q(k)$  for the hypergeometric distribution with parameters  $n_1, n_2$  and  $t$ .

## 20.38 The Logarithmic Distribution

unsigned int **gsl\_ran\_logarithmic**(const *gsl\_rng* \*r, double p)

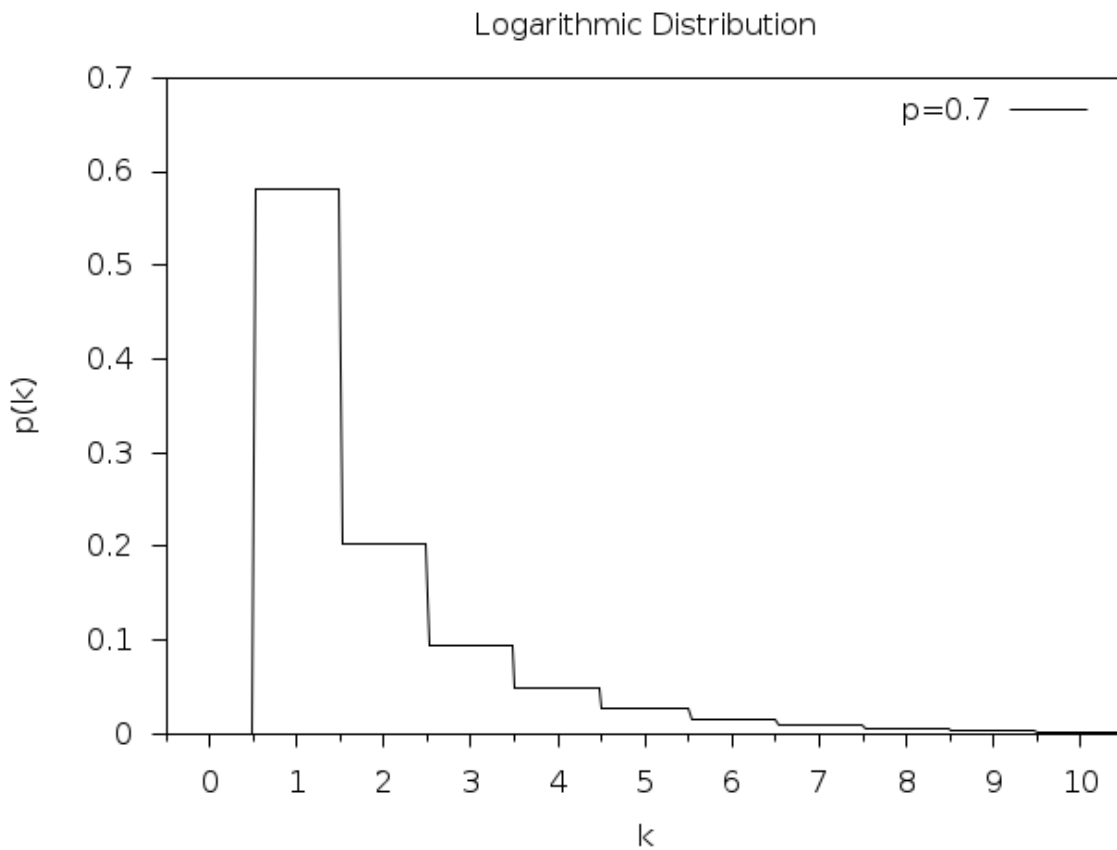
This function returns a random integer from the logarithmic distribution. The probability distribution for logarithmic random variates is,

$$p(k) = \frac{-1}{\log(1-p)} \left( \frac{p^k}{k} \right)$$

for  $k \geq 1$ .

double **gsl\_ran\_logarithmic\_pdf**(unsigned int k, double p)

This function computes the probability  $p(k)$  of obtaining  $k$  from a logarithmic distribution with probability parameter  $p$ , using the formula given above.



## 20.39 The Wishart Distribution

int **gsl\_ran\_wishart**(const *gsl\_rng* \*r, const double n, const *gsl\_matrix* \*L, *gsl\_matrix* \*result, *gsl\_matrix* \*work)

This function computes a random symmetric  $p$ -by- $p$  matrix from the Wishart distribution. The probability distribution for Wishart random variates is,

$$p(X) = \frac{|X|^{(n-p-1)/2} e^{-\text{tr}(V^{-1}X)/2}}{2^{\frac{np}{2}} |V|^{n/2} \Gamma_p(\frac{n}{2})}$$

Here,  $n > p - 1$  is the number of degrees of freedom,  $V$  is a symmetric positive definite  $p$ -by- $p$  scale matrix, whose Cholesky factor is specified by  $L$ , and *work* is  $p$ -by- $p$  workspace. The  $p$ -by- $p$  Wishart distributed matrix  $X$  is stored in *result* on output.

int **gsl\_ran\_wishart\_pdf**(const *gsl\_matrix* \*X, const *gsl\_matrix* \*L\_X, const double n, const *gsl\_matrix* \*L, double \*result, *gsl\_matrix* \*work)

int **gsl\_ran\_wishart\_log\_pdf**(const *gsl\_matrix* \*X, const *gsl\_matrix* \*L\_X, const double n, const *gsl\_matrix* \*L, double \*result, *gsl\_matrix* \*work)

These functions compute  $p(X)$  or  $\log p(X)$  for the  $p$ -by- $p$  matrix  $X$ , whose Cholesky factor is specified in  $L_X$ . The degrees of freedom is given by  $n$ , the Cholesky factor of the scale matrix  $V$  is specified in  $L$ , and *work* is  $p$ -by- $p$  workspace. The probably density value is returned in *result*.

## 20.40 Shuffling and Sampling

The following functions allow the shuffling and sampling of a set of objects. The algorithms rely on a random number generator as a source of randomness and a poor quality generator can lead to correlations in the output. In particular it is important to avoid generators with a short period. For more information see Knuth, v2, 3rd ed, Section 3.4.2, “Random Sampling and Shuffling”.

void `gsl_ran_shuffle`(const *gsl\_rng* \*r, void \*base, size\_t n, size\_t size)

This function randomly shuffles the order of *n* objects, each of size *size*, stored in the array `base[0..n-1]`. The output of the random number generator *r* is used to produce the permutation. The algorithm generates all possible *n!* permutations with equal probability, assuming a perfect source of random numbers.

The following code shows how to shuffle the numbers from 0 to 51:

```
int a[52];

for (i = 0; i < 52; i++)
{
    a[i] = i;
}

gsl_ran_shuffle (r, a, 52, sizeof (int));
```

int `gsl_ran_choose`(const *gsl\_rng* \*r, void \*dest, size\_t k, void \*src, size\_t n, size\_t size)

This function fills the array `dest[k]` with *k* objects taken randomly from the *n* elements of the array `src[0..n-1]`. The objects are each of size *size*. The output of the random number generator *r* is used to make the selection. The algorithm ensures all possible samples are equally likely, assuming a perfect source of randomness.

The objects are sampled **without** replacement, thus each object can only appear once in *dest*. It is required that *k* be less than or equal to *n*. The objects in *dest* will be in the same relative order as those in *src*. You will need to call `gsl_ran_shuffle(r, dest, n, size)` if you want to randomize the order.

The following code shows how to select a random sample of three unique numbers from the set 0 to 99:

```
double a[3], b[100];

for (i = 0; i < 100; i++)
{
    b[i] = (double) i;
}

gsl_ran_choose (r, a, 3, b, 100, sizeof (double));
```

void `gsl_ran_sample`(const *gsl\_rng* \*r, void \*dest, size\_t k, void \*src, size\_t n, size\_t size)

This function is like `gsl_ran_choose()` but samples *k* items from the original array of *n* items *src* with replacement, so the same object can appear more than once in the output sequence *dest*. There is no requirement that *k* be less than *n* in this case.

## 20.41 Examples

The following program demonstrates the use of a random number generator to produce variates from a distribution. It prints 10 samples from the Poisson distribution with a mean of 3.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;

  int i, n = 10;
  double mu = 3.0;

  /* create a generator chosen by the
     environment variable GSL_RNG_TYPE */

  gsl_rng_env_setup();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  /* print n random variates chosen from
     the poisson distribution with mean
     parameter mu */

  for (i = 0; i < n; i++)
  {
    unsigned int k = gsl_ran_poisson (r, mu);
    printf (" %u", k);
  }

  printf ("\n");
  gsl_rng_free (r);
  return 0;
}
```

If the library and header files are installed under `/usr/local` (the default location) then the program can be compiled with these options:

```
$ gcc -Wall demo.c -lgsl -lgslcblas -lm
```

Here is the output of the program,

```
2 5 5 2 1 0 3 4 1 1
```

The variates depend on the seed used by the generator. The seed for the default generator type `gsl_rng_default` can be changed with the `GSL_RNG_SEED` environment variable to produce a different stream of variates:



```
$ GSL_RNG_SEED=123 ./a.out
```

giving output

```
4 5 6 3 3 1 4 2 5 5
```

The following program generates a random walk in two dimensions.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

int
main (void)
{
  int i;
  double x = 0, y = 0, dx, dy;

  const gsl_rng_type * T;
  gsl_rng * r;

  gsl_rng_env_setup();
  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  printf ("%g %g\n", x, y);

  for (i = 0; i < 10; i++)
    {
      gsl_ran_dir_2d (r, &dx, &dy);
      x += dx; y += dy;
      printf ("%g %g\n", x, y);
    }

  gsl_rng_free (r);
  return 0;
}
```

Fig. 20.1 shows the output from the program.

The following program computes the upper and lower cumulative distribution functions for the standard normal distribution at  $x = 2$ .

```
#include <stdio.h>
#include <gsl/gsl_cdf.h>

int
main (void)
{
  double P, Q;
  double x = 2.0;

  P = gsl_cdf_ugaussian_P (x);
  printf ("prob(x < %f) = %f\n", x, P);
}
```

(continues on next page)

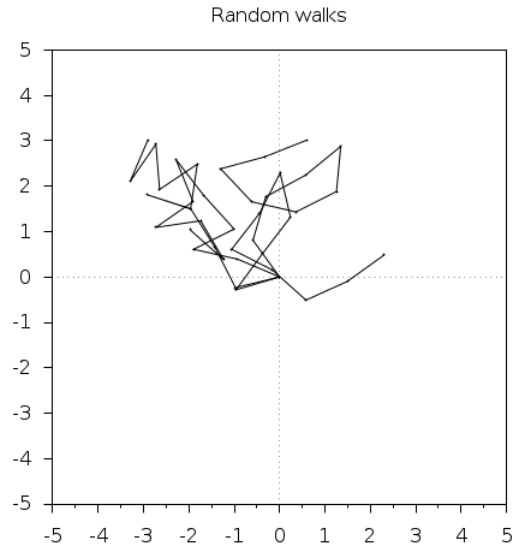


Fig. 20.1: Four 10-step random walks from the origin.

(continued from previous page)

```

Q = gsl_cdf_ugaussian_Q (x);
printf ("prob(x > %f) = %f\n", x, Q);

x = gsl_cdf_ugaussian_Pinv (P);
printf ("Pinv(%f) = %f\n", P, x);

x = gsl_cdf_ugaussian_Qinv (Q);
printf ("Qinv(%f) = %f\n", Q, x);

return 0;
}

```

Here is the output of the program,

```

prob(x < 2.000000) = 0.977250
prob(x > 2.000000) = 0.022750
Pinv(0.977250) = 2.000000
Qinv(0.022750) = 2.000000

```

## 20.42 References and Further Reading

For an encyclopaedic coverage of the subject readers are advised to consult the book “Non-Uniform Random Variate Generation” by Luc Devroye. It covers every imaginable distribution and provides hundreds of algorithms.

- Luc Devroye, “Non-Uniform Random Variate Generation”, Springer-Verlag, ISBN 0-387-96305-7. Available online at <http://cg.scs.carleton.ca/~luc/rnbookindex.html>.

The subject of random variate generation is also reviewed by Knuth, who describes algorithms for all the major distributions.

- Donald E. Knuth, “The Art of Computer Programming: Seminumerical Algorithms” (Vol 2, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896842.

The Particle Data Group provides a short review of techniques for generating distributions of random numbers in the “Monte Carlo” section of its Annual Review of Particle Physics.

- Review of Particle Properties, R.M. Barnett et al., Physical Review D54, 1 (1996) <http://pdg.lbl.gov/>.

The Review of Particle Physics is available online in postscript and pdf format.

An overview of methods used to compute cumulative distribution functions can be found in *Statistical Computing* by W.J. Kennedy and J.E. Gentle. Another general reference is *Elements of Statistical Computing* by R.A. Thisted.

- William E. Kennedy and James E. Gentle, *Statistical Computing* (1980), Marcel Dekker, ISBN 0-8247-6898-1.
- Ronald A. Thisted, *Elements of Statistical Computing* (1988), Chapman & Hall, ISBN 0-412-01371-1.

The cumulative distribution functions for the Gaussian distribution are based on the following papers,

- Rational Chebyshev Approximations Using Linear Equations, W.J. Cody, W. Fraser, J.F. Hart. *Numerische Mathematik* 12, 242–251 (1968).
- Rational Chebyshev Approximations for the Error Function, W.J. Cody. *Mathematics of Computation* 23, n107, 631–637 (July 1969).



## STATISTICS

This chapter describes the statistical functions in the library. The basic statistical functions include routines to compute the mean, variance and standard deviation. More advanced functions allow you to calculate absolute deviations, skewness, and kurtosis as well as the median and arbitrary percentiles. The algorithms use recurrence relations to compute average quantities in a stable way, without large intermediate values that might overflow.

The functions are available in versions for datasets in the standard floating-point and integer types. The versions for double precision floating-point data have the prefix `gsl_stats` and are declared in the header file `gsl_statistics_double.h`. The versions for integer data have the prefix `gsl_stats_int` and are declared in the header file `gsl_statistics_int.h`. All the functions operate on C arrays with a `stride` parameter specifying the spacing between elements.

### 21.1 Mean, Standard Deviation and Variance

double **gsl\_stats\_mean**(const double data[], size\_t stride, size\_t n)

This function returns the arithmetic mean of *data*, a dataset of length *n* with stride *stride*. The arithmetic mean, or *sample mean*, is denoted by  $\hat{\mu}$  and defined as,

$$\hat{\mu} = \frac{1}{N} \sum x_i$$

where  $x_i$  are the elements of the dataset *data*. For samples drawn from a gaussian distribution the variance of  $\hat{\mu}$  is  $\sigma^2/N$ .

double **gsl\_stats\_variance**(const double data[], size\_t stride, size\_t n)

This function returns the estimated, or *sample*, variance of *data*, a dataset of length *n* with stride *stride*. The estimated variance is denoted by  $\hat{\sigma}^2$  and is defined by,

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - \hat{\mu})^2$$

where  $x_i$  are the elements of the dataset *data*. Note that the normalization factor of  $1/(N-1)$  results from the derivation of  $\hat{\sigma}^2$  as an unbiased estimator of the population variance  $\sigma^2$ . For samples drawn from a Gaussian distribution the variance of  $\hat{\sigma}^2$  itself is  $2\sigma^4/N$ .

This function computes the mean via a call to `gsl_stats_mean()`. If you have already computed the mean then you can pass it directly to `gsl_stats_variance_m()`.

double **gsl\_stats\_variance\_m**(const double data[], size\_t stride, size\_t n, double mean)

This function returns the sample variance of *data* relative to the given value of *mean*. The function is computed with  $\hat{\mu}$  replaced by the value of *mean* that you supply,

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - mean)^2$$

double **gsl\_stats\_sd**(const double data[], size\_t stride, size\_t n)

double **gsl\_stats\_sd\_m**(const double data[], size\_t stride, size\_t n, double mean)

The standard deviation is defined as the square root of the variance. These functions return the square root of the corresponding variance functions above.

double **gsl\_stats\_tss**(const double data[], size\_t stride, size\_t n)

double **gsl\_stats\_tss\_m**(const double data[], size\_t stride, size\_t n, double mean)

These functions return the total sum of squares (TSS) of *data* about the mean. For *gsl\_stats\_tss\_m()* the user-supplied value of *mean* is used, and for *gsl\_stats\_tss()* it is computed using *gsl\_stats\_mean()*.

$$\text{TSS} = \sum (x_i - \text{mean})^2$$

double **gsl\_stats\_variance\_with\_fixed\_mean**(const double data[], size\_t stride, size\_t n, double mean)

This function computes an unbiased estimate of the variance of *data* when the population mean *mean* of the underlying distribution is known *a priori*. In this case the estimator for the variance uses the factor  $1/N$  and the sample mean  $\hat{\mu}$  is replaced by the known population mean  $\mu$ ,

$$\hat{\sigma}^2 = \frac{1}{N} \sum (x_i - \mu)^2$$

double **gsl\_stats\_sd\_with\_fixed\_mean**(const double data[], size\_t stride, size\_t n, double mean)

This function calculates the standard deviation of *data* for a fixed population mean *mean*. The result is the square root of the corresponding variance function.

## 21.2 Absolute deviation

double **gsl\_stats\_absdev**(const double data[], size\_t stride, size\_t n)

This function computes the absolute deviation from the mean of *data*, a dataset of length *n* with stride *stride*. The absolute deviation from the mean is defined as,

$$\text{absdev} = \frac{1}{N} \sum |x_i - \hat{\mu}|$$

where  $x_i$  are the elements of the dataset *data*. The absolute deviation from the mean provides a more robust measure of the width of a distribution than the variance. This function computes the mean of *data* via a call to *gsl\_stats\_mean()*.

double **gsl\_stats\_absdev\_m**(const double data[], size\_t stride, size\_t n, double mean)

This function computes the absolute deviation of the dataset *data* relative to the given value of *mean*,

$$\text{absdev} = \frac{1}{N} \sum |x_i - \text{mean}|$$

This function is useful if you have already computed the mean of *data* (and want to avoid recomputing it), or wish to calculate the absolute deviation relative to another value (such as zero, or the median).

## 21.3 Higher moments (skewness and kurtosis)

double **gsl\_stats\_skew**(const double data[], size\_t stride, size\_t n)

This function computes the skewness of *data*, a dataset of length *n* with stride *stride*. The skewness is defined as,

$$skew = \frac{1}{N} \sum \left( \frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^3$$

where  $x_i$  are the elements of the dataset *data*. The skewness measures the asymmetry of the tails of a distribution.

The function computes the mean and estimated standard deviation of *data* via calls to *gsl\_stats\_mean()* and *gsl\_stats\_sd()*.

double **gsl\_stats\_skew\_m\_sd**(const double data[], size\_t stride, size\_t n, double mean, double sd)

This function computes the skewness of the dataset *data* using the given values of the mean *mean* and standard deviation *sd*,

$$skew = \frac{1}{N} \sum \left( \frac{x_i - mean}{sd} \right)^3$$

These functions are useful if you have already computed the mean and standard deviation of *data* and want to avoid recomputing them.

double **gsl\_stats\_kurtosis**(const double data[], size\_t stride, size\_t n)

This function computes the kurtosis of *data*, a dataset of length *n* with stride *stride*. The kurtosis is defined as,

$$kurtosis = \left( \frac{1}{N} \sum \left( \frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^4 \right) - 3$$

The kurtosis measures how sharply peaked a distribution is, relative to its width. The kurtosis is normalized to zero for a Gaussian distribution.

double **gsl\_stats\_kurtosis\_m\_sd**(const double data[], size\_t stride, size\_t n, double mean, double sd)

This function computes the kurtosis of the dataset *data* using the given values of the mean *mean* and standard deviation *sd*,

$$kurtosis = \frac{1}{N} \left( \sum \left( \frac{x_i - mean}{sd} \right)^4 \right) - 3$$

This function is useful if you have already computed the mean and standard deviation of *data* and want to avoid recomputing them.

## 21.4 Autocorrelation

double **gsl\_stats\_lag1\_autocorrelation**(const double data[], const size\_t stride, const size\_t n)

This function computes the lag-1 autocorrelation of the dataset *data*.

$$a_1 = \frac{\sum_{i=2}^n (x_i - \hat{\mu})(x_{i-1} - \hat{\mu})}{\sum_{i=1}^n (x_i - \hat{\mu})(x_i - \hat{\mu})}$$

double **gsl\_stats\_lag1\_autocorrelation\_m**(const double data[], const size\_t stride, const size\_t n, const double mean)

This function computes the lag-1 autocorrelation of the dataset *data* using the given value of the mean *mean*.

## 21.5 Covariance

double **gsl\_stats\_covariance**(const double data1[], const size\_t stride1, const double data2[], const size\_t stride2, const size\_t n)

This function computes the covariance of the datasets *data1* and *data2* which must both be of the same length *n*.

$$covar = \frac{1}{(n-1)} \sum_{i=1}^n (x_i - \hat{x})(y_i - \hat{y})$$

double **gsl\_stats\_covariance\_m**(const double data1[], const size\_t stride1, const double data2[], const size\_t stride2, const size\_t n, const double mean1, const double mean2)

This function computes the covariance of the datasets *data1* and *data2* using the given values of the means, *mean1* and *mean2*. This is useful if you have already computed the means of *data1* and *data2* and want to avoid recomputing them.

## 21.6 Correlation

double **gsl\_stats\_correlation**(const double data1[], const size\_t stride1, const double data2[], const size\_t stride2, const size\_t n)

This function efficiently computes the Pearson correlation coefficient between the datasets *data1* and *data2* which must both be of the same length *n*.

$$r = \frac{cov(x, y)}{\hat{\sigma}_x \hat{\sigma}_y} = \frac{\frac{1}{n-1} \sum (x_i - \hat{x})(y_i - \hat{y})}{\sqrt{\frac{1}{n-1} \sum (x_i - \hat{x})^2} \sqrt{\frac{1}{n-1} \sum (y_i - \hat{y})^2}}$$

double **gsl\_stats\_spearman**(const double data1[], const size\_t stride1, const double data2[], const size\_t stride2, const size\_t n, double work[])

This function computes the Spearman rank correlation coefficient between the datasets *data1* and *data2* which must both be of the same length *n*. Additional workspace of size  $2 * n$  is required in *work*. The Spearman rank correlation between vectors *x* and *y* is equivalent to the Pearson correlation between the ranked vectors  $x_R$  and  $y_R$ , where ranks are defined to be the average of the positions of an element in the ascending order of the values.

## 21.7 Weighted Samples

The functions described in this section allow the computation of statistics for weighted samples. The functions accept an array of samples,  $x_i$ , with associated weights,  $w_i$ . Each sample  $x_i$  is considered as having been drawn from a Gaussian distribution with variance  $\sigma_i^2$ . The sample weight  $w_i$  is defined as the reciprocal of this variance,  $w_i = 1/\sigma_i^2$ . Setting a weight to zero corresponds to removing a sample from a dataset.

double **gsl\_stats\_wmean**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n)

This function returns the weighted mean of the dataset *data* with stride *stride* and length *n*, using the set of weights *w* with stride *wstride* and length *n*. The weighted mean is defined as,

$$\hat{\mu} = \frac{\sum w_i x_i}{\sum w_i}$$



double **gsl\_stats\_wvariance**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n)

This function returns the estimated variance of the dataset *data* with stride *stride* and length *n*, using the set of weights *w* with stride *wstride* and length *n*. The estimated variance of a weighted dataset is calculated as,

$$\hat{\sigma}^2 = \frac{\sum w_i}{(\sum w_i)^2 - \sum (w_i^2)} \sum w_i (x_i - \hat{\mu})^2$$

Note that this expression reduces to an unweighted variance with the familiar  $1/(N - 1)$  factor when there are *N* equal non-zero weights.

double **gsl\_stats\_wvariance\_m**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n, double wmean)

This function returns the estimated variance of the weighted dataset *data* using the given weighted mean *wmean*.

double **gsl\_stats\_wsd**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n)

The standard deviation is defined as the square root of the variance. This function returns the square root of the corresponding variance function *gsl\_stats\_wvariance()* above.

double **gsl\_stats\_wsd\_m**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n, double wmean)

This function returns the square root of the corresponding variance function *gsl\_stats\_wvariance\_m()* above.

double **gsl\_stats\_wvariance\_with\_fixed\_mean**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n, const double mean)

This function computes an unbiased estimate of the variance of the weighted dataset *data* when the population mean *mean* of the underlying distribution is known *a priori*. In this case the estimator for the variance replaces the sample mean  $\hat{\mu}$  by the known population mean  $\mu$ ,

$$\hat{\sigma}^2 = \frac{\sum w_i (x_i - \mu)^2}{\sum w_i}$$

double **gsl\_stats\_wsd\_with\_fixed\_mean**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n, const double mean)

The standard deviation is defined as the square root of the variance. This function returns the square root of the corresponding variance function above.

double **gsl\_stats\_wtss**(const double w[], const size\_t wstride, const double data[], size\_t stride, size\_t n)

double **gsl\_stats\_wtss\_m**(const double w[], const size\_t wstride, const double data[], size\_t stride, size\_t n, double wmean)

These functions return the weighted total sum of squares (TSS) of *data* about the weighted mean. For *gsl\_stats\_wtss\_m()* the user-supplied value of *wmean* is used, and for *gsl\_stats\_wtss()* it is computed using *gsl\_stats\_wmean()*.

$$\text{TSS} = \sum w_i (x_i - \text{wmean})^2$$

double **gsl\_stats\_wabsdev**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n)

This function computes the weighted absolute deviation from the weighted mean of *data*. The absolute deviation from the mean is defined as,

$$\text{absdev} = \frac{\sum w_i |x_i - \hat{\mu}|}{\sum w_i}$$

double **gsl\_stats\_wabsdev\_m**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n, double wmean)

This function computes the absolute deviation of the weighted dataset *data* about the given weighted mean *wmean*.

double **gsl\_stats\_wskew**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n)

This function computes the weighted skewness of the dataset *data*.

$$skew = \frac{\sum w_i ((x_i - \hat{x}) / \hat{\sigma})^3}{\sum w_i}$$

double **gsl\_stats\_wskew\_m\_sd**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n, double wmean, double wsd)

This function computes the weighted skewness of the dataset *data* using the given values of the weighted mean and weighted standard deviation, *wmean* and *wsd*.

double **gsl\_stats\_wkurtosis**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n)

This function computes the weighted kurtosis of the dataset *data*.

$$kurtosis = \frac{\sum w_i ((x_i - \hat{x}) / \hat{\sigma})^4}{\sum w_i} - 3$$

double **gsl\_stats\_wkurtosis\_m\_sd**(const double w[], size\_t wstride, const double data[], size\_t stride, size\_t n, double wmean, double wsd)

This function computes the weighted kurtosis of the dataset *data* using the given values of the weighted mean and weighted standard deviation, *wmean* and *wsd*.

## 21.8 Maximum and Minimum values

The following functions find the maximum and minimum values of a dataset (or their indices). If the data contains NaN-s then a NaN will be returned, since the maximum or minimum value is undefined. For functions which return an index, the location of the first NaN in the array is returned.

double **gsl\_stats\_max**(const double data[], size\_t stride, size\_t n)

This function returns the maximum value in *data*, a dataset of length *n* with stride *stride*. The maximum value is defined as the value of the element  $x_i$  which satisfies  $x_i \geq x_j$  for all  $j$ .

If you want instead to find the element with the largest absolute magnitude you will need to apply `fabs()` or `abs()` to your data before calling this function.

double **gsl\_stats\_min**(const double data[], size\_t stride, size\_t n)

This function returns the minimum value in *data*, a dataset of length *n* with stride *stride*. The minimum value is defined as the value of the element  $x_i$  which satisfies  $x_i \leq x_j$  for all  $j$ .

If you want instead to find the element with the smallest absolute magnitude you will need to apply `fabs()` or `abs()` to your data before calling this function.

void **gsl\_stats\_minmax**(double \*min, double \*max, const double data[], size\_t stride, size\_t n)

This function finds both the minimum and maximum values *min*, *max* in *data* in a single pass.

size\_t **gsl\_stats\_max\_index**(const double data[], size\_t stride, size\_t n)

This function returns the index of the maximum value in *data*, a dataset of length *n* with stride *stride*. The maximum value is defined as the value of the element  $x_i$  which satisfies  $x_i \geq x_j$  for all  $j$ . When there are several equal maximum elements then the first one is chosen.

size\_t **gsl\_stats\_min\_index**(const double data[], size\_t stride, size\_t n)

This function returns the index of the minimum value in *data*, a dataset of length *n* with stride *stride*. The minimum value is defined as the value of the element  $x_i$  which satisfies  $x_i \leq x_j$  for all  $j$ . When there are several equal minimum elements then the first one is chosen.

void **gsl\_stats\_minmax\_index**(size\_t \*min\_index, size\_t \*max\_index, const double data[], size\_t stride, size\_t n)

This function returns the indexes *min\_index*, *max\_index* of the minimum and maximum values in *data* in a single pass.

## 21.9 Median and Percentiles

The median and percentile functions described in this section operate on sorted data in  $O(1)$  time. There is also a routine for computing the median of an unsorted input array in average  $O(n)$  time using the quickselect algorithm. For convenience we use *quantiles*, measured on a scale of 0 to 1, instead of percentiles (which use a scale of 0 to 100).

double **gsl\_stats\_median\_from\_sorted\_data**(const double sorted\_data[], const size\_t stride, const size\_t n)

This function returns the median value of *sorted\_data*, a dataset of length  $n$  with stride *stride*. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function *gsl\_sort()* should always be used first.

When the dataset has an odd number of elements the median is the value of element  $(n-1)/2$ . When the dataset has an even number of elements the median is the mean of the two nearest middle values, elements  $(n-1)/2$  and  $n/2$ . Since the algorithm for computing the median involves interpolation this function always returns a floating-point number, even for integer data types.

double **gsl\_stats\_median**(double data[], const size\_t stride, const size\_t n)

This function returns the median value of *data*, a dataset of length  $n$  with stride *stride*. The median is found using the quickselect algorithm. The input array does not need to be sorted, but note that the algorithm rearranges the array and so the input is not preserved on output.

double **gsl\_stats\_quantile\_from\_sorted\_data**(const double sorted\_data[], size\_t stride, size\_t n, double f)

This function returns a quantile value of *sorted\_data*, a double-precision array of length  $n$  with stride *stride*. The elements of the array must be in ascending numerical order. The quantile is determined by the *f*, a fraction between 0 and 1. For example, to compute the value of the 75th percentile *f* should have the value 0.75.

There are no checks to see whether the data are sorted, so the function *gsl\_sort()* should always be used first.

The quantile is found by interpolation, using the formula

$$\text{quantile} = (1 - \delta)x_i + \delta x_{i+1}$$

where  $i$  is  $\text{floor}((n-1)f)$  and  $\delta$  is  $(n-1)f - i$ .

Thus the minimum value of the array (*data*[0\**stride*]) is given by *f* equal to zero, the maximum value (*data*[( $n-1$ )\**stride*]) is given by *f* equal to one and the median value is given by *f* equal to 0.5. Since the algorithm for computing quantiles involves interpolation this function always returns a floating-point number, even for integer data types.

### 21.10 Order Statistics

The  $k$ -th *order statistic* of a sample is equal to its  $k$ -th smallest value. The  $k$ -th order statistic of a set of  $n$  values  $x = \{x_i\}, 1 \leq i \leq n$  is denoted  $x_{(k)}$ . The median of the set  $x$  is equal to  $x_{(\frac{n}{2})}$  if  $n$  is odd, or the average of  $x_{(\frac{n}{2})}$  and  $x_{(\frac{n}{2}+1)}$  if  $n$  is even. The  $k$ -th smallest element of a length  $n$  vector can be found in average  $O(n)$  time using the quickselect algorithm.

double **gsl\_stats\_select**(double data[], const size\_t stride, const size\_t n, const size\_t k)

This function finds the  $k$ -th smallest element of the input array *data* of length  $n$  and stride *stride* using the quickselect method. The algorithm rearranges the elements of *data* and so the input array is not preserved on output.

## 21.11 Robust Location Estimates

A *location estimate* refers to a typical or central value which best describes a given dataset. The mean and median are both examples of location estimators. However, the mean has a severe sensitivity to data outliers and can give erroneous values when even a small number of outliers are present. The median on the other hand, has a strong insensitivity to data outliers, but due to its non-smoothness it can behave unexpectedly in certain situations. GSL offers the following alternative location estimators, which are robust to the presence of outliers.

### 21.11.1 Trimmed Mean

The trimmed mean, or *truncated mean*, discards a certain number of smallest and largest samples from the input vector before computing the mean of the remaining samples. The amount of trimming is specified by a factor  $\alpha \in [0, 0.5]$ . Then the number of samples discarded from both ends of the input vector is  $\lfloor \alpha n \rfloor$ , where  $n$  is the length of the input. So to discard 25% of the samples from each end, one would set  $\alpha = 0.25$ .

```
double gsl_stats_trmean_from_sorted_data(const double alpha, const double sorted_data[], const size_t
                                         stride, const size_t n)
```

This function returns the trimmed mean of *sorted\_data*, a dataset of length  $n$  with stride *stride*. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function *gsl\_sort()* should always be used first. The trimming factor  $\alpha$  is given in *alpha*. If  $\alpha \geq 0.5$ , then the median of the input is returned.

### 21.11.2 Gastwirth Estimator

Gastwirth's location estimator is a weighted sum of three order statistics,

$$gastwirth = 0.3 \times Q_{\frac{1}{3}} + 0.4 \times Q_{\frac{1}{2}} + 0.3 \times Q_{\frac{2}{3}}$$

where  $Q_{\frac{1}{3}}$  is the one-third quantile,  $Q_{\frac{1}{2}}$  is the one-half quantile (i.e. median), and  $Q_{\frac{2}{3}}$  is the two-thirds quantile.

```
double gsl_stats_gastwirth_from_sorted_data(const double sorted_data[], const size_t stride, const size_t n)
```

This function returns the Gastwirth location estimator of *sorted\_data*, a dataset of length  $n$  with stride *stride*. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function *gsl\_sort()* should always be used first.

## 21.12 Robust Scale Estimates

A *robust scale estimate*, also known as a robust measure of scale, attempts to quantify the statistical dispersion (variability, scatter, spread) in a set of data which may contain outliers. In such datasets, the usual variance or standard deviation scale estimate can be rendered useless by even a single outlier.

### 21.12.1 Median Absolute Deviation (MAD)

The median absolute deviation (MAD) is defined as

$$MAD = 1.4826 \times \text{median} \{|x_i - \text{median}(x)|\}$$

In words, first the median of all samples is computed. Then the median is subtracted from all samples in the input to find the deviation of each sample from the median. The median of all absolute deviations is then the MAD. The factor 1.4826 makes the MAD an unbiased estimator of the standard deviation for Gaussian data. The median absolute deviation has an asymptotic efficiency of 37%.

double **gsl\_stats\_mad0**(const double data[], const size\_t stride, const size\_t n, double work[])

double **gsl\_stats\_mad**(const double data[], const size\_t stride, const size\_t n, double work[])

These functions return the median absolute deviation of *data*, a dataset of length *n* and stride *stride*. The **mad0** function calculates median  $\{|x_i - \text{median}(x)|\}$  (i.e. the *MAD* statistic without the bias correction scale factor). These functions require additional workspace of size *n* provided in *work*.

### 21.12.2 $S_n$ Statistic

The  $S_n$  statistic developed by Croux and Rousseeuw is defined as

$$S_n = 1.1926 \times c_n \times \text{median}_i \{ \text{median}_j (|x_i - x_j|) \}$$

For each sample  $x_i, 1 \leq i \leq n$ , the median of the values  $|x_i - x_j|$  is computed for all  $x_j, 1 \leq j \leq n$ . This yields *n* values, whose median then gives the final  $S_n$ . The factor 1.1926 makes  $S_n$  an unbiased estimate of the standard deviation for Gaussian data. The factor  $c_n$  is a correction factor to correct bias in small sample sizes.  $S_n$  has an asymptotic efficiency of 58%.

double **gsl\_stats\_Sn0\_from\_sorted\_data**(const double sorted\_data[], const size\_t stride, const size\_t n, double work[])

double **gsl\_stats\_Sn\_from\_sorted\_data**(const double sorted\_data[], const size\_t stride, const size\_t n, double work[])

These functions return the  $S_n$  statistic of *sorted\_data*, a dataset of length *n* with stride *stride*. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function *gsl\_sort()* should always be used first. The **Sn0** function calculates  $\text{median}_i \{ \text{median}_j (|x_i - x_j|) \}$  (i.e. the  $S_n$  statistic without the bias correction scale factors). These functions require additional workspace of size *n* provided in *work*.

### 21.12.3 $Q_n$ Statistic

The  $Q_n$  statistic developed by Croux and Rousseeuw is defined as

$$Q_n = 2.21914 \times d_n \times \{ |x_i - x_j|, i < j \}_{(k)}$$

The factor 2.21914 makes  $Q_n$  an unbiased estimate of the standard deviation for Gaussian data. The factor  $d_n$  is a correction factor to correct bias in small sample sizes. The order statistic is

$$k = \left( \left\lfloor \frac{n}{2} \right\rfloor + 1 \right)_2$$

$Q_n$  has an asymptotic efficiency of 82%.

double **gsl\_stats\_Qn0\_from\_sorted\_data**(const double sorted\_data[], const size\_t stride, const size\_t n, double work[], int work\_int[])

double **gsl\_stats\_Qn\_from\_sorted\_data**(const double sorted\_data[], const size\_t stride, const size\_t n, double work[], int work\_int[])

These functions return the  $Q_n$  statistic of *sorted\_data*, a dataset of length *n* with stride *stride*. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function *gsl\_sort()* should always be used first. The **Qn0** function calculates  $\{ |x_i - x_j|, i < j \}_{(k)}$  (i.e.  $Q_n$  without the bias correction scale factors). These functions require additional workspace of size  $3n$  provided in *work* and integer workspace of size  $5n$  provided in *work\_int*.

## 21.13 Examples

Here is a basic example of how to use the statistical functions:

```
#include <stdio.h>
#include <gsl/gsl_statistics.h>

int
main(void)
{
    double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
    double mean, variance, largest, smallest;

    mean      = gsl_stats_mean(data, 1, 5);
    variance  = gsl_stats_variance(data, 1, 5);
    largest   = gsl_stats_max(data, 1, 5);
    smallest  = gsl_stats_min(data, 1, 5);

    printf ("The dataset is %g, %g, %g, %g, %g\n",
           data[0], data[1], data[2], data[3], data[4]);

    printf ("The sample mean is %g\n", mean);
    printf ("The estimated variance is %g\n", variance);
    printf ("The largest value is %g\n", largest);
    printf ("The smallest value is %g\n", smallest);
    return 0;
}
```

The program should produce the following output,

```
The dataset is 17.2, 18.1, 16.5, 18.3, 12.6
The sample mean is 16.54
The estimated variance is 5.373
The largest value is 18.3
The smallest value is 12.6
```

Here is an example using sorted data,

```
#include <stdio.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_statistics.h>

int
main(void)
{
    double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
    double median, upperq, lowerq;

    printf ("Original dataset: %g, %g, %g, %g, %g\n",
           data[0], data[1], data[2], data[3], data[4]);

    gsl_sort (data, 1, 5);
```

(continues on next page)

(continued from previous page)

```

printf ("Sorted dataset: %g, %g, %g, %g, %g\n",
        data[0], data[1], data[2], data[3], data[4]);

median
    = gsl_stats_median_from_sorted_data (data,
                                         1, 5);

upperq
    = gsl_stats_quantile_from_sorted_data (data,
                                           1, 5,
                                           0.75);

lowerq
    = gsl_stats_quantile_from_sorted_data (data,
                                           1, 5,
                                           0.25);

printf ("The median is %g\n", median);
printf ("The upper quartile is %g\n", upperq);
printf ("The lower quartile is %g\n", lowerq);
return 0;
}

```

This program should produce the following output,

```

Original dataset: 17.2, 18.1, 16.5, 18.3, 12.6
Sorted dataset: 12.6, 16.5, 17.2, 18.1, 18.3
The median is 17.2
The upper quartile is 18.1
The lower quartile is 16.5

```

## 21.14 References and Further Reading

The standard reference for almost any topic in statistics is the multi-volume *Advanced Theory of Statistics* by Kendall and Stuart.

- Maurice Kendall, Alan Stuart, and J. Keith Ord. *The Advanced Theory of Statistics* (multiple volumes) reprinted as *Kendall's Advanced Theory of Statistics*. Wiley, ISBN 047023380X.

Many statistical concepts can be more easily understood by a Bayesian approach. The following book by Gelman, Carlin, Stern and Rubin gives a comprehensive coverage of the subject.

- Andrew Gelman, John B. Carlin, Hal S. Stern, Donald B. Rubin. *Bayesian Data Analysis*. Chapman & Hall, ISBN 0412039915.

For physicists the Particle Data Group provides useful reviews of Probability and Statistics in the “Mathematical Tools” section of its Annual Review of Particle Physics.

- *Review of Particle Properties*, R.M. Barnett et al., Physical Review D54, 1 (1996)

The Review of Particle Physics is available online at the website <http://pdg.lbl.gov/>.

The following papers describe robust scale estimation,

- C. Croux and P. J. Rousseeuw, *Time-Efficient algorithms for two highly robust estimators of scale*, Comp. Stat., Physica, Heidelberg, 1992.

- P. J. Rousseeuw and C. Croux, *Explicit scale estimators with high breakdown point*, L1-Statistical Analysis and Related Methods, pp. 77-92, 1992.



## RUNNING STATISTICS

This chapter describes routines for computing running statistics, also known as online statistics, of data. These routines are suitable for handling large datasets for which it may be inconvenient or impractical to store in memory all at once. The data can be processed in a single pass, one point at a time. Each time a data point is added to the accumulator, internal parameters are updated in order to compute the current mean, variance, standard deviation, skewness, and kurtosis. These statistics are exact, and are updated with numerically stable single-pass algorithms. The median and arbitrary quantiles are also available, however these calculations use algorithms which provide approximations, and grow more accurate as more data is added to the accumulator.

The functions described in this chapter are declared in the header file `gsl_rstat.h`.

### 22.1 Initializing the Accumulator

type **gsl\_rstat\_workspace**

This workspace contains parameters used to calculate various statistics and are updated after each data point is added to the accumulator.

*gsl\_rstat\_workspace* \***gsl\_rstat\_alloc**(void)

This function allocates a workspace for computing running statistics. The size of the workspace is  $O(1)$ .

void **gsl\_rstat\_free**(*gsl\_rstat\_workspace* \*w)

This function frees the memory associated with the workspace *w*.

int **gsl\_rstat\_reset**(*gsl\_rstat\_workspace* \*w)

This function resets the workspace *w* to its initial state, so it can begin working on a new set of data.

### 22.2 Adding Data to the Accumulator

int **gsl\_rstat\_add**(const double x, *gsl\_rstat\_workspace* \*w)

This function adds the data point *x* to the statistical accumulator, updating calculations of the mean, variance, standard deviation, skewness, kurtosis, and median.

size\_t **gsl\_rstat\_n**(const *gsl\_rstat\_workspace* \*w)

This function returns the number of data so far added to the accumulator.

## 22.3 Current Statistics

double **gsl\_rstat\_min**(const *gsl\_rstat\_workspace* \*w)

This function returns the minimum value added to the accumulator.

double **gsl\_rstat\_max**(const *gsl\_rstat\_workspace* \*w)

This function returns the maximum value added to the accumulator.

double **gsl\_rstat\_mean**(const *gsl\_rstat\_workspace* \*w)

This function returns the mean of all data added to the accumulator, defined as

$$\hat{\mu} = \frac{1}{N} \sum x_i$$

double **gsl\_rstat\_variance**(const *gsl\_rstat\_workspace* \*w)

This function returns the variance of all data added to the accumulator, defined as

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - \hat{\mu})^2$$

double **gsl\_rstat\_sd**(const *gsl\_rstat\_workspace* \*w)

This function returns the standard deviation of all data added to the accumulator, defined as the square root of the variance given above.

double **gsl\_rstat\_sd\_mean**(const *gsl\_rstat\_workspace* \*w)

This function returns the standard deviation of the mean, defined as

$$\hat{\sigma}_{\hat{\mu}} = \frac{\hat{\sigma}}{\sqrt{N}}$$

double **gsl\_rstat\_rms**(const *gsl\_rstat\_workspace* \*w)

This function returns the root mean square of all data added to the accumulator, defined as

$$rms = \sqrt{\frac{1}{N} \sum x_i^2}$$

double **gsl\_rstat\_skew**(const *gsl\_rstat\_workspace* \*w)

This function returns the skewness of all data added to the accumulator, defined as

$$skew = \frac{1}{N} \sum \left( \frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^3$$

double **gsl\_rstat\_kurtosis**(const *gsl\_rstat\_workspace* \*w)

This function returns the kurtosis of all data added to the accumulator, defined as

$$kurtosis = \left( \frac{1}{N} \sum \left( \frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^4 \right) - 3$$

double **gsl\_rstat\_median**(*gsl\_rstat\_workspace* \*w)

This function returns an estimate of the median of the data added to the accumulator.

## 22.4 Quantiles

The functions in this section estimate quantiles dynamically without storing the entire dataset, using the algorithm of Jain and Chlamtec, 1985. Only five points (markers) are stored which represent the minimum and maximum of the data, as well as current estimates of the  $p/2$ -,  $p$ -, and  $(1 + p)/2$ -quantiles. Each time a new data point is added, the marker positions and heights are updated.

type **gsl\_rstat\_quantile\_workspace**

This workspace contains parameters for estimating quantiles of the current dataset

*gsl\_rstat\_quantile\_workspace* \***gsl\_rstat\_quantile\_alloc**(const double p)

This function allocates a workspace for the dynamic estimation of  $p$ -quantiles, where  $p$  is between 0 and 1. The median corresponds to  $p = 0.5$ . The size of the workspace is  $O(1)$ .

void **gsl\_rstat\_quantile\_free**(*gsl\_rstat\_quantile\_workspace* \*w)

This function frees the memory associated with the workspace  $w$ .

int **gsl\_rstat\_quantile\_reset**(*gsl\_rstat\_quantile\_workspace* \*w)

This function resets the workspace  $w$  to its initial state, so it can begin working on a new set of data.

int **gsl\_rstat\_quantile\_add**(const double x, *gsl\_rstat\_quantile\_workspace* \*w)

This function updates the estimate of the  $p$ -quantile with the new data point  $x$ .

double **gsl\_rstat\_quantile\_get**(*gsl\_rstat\_quantile\_workspace* \*w)

This function returns the current estimate of the  $p$ -quantile.

## 22.5 Examples

Here is a basic example of how to use the statistical functions:

```
#include <stdio.h>
#include <gsl/gsl_rstat.h>

int
main(void)
{
    double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
    double mean, variance, largest, smallest, sd,
           rms, sd_mean, median, skew, kurtosis;
    gsl_rstat_workspace *rstat_p = gsl_rstat_alloc();
    size_t i, n;

    /* add data to rstat accumulator */
    for (i = 0; i < 5; ++i)
        gsl_rstat_add(data[i], rstat_p);

    mean      = gsl_rstat_mean(rstat_p);
    variance  = gsl_rstat_variance(rstat_p);
    largest   = gsl_rstat_max(rstat_p);
    smallest  = gsl_rstat_min(rstat_p);
    median    = gsl_rstat_median(rstat_p);
    sd        = gsl_rstat_sd(rstat_p);
    sd_mean   = gsl_rstat_sd_mean(rstat_p);
    skew      = gsl_rstat_skew(rstat_p);
```

(continues on next page)

(continued from previous page)

```

rms      = gsl_rstat_rms(rstat_p);
kurtosis = gsl_rstat_kurtosis(rstat_p);
n        = gsl_rstat_n(rstat_p);

printf ("The dataset is %g, %g, %g, %g, %g\n",
        data[0], data[1], data[2], data[3], data[4]);

printf ("The sample mean is %g\n", mean);
printf ("The estimated variance is %g\n", variance);
printf ("The largest value is %g\n", largest);
printf ("The smallest value is %g\n", smallest);
printf("The median is %g\n", median);
printf("The standard deviation is %g\n", sd);
printf("The root mean square is %g\n", rms);
printf("The standard deviation of the mean is %g\n", sd_mean);
printf("The skew is %g\n", skew);
printf("The kurtosis %g\n", kurtosis);
printf("There are %zu items in the accumulator\n", n);

gsl_rstat_reset(rstat_p);
n = gsl_rstat_n(rstat_p);
printf("There are %zu items in the accumulator\n", n);

gsl_rstat_free(rstat_p);

return 0;
}

```

The program should produce the following output,

```

The dataset is 17.2, 18.1, 16.5, 18.3, 12.6
The sample mean is 16.54
The estimated variance is 5.373
The largest value is 18.3
The smallest value is 12.6
The median is 17.2
The standard deviation is 2.31797
The root mean square is 16.6694
The standard deviation of the mean is 1.03663
The skew is -0.829058
The kurtosis -1.2217
There are 5 items in the accumulator
There are 0 items in the accumulator

```

This next program estimates the lower quartile, median and upper quartile from 10,000 samples of a random Rayleigh distribution, using the  $P^2$  algorithm of Jain and Chlamtec. For comparison, the exact values are also computed from the sorted dataset.

```

#include <stdio.h>
#include <stdlib.h>
#include <gsl/gsl_rstat.h>
#include <gsl/gsl_statistics.h>

```

(continues on next page)

(continued from previous page)

```

#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sort.h>

int
main(void)
{
    const size_t N = 10000;
    double *data = malloc(N * sizeof(double));
    gsl_rstat_quantile_workspace *work_25 = gsl_rstat_quantile_alloc(0.25);
    gsl_rstat_quantile_workspace *work_50 = gsl_rstat_quantile_alloc(0.5);
    gsl_rstat_quantile_workspace *work_75 = gsl_rstat_quantile_alloc(0.75);
    gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
    double exact_p25, exact_p50, exact_p75;
    double val_p25, val_p50, val_p75;
    size_t i;

    /* add data to quantile accumulators; also store data for exact
     * comparisons */
    for (i = 0; i < N; ++i)
    {
        data[i] = gsl_ran_rayleigh(r, 1.0);
        gsl_rstat_quantile_add(data[i], work_25);
        gsl_rstat_quantile_add(data[i], work_50);
        gsl_rstat_quantile_add(data[i], work_75);
    }

    /* exact values */
    gsl_sort(data, 1, N);
    exact_p25 = gsl_stats_quantile_from_sorted_data(data, 1, N, 0.25);
    exact_p50 = gsl_stats_quantile_from_sorted_data(data, 1, N, 0.5);
    exact_p75 = gsl_stats_quantile_from_sorted_data(data, 1, N, 0.75);

    /* estimated values */
    val_p25 = gsl_rstat_quantile_get(work_25);
    val_p50 = gsl_rstat_quantile_get(work_50);
    val_p75 = gsl_rstat_quantile_get(work_75);

    printf ("The dataset is %g, %g, %g, %g, %g, ...\\n",
           data[0], data[1], data[2], data[3], data[4]);

    printf ("0.25 quartile: exact = %.5f, estimated = %.5f, error = %.6e\\n",
           exact_p25, val_p25, (val_p25 - exact_p25) / exact_p25);
    printf ("0.50 quartile: exact = %.5f, estimated = %.5f, error = %.6e\\n",
           exact_p50, val_p50, (val_p50 - exact_p50) / exact_p50);
    printf ("0.75 quartile: exact = %.5f, estimated = %.5f, error = %.6e\\n",
           exact_p75, val_p75, (val_p75 - exact_p75) / exact_p75);

    gsl_rstat_quantile_free(work_25);
    gsl_rstat_quantile_free(work_50);
    gsl_rstat_quantile_free(work_75);
    gsl_rng_free(r);

```

(continues on next page)

(continued from previous page)

```
free(data);  
  
return 0;  
}
```

The program should produce the following output,

```
The dataset is 0.00645272, 0.0074002, 0.0120706, 0.0207256, 0.0227282, ...  
0.25 quartile: exact = 0.75766, estimated = 0.75580, error = -2.450209e-03  
0.50 quartile: exact = 1.17508, estimated = 1.17438, error = -5.995912e-04  
0.75 quartile: exact = 1.65347, estimated = 1.65696, error = 2.110571e-03
```

## 22.6 References and Further Reading

The algorithm used to dynamically estimate  $p$ -quantiles is described in the paper,

- R. Jain and I. Chlamtac. *The  $P^2$  algorithm for dynamic calculation of quantiles and histograms without storing observations*, Communications of the ACM, Volume 28 (October), Number 10, 1985, p. 1076-1085.

## MOVING WINDOW STATISTICS

This chapter describes routines for computing *moving window statistics* (also called rolling statistics and running statistics), using a window around a sample which is used to calculate various local statistical properties of an input data stream. The window is then slid forward by one sample to process the next data point and so on.

The functions described in this chapter are declared in the header file `gsl_movstat.h`.

### 23.1 Introduction

This chapter is concerned with calculating various statistics from subsets of a given dataset. The main idea is to compute statistics in the vicinity of a given data sample by defining a *window* which includes the sample itself as well as some specified number of samples before and after the sample in question. For a sample  $x_i$ , we define a window  $W_i^{H,J}$  as

$$W_i^{H,J} = \{x_{i-H}, \dots, x_i, \dots, x_{i+J}\}$$

The parameters  $H$  and  $J$  are non-negative integers specifying the number of samples to include before and after the sample  $x_i$ . Statistics such as the mean and standard deviation of the window  $W_i^{H,J}$  may be computed, and then the window is shifted forward by one sample to focus on  $x_{i+1}$ . The total number of samples in the window is  $K = H + J + 1$ . To define a symmetric window centered on  $x_i$ , one would set  $H = J = \lfloor K/2 \rfloor$ .

### 23.2 Handling Endpoints

When processing samples near the ends of the input signal, there will not be enough samples to fill the window  $W_i^{H,J}$  defined above. Therefore the user must specify how to construct the windows near the end points. This is done by passing an input argument of type `gsl_movstat_end_t`:

type `gsl_movstat_end_t`

This data type specifies how to construct windows near end points and can be selected from the following choices:

#### **GSL\_MOVSTAT\_END\_PADZERO**

With this option, a full window of length  $K$  will be constructed by inserting zeros into the window near the signal end points. Effectively, the input signal is modified to

$$\tilde{x} = \{\underbrace{0, \dots, 0}_{H \text{ zeros}}, x_1, x_2, \dots, x_{n-1}, x_n, \underbrace{0, \dots, 0}_{J \text{ zeros}}\}$$

to ensure a well-defined window for all  $x_i$ .

**GSL\_MOVSTAT\_END\_PADVALUE**

With this option, a full window of length  $K$  will be constructed by padding the window with the first and last sample in the input signal. Effectively, the input signal is modified to

$$\tilde{x} = \underbrace{\{x_1, \dots, x_1\}}_H, x_1, x_2, \dots, x_{n-1}, x_n, \underbrace{\{x_n, \dots, x_n\}}_J$$

**GSL\_MOVSTAT\_END\_TRUNCATE**

With this option, no padding is performed, and the windows are simply truncated as the end points are approached.

## 23.3 Allocation for Moving Window Statistics

type **gsl\_movstat\_workspace**

The moving window statistical routines use a common workspace.

*gsl\_movstat\_workspace* \***gsl\_movstat\_alloc**(const size\_t K)

This function allocates a workspace for computing symmetric, centered moving statistics with a window length of  $K$  samples. In this case,  $H = J = \lfloor K/2 \rfloor$ . The size of the workspace is  $O(7K)$ .

*gsl\_movstat\_workspace* \***gsl\_movstat\_alloc2**(const size\_t H, const size\_t J)

This function allocates a workspace for computing moving statistics using a window with  $H$  samples prior to the current sample, and  $J$  samples after the current sample. The total window size is  $K = H + J + 1$ . The size of the workspace is  $O(7K)$ .

void \***gsl\_movstat\_free**(*gsl\_movstat\_workspace* \*w)

This function frees the memory associated with  $w$ .

## 23.4 Moving Mean

The moving window mean calculates the mean of the values of each window  $W_i^{H,J}$ .

$$\hat{\mu}_i = \frac{1}{|W_i^{H,J}|} \sum_{x_m \in W_i^{H,J}} x_m$$

Here,  $|W_i^{H,J}|$  represents the number of elements in the window  $W_i^{H,J}$ . This will normally be  $K$ , unless the `GSL_MOVSTAT_END_TRUNCATE` option is selected, in which case it could be less than  $K$  near the signal end points.

int **gsl\_movstat\_mean**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_vector* \*x, *gsl\_vector* \*y, *gsl\_movstat\_workspace* \*w)

This function computes the moving window mean of the input vector  $x$ , storing the output in  $y$ . The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed to have  $x = y$  for an in-place moving mean.



## 23.5 Moving Variance and Standard Deviation

The moving window variance calculates the *sample variance* of the values of each window  $W_i^{H,J}$ , defined by

$$\hat{\sigma}_i^2 = \frac{1}{(|W_i^{H,J}| - 1)} \sum_{x_m \in W_i^{H,J}} (x_m - \hat{\mu}_i)^2$$

where  $\hat{\mu}_i$  is the mean of  $W_i^{H,J}$  defined above. The standard deviation  $\hat{\sigma}_i$  is the square root of the variance.

int **gsl\_movstat\_variance**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_vector* \*x, *gsl\_vector* \*y, *gsl\_movstat\_workspace* \*w)

This function computes the moving window variance of the input vector **x**, storing the output in **y**. The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed to have **x = y** for an in-place moving variance.

int **gsl\_movstat\_sd**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_vector* \*x, *gsl\_vector* \*y, *gsl\_movstat\_workspace* \*w)

This function computes the moving window standard deviation of the input vector **x**, storing the output in **y**. The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed to have **x = y** for an in-place moving standard deviation.

## 23.6 Moving Minimum and Maximum

The moving minimum/maximum calculates the minimum and maximum values of each window  $W_i^{H,J}$ .

$$y_i^{min} = \min(W_i^{H,J})$$

$$y_i^{max} = \max(W_i^{H,J})$$

int **gsl\_movstat\_min**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_vector* \*x, *gsl\_vector* \*y, *gsl\_movstat\_workspace* \*w)

This function computes the moving minimum of the input vector **x**, storing the result in **y**. The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed to have **x = y** for an in-place moving minimum.

int **gsl\_movstat\_max**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_vector* \*x, *gsl\_vector* \*y, *gsl\_movstat\_workspace* \*w)

This function computes the moving maximum of the input vector **x**, storing the result in **y**. The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed to have **x = y** for an in-place moving maximum.

int **gsl\_movstat\_minmax**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_vector* \*x, *gsl\_vector* \*y\_min, *gsl\_vector* \*y\_max, *gsl\_movstat\_workspace* \*w)

This function computes the moving minimum and maximum of the input vector **x**, storing the window minimums in **y\_min** and the window maximums in **y\_max**. The parameter *endtype* specifies how windows near the ends of the input should be handled.

## 23.7 Moving Sum

The moving window sum calculates the sum of the values of each window  $W_i^{H,J}$ .

$$y_i = \sum_{x_m \in W_i^{H,J}} x_m$$

```
int gsl_movstat_sum(const gsl_movstat_end_t endtype, const gsl_vector *x, gsl_vector *y,  
                   gsl_movstat_workspace *w)
```

This function computes the moving window sum of the input vector  $x$ , storing the output in  $y$ . The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed to have  $x = y$  for an in-place moving sum.

## 23.8 Moving Median

The moving median calculates the median of the window  $W_i^{H,J}$  for each sample  $x_i$ :

$$y_i = \text{median} \left( W_i^{H,J} \right)$$

```
int gsl_movstat_median(const gsl_movstat_end_t endtype, const gsl_vector *x, gsl_vector *y,  
                       gsl_movstat_workspace *w)
```

This function computes the moving median of the input vector  $x$ , storing the output in  $y$ . The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed for  $x = y$  for an in-place moving window median.

## 23.9 Robust Scale Estimation

A common problem in statistics is to quantify the dispersion (also known as the variability, scatter, and spread) of a set of data. Often this is done by calculating the variance or standard deviation. However these statistics are strongly influenced by outliers, and can often provide erroneous results when even a small number of outliers are present.

Several useful statistics have emerged to provide robust estimates of scale which are not as susceptible to data outliers. A few of these statistical scale estimators are described below.

### 23.9.1 Moving MAD

The median absolute deviation (MAD) for the window  $W_i^{H,J}$  is defined to be the median of the absolute deviations from the window's median:

$$MAD_i = 1.4826 \times \text{median} \left( \left| W_i^{H,J} - \text{median} \left( W_i^{H,J} \right) \right| \right)$$

The factor of 1.4826 makes the MAD an unbiased estimator of the standard deviation for Gaussian data. The MAD has an efficiency of 37%. See [here](#) for more information.

```
int gsl_movstat_mad0(const gsl_movstat_end_t endtype, const gsl_vector *x, gsl_vector *xmedian, gsl_vector  
                    *xmad, gsl_movstat_workspace *w)
```

```
int gsl_movstat_mad(const gsl_movstat_end_t endtype, const gsl_vector *x, gsl_vector *xmedian, gsl_vector
*xmad, gsl_movstat_workspace *w)
```

These functions compute the moving MAD of the input vector  $x$  and store the result in  $xmad$ . The medians of each window  $W_i^{H,J}$  are stored in  $xmedian$  on output. The inputs  $x$ ,  $xmedian$ , and  $xmad$  must all be the same length. The parameter *endtype* specifies how windows near the ends of the input should be handled. The function `mad0` does not include the scale factor of 1.4826, while the function `mad` does include this factor.

### 23.9.2 Moving QQR

The  $q$ -quantile range (QQR) is the difference between the  $(1 - q)$  and  $q$  quantiles of a set of data,

$$QQR = Q_{1-q} - Q_q$$

The case  $q = 0.25$  corresponds to the well-known *interquartile range (IQR)*, which is the difference between the 75th and 25th percentiles of a set of data. The QQR is a *trimmed estimator*, the main idea being to discard the largest and smallest values in a data window and compute a scale estimate from the remaining middle values. In the case of the IQR, the largest and smallest 25% of the data are discarded and the scale is estimated from the remaining (middle) 50%.

```
int gsl_movstat_qqr(const gsl_movstat_end_t endtype, const gsl_vector *x, const double q, gsl_vector *xqqr,
gsl_movstat_workspace *w)
```

This function computes the moving QQR of the input vector  $x$  and stores the  $q$ -quantile ranges of each window  $W_i^{H,J}$  in  $xqqr$ . The quantile parameter  $q$  must be between 0 and 0.5. The input  $q = 0.25$  corresponds to the IQR. The inputs  $x$  and  $xqqr$  must be the same length. The parameter *endtype* specifies how windows near the ends of the input should be handled.

### 23.9.3 Moving $S_n$

The  $S_n$  statistic proposed by Croux and Rousseeuw is based on pairwise differences between all samples in the window. It has an efficiency of 58%, significantly higher than the MAD. See [here](#) for more information.

```
int gsl_movstat_Sn(const gsl_movstat_end_t endtype, const gsl_vector *x, gsl_vector *xscale,
gsl_movstat_workspace *w)
```

This function computes the moving  $S_n$  of the input vector  $x$  and stores the output in  $xscale$ . The inputs  $x$  and  $xscale$  must be the same length. The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed for  $x = xscale$  for an in-place moving window  $S_n$ .

### 23.9.4 Moving $Q_n$

The  $Q_n$  statistic proposed by Croux and Rousseeuw is loosely based on the Hodges-Lehmann location estimator. It has a relatively high efficiency of 82%. See [here](#) for more information.

```
int gsl_movstat_Qn(const gsl_movstat_end_t endtype, const gsl_vector *x, gsl_vector *xscale,
gsl_movstat_workspace *w)
```

This function computes the moving  $Q_n$  of the input vector  $x$  and stores the output in  $xscale$ . The inputs  $x$  and  $xscale$  must be the same length. The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed for  $x = xscale$  for an in-place moving window  $Q_n$ .

## 23.10 User-defined Moving Statistics

GSL offers an interface for users to define their own moving window statistics functions, without needing to implement the edge-handling and accumulator machinery. This can be done by explicitly constructing the windows  $W_i^{H,J}$  for a given input signal (*gsl\_movstat\_fill()*), or by calculating a user-defined function for each window automatically. In order to apply a user-defined function to each window, users must define a variable of type *gsl\_movstat\_function* to pass into *gsl\_movstat\_apply()*. This structure is defined as follows.

type **gsl\_movstat\_function**

Structure specifying user-defined moving window statistical function:

```
typedef struct
{
    double (* function) (const size_t n, double x[], void * params);
    void * params;
} gsl_movstat_function;
```

This structure contains a pointer to the user-defined function as well as possible parameters to pass to the function.

double (\***function**)(const size\_t n, double x[], void \*params)

This function returns the user-defined statistic of the array *x* of length *n*. User-specified parameters are passed in via *params*. It is allowed to modify the array *x*.

void \***params**

User-specified parameters to be passed into the function.

int **gsl\_movstat\_apply**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_movstat\_function* \*F, const *gsl\_vector* \*x, *gsl\_vector* \*y, *gsl\_movstat\_workspace* \*w)

This function applies the user-defined moving window statistic specified in *F* to the input vector *x*, storing the output in *y*. The parameter *endtype* specifies how windows near the ends of the input should be handled. It is allowed for *x = y* for an in-place moving window calculation.

size\_t **gsl\_movstat\_fill**(const *gsl\_movstat\_end\_t* endtype, const *gsl\_vector* \*x, const size\_t idx, const size\_t H, const size\_t J, double \*window)

This function explicitly constructs the sliding window for the input vector *x* which is centered on the sample *idx*. On output, the array *window* will contain  $W_{idx}^{H,J}$ . The number of samples to the left and right of the sample *idx* are specified by *H* and *J* respectively. The parameter *endtype* specifies how windows near the ends of the input should be handled. The function returns the size of the window.

## 23.11 Accumulators

Many of the algorithms of this chapter are based on an accumulator design, which process the input vector one sample at a time, updating calculations of the desired statistic for the current window. Each accumulator is stored in the following structure:

type **gsl\_movstat\_accum**

Structure specifying accumulator for moving window statistics:

```
typedef struct
{
    size_t (* size) (const size_t n);
    int (* init) (const size_t n, void * vstate);
    int (* insert) (const double x, void * vstate);
```

(continues on next page)

(continued from previous page)

```

int (* delete) (void * vstate);
int (* get) (void * params, double * result, const void * vstate);
} gsl_movstat_accum;

```

The structure contains function pointers responsible for performing different tasks for the accumulator.

`size_t (*size)(const size_t n)`

This function returns the size of the workspace (in bytes) needed by the accumulator for a moving window of length  $n$ .

`int (*init)(const size_t n, void *vstate)`

This function initializes the workspace `vstate` for a moving window of length  $n$ .

`int (*insert)(const double x, void *vstate)`

This function inserts a single sample  $x$  into the accumulator, updating internal calculations of the desired statistic. If the accumulator is full (i.e.  $n$  samples have already been inserted), then the oldest sample is deleted from the accumulator.

`int (*delete)(void *vstate)`

This function deletes the oldest sample from the accumulator, updating internal calculations of the desired statistic.

`int (*get)(void *params, double *result, const void *vstate)`

This function stores the desired statistic for the current window in `result`. The input `params` specifies optional parameters for calculating the statistic.

The following accumulators of type `gsl_movstat_accum` are defined by GSL to perform moving window statistics calculations.

`gsl_movstat_accum *gsl_movstat_accum_min`

`gsl_movstat_accum *gsl_movstat_accum_max`

`gsl_movstat_accum *gsl_movstat_accum_minmax`

These accumulators calculate moving window minimum/maximums efficiently, using the algorithm of D. Lemire.

`gsl_movstat_accum *gsl_movstat_accum_mean`

`gsl_movstat_accum *gsl_movstat_accum_sd`

`gsl_movstat_accum *gsl_movstat_accum_variance`

These accumulators calculate the moving window mean, standard deviation, and variance, using the algorithm of B. P. Welford.

`gsl_movstat_accum *gsl_movstat_accum_median`

This accumulator calculates the moving window median using the min/max heap algorithm of Härdle and Steiger.

`gsl_movstat_accum *gsl_movstat_accum_Sn`

`gsl_movstat_accum *gsl_movstat_accum_Qn`

These accumulators calculate the moving window  $S_n$  and  $Q_n$  statistics developed by Croux and Rousseeuw.

`gsl_movstat_accum *gsl_movstat_accum_sum`

This accumulator calculates the moving window sum.

`gsl_movstat_accum *gsl_movstat_accum_qqr`

This accumulator calculates the moving window q-quantile range.

## 23.12 Examples

### 23.12.1 Example 1

The following example program computes the moving mean, minimum and maximum of a noisy sinusoid signal of length  $N = 500$  with a symmetric moving window of size  $K = 11$ .

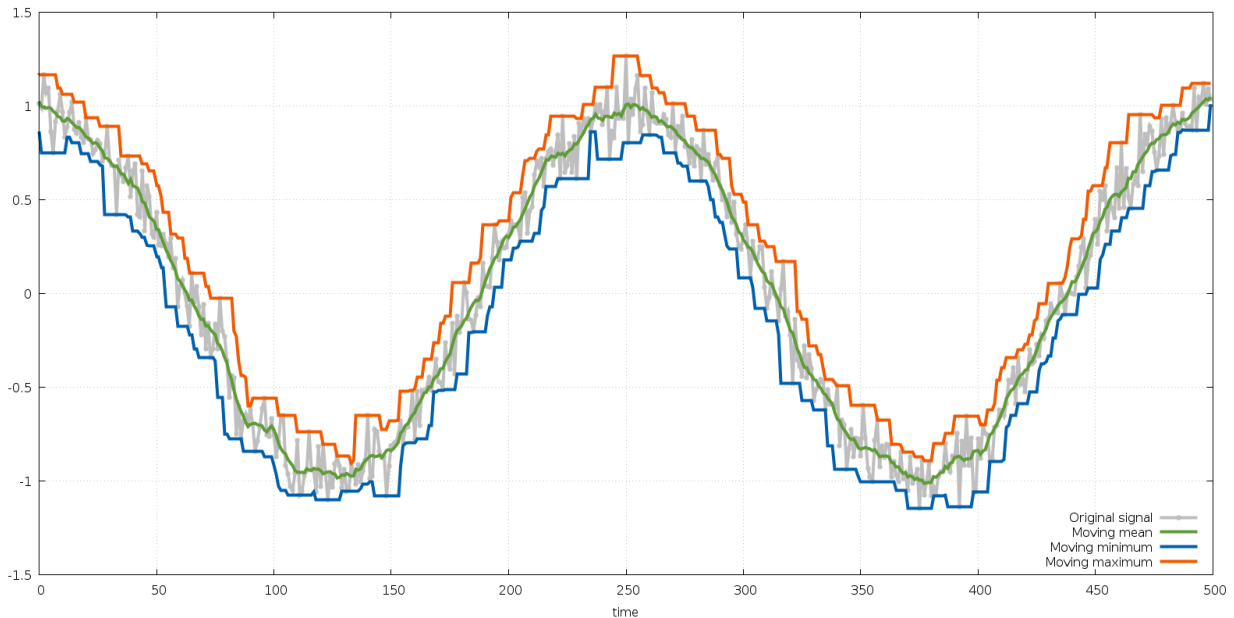


Fig. 23.1: Original signal time series (gray) with moving mean (green), moving minimum (blue), and moving maximum (orange).

The program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_movstat.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
    const size_t N = 500;          /* length of time series */
    const size_t K = 11;         /* window size */
    gsl_movstat_workspace * w = gsl_movstat_alloc(K);
    gsl_vector *x = gsl_vector_alloc(N);
    gsl_vector *xmean = gsl_vector_alloc(N);
    gsl_vector *xmin = gsl_vector_alloc(N);
    gsl_vector *xmax = gsl_vector_alloc(N);
    gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
```

(continues on next page)

(continued from previous page)

```

size_t i;

for (i = 0; i < N; ++i)
{
    double xi = cos(4.0 * M_PI * i / (double) N);
    double ei = gsl_ran_gaussian(r, 0.1);

    gsl_vector_set(x, i, xi + ei);
}

/* compute moving statistics */
gsl_movstat_mean(GSL_MOVSTAT_END_PADVALUE, x, xmean, w);
gsl_movstat_minmax(GSL_MOVSTAT_END_PADVALUE, x, xmin, xmax, w);

/* print results */
for (i = 0; i < N; ++i)
{
    printf("%zu %f %f %f %f\n",
           i,
           gsl_vector_get(x, i),
           gsl_vector_get(xmean, i),
           gsl_vector_get(xmin, i),
           gsl_vector_get(xmax, i));
}

gsl_vector_free(x);
gsl_vector_free(xmean);
gsl_rng_free(r);
gsl_movstat_free(w);

return 0;
}

```

### 23.12.2 Example 2: Robust Scale

The following example program analyzes a time series of length  $N = 1000$  composed of Gaussian random variates with zero mean whose standard deviation changes in a piecewise constant fashion as shown in the table below.

Sample Range	$\sigma$
1-200	1.0
201-450	5.0
451-600	1.0
601-850	3.0
851-1000	5.0

Additionally, about 1% of the samples are perturbed to represent outliers by adding  $\pm 15$  to the random Gaussian variate. The program calculates the moving statistics MAD, IQR,  $S_n$ ,  $Q_n$ , and the standard deviation using a symmetric moving window of length  $K = 41$ . The results are shown in [Fig. 23.2](#).

The robust statistics follow the true standard deviation piecewise changes well, without being influenced by the outliers. The moving standard deviation (gray curve) is heavily influenced by the presence of the outliers. The program is given

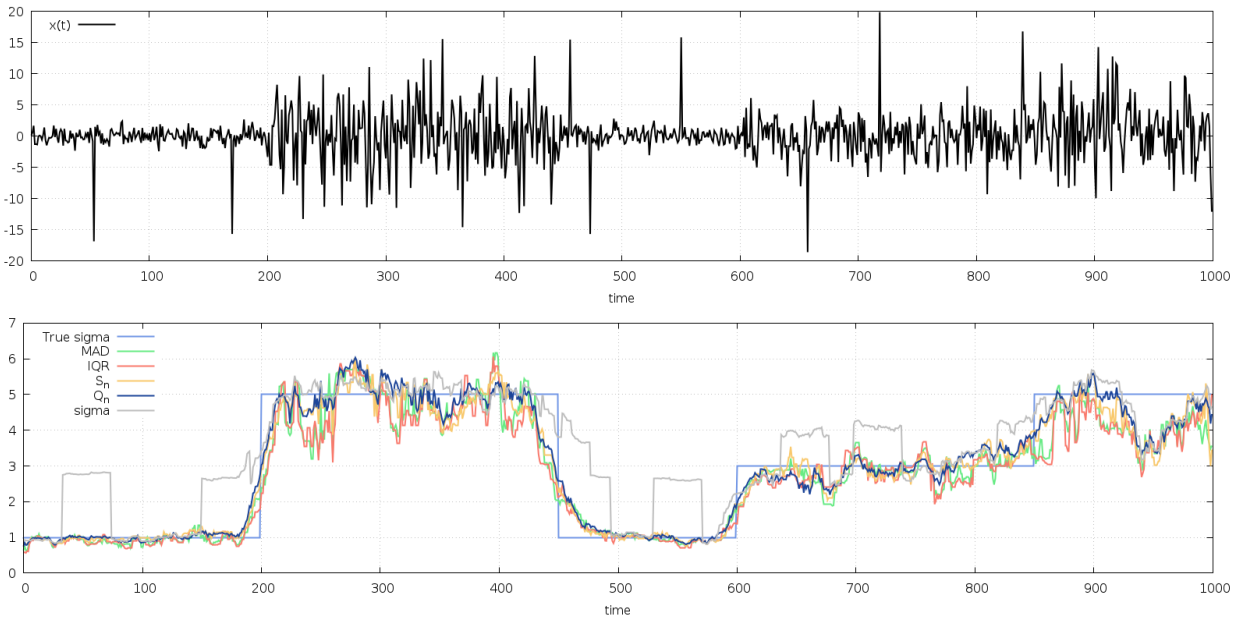


Fig. 23.2: Top: time series of piecewise constant variance. Bottom: scale estimates using a moving window; the true sigma value is in light blue, MAD in green, IQR in red,  $S_n$  in yellow, and  $Q_n$  in dark blue. The moving standard deviation is shown in gray.

below.

```

#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_movstat.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
    const size_t N = 1000;                /* length of time series */
    const double sigma[] = { 1.0, 5.0, 1.0, 3.0, 5.0 }; /* variances */
    const size_t N_sigma[] = { 200, 450, 600, 850, 1000 }; /* samples where variance_
↪changes */
    const size_t K = 41;                  /* window size */
    gsl_vector *x = gsl_vector_alloc(N);
    gsl_vector *xmedian = gsl_vector_alloc(N);
    gsl_vector *xmad = gsl_vector_alloc(N);
    gsl_vector *xiqr = gsl_vector_alloc(N);
    gsl_vector *xSn = gsl_vector_alloc(N);
    gsl_vector *xQn = gsl_vector_alloc(N);
    gsl_vector *xsd = gsl_vector_alloc(N);
    gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
    gsl_movstat_workspace * w = gsl_movstat_alloc(K);

```

(continues on next page)



(continued from previous page)

```

size_t idx = 0;
size_t i;

for (i = 0; i < N; ++i)
{
    double gi = gsl_ran_gaussian(r, sigma[idx]);
    double u = gsl_rng_uniform(r);
    double outlier = (u < 0.01) ? 15.0 * GSL_SIGN(gi) : 0.0;
    double xi = gi + outlier;

    gsl_vector_set(x, i, xi);

    if (i == N_sigma[idx] - 1)
        ++idx;
}

/* compute moving statistics */
gsl_movstat_mad(GSL_MOVSTAT_END_TRUNCATE, x, xmedian, xmad, w);
gsl_movstat_qqr(GSL_MOVSTAT_END_TRUNCATE, x, 0.25, xiqr, w);
gsl_movstat_Sn(GSL_MOVSTAT_END_TRUNCATE, x, xSn, w);
gsl_movstat_Qn(GSL_MOVSTAT_END_TRUNCATE, x, xQn, w);
gsl_movstat_sd(GSL_MOVSTAT_END_TRUNCATE, x, xsd, w);

/* scale IQR by factor to approximate standard deviation */
gsl_vector_scale(xiqr, 0.7413);

/* print results */
idx = 0;
for (i = 0; i < N; ++i)
{
    printf("%zu %f %f %f %f %f %f %f\n",
        i,
        gsl_vector_get(x, i),
        sigma[idx],
        gsl_vector_get(xmad, i),
        gsl_vector_get(xiqr, i),
        gsl_vector_get(xSn, i),
        gsl_vector_get(xQn, i),
        gsl_vector_get(xsd, i));

    if (i == N_sigma[idx] - 1)
        ++idx;
}

gsl_vector_free(x);
gsl_vector_free(xmedian);
gsl_vector_free(xmad);
gsl_vector_free(xiqr);
gsl_vector_free(xSn);
gsl_vector_free(xQn);
gsl_vector_free(xsd);
gsl_rng_free(r);

```

(continues on next page)

(continued from previous page)

```

gsl_movstat_free(w);

return 0;
}

```

### 23.12.3 Example 3: User-defined Moving Window

This example program illustrates how a user can define their own moving window function to apply to an input vector. It constructs a random noisy time series of length  $N = 1000$  with some outliers added. Then it applies a moving window trimmed mean to the time series with trim parameter  $\alpha = 0.1$ . The length of the moving window is  $K = 11$ , so the smallest and largest sample of each window is discarded prior to computing the mean. The results are shown in Fig. 23.3.

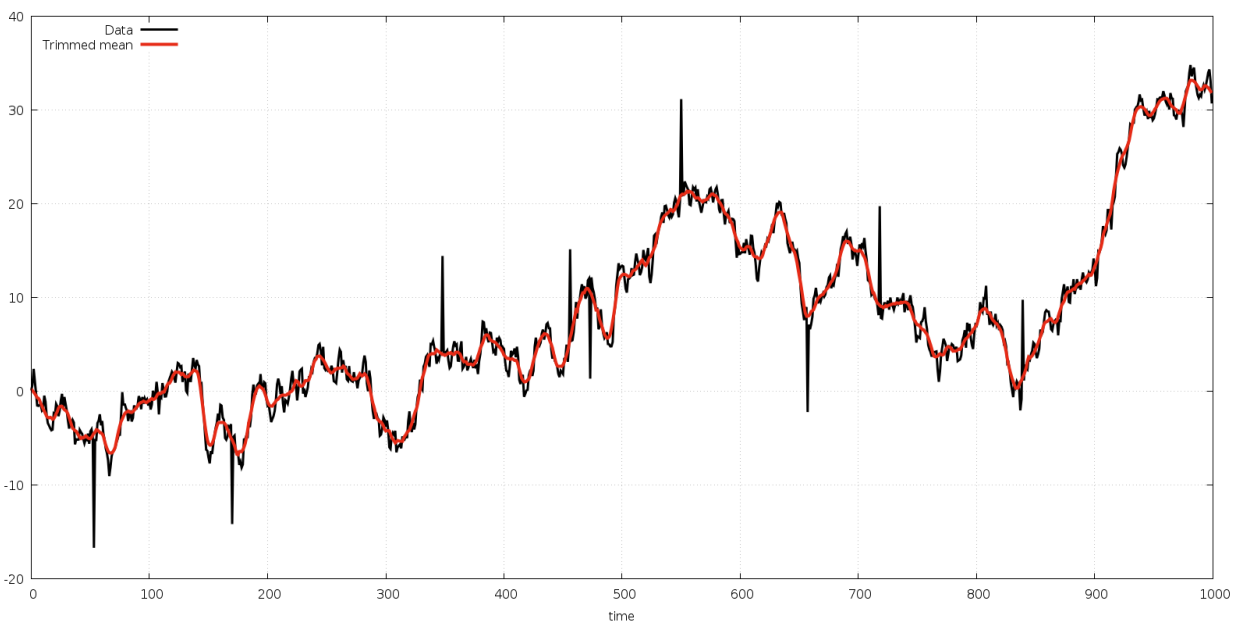


Fig. 23.3: Noisy time series data (black) with moving window trimmed mean (red)

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_movstat.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_statistics.h>

double
func(const size_t n, double x[], void * params)

```

(continues on next page)

(continued from previous page)

```

{
  const double alpha = *(double *) params;

  gsl_sort(x, 1, n);

  return gsl_stats_trmean_from_sorted_data(alpha, x, 1, n);
}

int
main(void)
{
  const size_t N = 1000;          /* length of time series */
  const size_t K = 11;          /* window size */
  double alpha = 0.1;           /* trimmed mean parameter */
  gsl_vector *x = gsl_vector_alloc(N); /* input vector */
  gsl_vector *y = gsl_vector_alloc(N); /* filtered output vector for alpha */
  gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
  gsl_movstat_workspace *w = gsl_movstat_alloc(K);
  gsl_movstat_function F;
  size_t i;
  double sum = 0.0;

  /* generate input signal */
  for (i = 0; i < N; ++i)
  {
    double ui = gsl_ran_gaussian(r, 1.0);
    double outlier = (gsl_rng_uniform(r) < 0.01) ? 10.0 * GSL_SIGN(ui) : 0.0;
    sum += ui;
    gsl_vector_set(x, i, sum + outlier);
  }

  /* apply moving window function */
  F.function = func;
  F.params = &alpha;
  gsl_movstat_apply(GSL_MOVSTAT_END_PADVALUE, &F, x, y, w);

  /* print results */
  for (i = 0; i < N; ++i)
  {
    double xi = gsl_vector_get(x, i);
    double yi = gsl_vector_get(y, i);

    printf("%f %f\n", xi, yi);
  }

  gsl_vector_free(x);
  gsl_vector_free(y);
  gsl_rng_free(r);
  gsl_movstat_free(w);

  return 0;
}

```

## 23.13 References and Further Reading

The following publications are relevant to the algorithms described in this chapter,

- W.Hardle and W. Steiger, *Optimal Median Smoothing*, Appl. Statist., 44 (2), 1995.
- D. Lemire, *Streaming Maximum-Minimum Filter Using No More than Three Comparisons per Element*, Nordic Journal of Computing, 13 (4), 2006 (<https://arxiv.org/abs/cs/0610046>).
- B. P. Welford, *Note on a method for calculating corrected sums of squares and products*, Technometrics, 4 (3), 1962.

## DIGITAL FILTERING

### 24.1 Introduction

The filters discussed in this chapter are based on the following moving data window which is centered on  $i$ -th sample:

$$W_i^H = \{x_{i-H}, \dots, x_i, \dots, x_{i+H}\}$$

Here,  $H$  is a non-negative integer called the *window half-length*, which represents the number of samples before and after sample  $i$ . The total window length is  $K = 2H + 1$ .

### 24.2 Handling Endpoints

When processing samples near the ends of the input signal, there will not be enough samples to fill the window  $W_i^H$  defined above. Therefore the user must specify how to construct the windows near the end points. This is done by passing an input argument of type `gsl_filter_end_t`:

type `gsl_filter_end_t`

This data type specifies how to construct windows near end points and can be selected from the following choices:

#### `GSL_FILTER_END_PADZERO`

With this option, a full window of length  $K$  will be constructed by inserting zeros into the window near the signal end points. Effectively, the input signal is modified to

$$\tilde{x} = \{\underbrace{0, \dots, 0}_{H \text{ zeros}}, x_1, x_2, \dots, x_{n-1}, x_n, \underbrace{0, \dots, 0}_{H \text{ zeros}}\}$$

to ensure a well-defined window for all  $x_i$ .

#### `GSL_FILTER_END_PADVALUE`

With this option, a full window of length  $K$  will be constructed by padding the window with the first and last sample in the input signal. Effectively, the input signal is modified to

$$\tilde{x} = \{\underbrace{x_1, \dots, x_1}_H, x_1, x_2, \dots, x_{n-1}, x_n, \underbrace{x_n, \dots, x_n}_H\}$$

#### `GSL_FILTER_END_TRUNCATE`

With this option, no padding is performed, and the windows are simply truncated as the end points are approached.

## 24.3 Linear Digital Filters

### 24.3.1 Gaussian Filter

The Gaussian filter convolves the input signal with a Gaussian kernel or window. This filter is often used as a smoothing or noise reduction filter. The Gaussian kernel is defined by

$$G(k) = e^{-\frac{1}{2}\left(\frac{k}{(K-1)/2}\right)^2} = e^{-k^2/2\sigma^2}$$

for  $-(K-1)/2 \leq k \leq (K-1)/2$ , and  $K$  is the size of the kernel. The parameter  $\alpha$  specifies the number of standard deviations  $\sigma$  desired in the kernel. So for example setting  $\alpha = 3$  would define a Gaussian window of length  $K$  which spans  $\pm 3\sigma$ . It is often more convenient to specify the parameter  $\alpha$  rather than the standard deviation  $\sigma$  when constructing the kernel, since a fixed value of  $\alpha$  would correspond to the same shape of Gaussian regardless of the size  $K$ . The appropriate value of the standard deviation depends on  $K$  and is related to  $\alpha$  as

$$\sigma = \frac{K-1}{2\alpha}$$

The routines below accept  $\alpha$  as an input argument instead of  $\sigma$ .

The Gaussian filter offers a convenient way of differentiating and smoothing an input signal in a single pass. Using the derivative property of a convolution,

$$\frac{d}{dt}(G * x) = \frac{dG}{dt} * x$$

the input signal  $x(t)$  can be smoothed and differentiated at the same time by convolution with a derivative Gaussian kernel, which can be readily computed from the analytic expression above. The same principle applies to higher order derivatives.

`gsl_filter_gaussian_workspace *gsl_filter_gaussian_alloc(const size_t K)`

This function initializes a workspace for Gaussian filtering using a kernel of size  $K$ . Here,  $H = K/2$ . If  $K$  is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is  $O(K)$ .

`void gsl_filter_gaussian_free(gsl_filter_gaussian_workspace *w)`

This function frees the memory associated with  $w$ .

`int gsl_filter_gaussian(const gsl_filter_end_t endtype, const double alpha, const size_t order, const gsl_vector *x, gsl_vector *y, gsl_filter_gaussian_workspace *w)`

This function applies a Gaussian filter parameterized by *alpha* to the input vector  $x$ , storing the output in  $y$ . The derivative order is specified by *order*, with  $0$  corresponding to a Gaussian,  $1$  corresponding to a first derivative Gaussian, and so on. The parameter *endtype* specifies how the signal end points are handled. It is allowed for  $x = y$  for an in-place filter.

`int gsl_filter_gaussian_kernel(const double alpha, const size_t order, const int normalize, gsl_vector *kernel)`

This function constructs a Gaussian kernel parameterized by *alpha* and stores the output in *kernel*. The parameter *order* specifies the derivative order, with  $0$  corresponding to a Gaussian,  $1$  corresponding to a first derivative Gaussian, and so on. If *normalize* is set to  $1$ , then the kernel will be normalized to sum to one on output. If *normalize* is set to  $0$ , no normalization is performed.

## 24.4 Nonlinear Digital Filters

The nonlinear digital filters described below are based on the window median, which is given by

$$m_i = \text{median} \{W_i^H\} = \text{median} \{x_{i-H}, \dots, x_i, \dots, x_{i+H}\}$$

The median is considered robust to local outliers, unlike the mean. Median filters can preserve sharp edges while at the same removing signal noise, and are used in a wide range of applications.

### 24.4.1 Standard Median Filter

The *standard median filter* (SMF) simply replaces the sample  $x_i$  by the median  $m_i$  of the window  $W_i^H$ : This filter has one tuning parameter given by  $H$ . The standard median filter is considered highly resistant to local outliers and local noise in the data sequence  $\{x_i\}$ .

`gsl_filter_median_workspace *gsl_filter_median_alloc(const size_t K)`

This function initializes a workspace for standard median filtering using a symmetric centered moving window of size  $K$ . Here,  $H = K/2$ . If  $K$  is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is  $O(7K)$ .

`void gsl_filter_median_free(gsl_filter_median_workspace *w)`

This function frees the memory associated with  $w$ .

`int gsl_filter_median(const gsl_filter_end_t endtype, const gsl_vector *x, gsl_vector *y, gsl_filter_median_workspace *w)`

This function applies a standard median filter to the input  $x$ , storing the output in  $y$ . The parameter *endtype* specifies how the signal end points are handled. It is allowed to have  $x = y$  for an in-place filter.

### 24.4.2 Recursive Median Filter

The *recursive median filter* (RMF) is a modification of the SMF to include previous filter outputs in the window before computing the median. The filter's response is

$$y_i = \text{median} (y_{i-H}, \dots, y_{i-1}, x_i, x_{i+1}, \dots, x_{i+H})$$

Sometimes, the SMF must be applied several times in a row to achieve adequate smoothing (i.e. a cascade filter). The RMF, on the other hand, converges to a *root sequence* in one pass, and can sometimes provide a smoother result than several passes of the SMF. A root sequence is an input which is left unchanged by the filter. So there is no need to apply a recursive median filter twice to an input vector.

`gsl_filter_rmedian_workspace *gsl_filter_rmedian_alloc(const size_t K)`

This function initializes a workspace for recursive median filtering using a symmetric centered moving window of size  $K$ . Here,  $H = K/2$ . If  $K$  is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is  $O(K)$ .

`void gsl_filter_rmedian_free(gsl_filter_rmedian_workspace *w)`

This function frees the memory associated with  $w$ .

`int gsl_filter_rmedian(const gsl_filter_end_t endtype, const gsl_vector *x, gsl_vector *y, gsl_filter_rmedian_workspace *w)`

This function applies a recursive median filter to the input  $x$ , storing the output in  $y$ . The parameter *endtype* specifies how the signal end points are handled. It is allowed to have  $x = y$  for an in-place filter.

### 24.4.3 Impulse Detection Filter

Impulsive noise is characterized by short sequences of data points distinct from those in the surrounding neighborhood. This section describes a powerful class of filters, also known as *impulse rejection filters* and *decision-based filters*, designed to detect and remove such outliers from data. The filter's response is given by

$$y_i = \begin{cases} x_i, & |x_i - m_i| \leq tS_i \\ m_i, & |x_i - m_i| > tS_i \end{cases}$$

where  $m_i$  is the median value of the window  $W_i^H$ ,  $S_i$  is a robust estimate of the scatter or dispersion for the window  $W_i^H$ , and  $t$  is a tuning parameter specifying the number of scale factors needed to determine that a point is an outlier. The main idea is that the median  $m_i$  will be unaffected by a small number of outliers in the window, and so a given sample  $x_i$  is tested to determine how far away it is from the median in terms of the local scale estimate  $S_i$ . Samples which are more than  $t$  scale estimates away from the median are labeled as outliers and replaced by the window median  $m_i$ . Samples which are less than  $t$  scale estimates from the median are left unchanged by the filter.

Note that when  $t = 0$ , the impulse detection filter is equivalent to the standard median filter. When  $t \rightarrow \infty$ , it becomes the identity filter. This means the impulse detection filter can be viewed as a “less aggressive” version of the standard median filter, becoming less aggressive as  $t$  is increased. Note that this filter modifies only samples identified as outliers, while the standard median filter changes all samples to the local median, regardless of whether they are outliers. This fact, plus the additional flexibility offered by the additional tuning parameter  $t$  can make the impulse detection filter a better choice for some applications.

It is important to have a robust and accurate scale estimate  $S_i$  in order to detect impulse outliers even in the presence of noise. The window standard deviation is not typically a good choice, as it can be significantly perturbed by the presence of even one outlier. GSL offers the following choices (specified by a parameter of type `gsl_filter_scale_t`) for computing the scale estimate  $S_i$ , all of which are robust to the presence of impulse outliers.

type `gsl_filter_scale_t`

This type specifies how the scale estimate  $S_i$  of the window  $W_i^H$  is calculated.

#### **GSL\_FILTER\_SCALE\_MAD**

This option specifies the median absolute deviation (MAD) scale estimate, defined by

$$S_i = 1.4826 \times \text{median} \{|W_i^H - m_i|\}$$

This choice of scale estimate is also known as the *Hampel filter* in the statistical literature. See [here](#) for more information.

#### **GSL\_FILTER\_SCALE\_IQR**

This option specifies the interquartile range (IQR) scale estimate, defined as the difference between the 75th and 25th percentiles of the window  $W_i^H$ ,

$$S_i = 0.7413 (Q_{0.75} - Q_{0.25})$$

where  $Q_p$  is the  $p$ -quantile of the window  $W_i^H$ . The idea is to throw away the largest and smallest 25% of the window samples (where the outliers would be), and estimate a scale from the middle 50%. The factor 0.7413 provides an unbiased estimate of the standard deviation for Gaussian data.

#### **GSL\_FILTER\_SCALE\_SN**

This option specifies the so-called  $S_n$  statistic proposed by Croux and Rousseeuw. See [here](#) for more information.

#### **GSL\_FILTER\_SCALE\_QN**

This option specifies the so-called  $Q_n$  statistic proposed by Croux and Rousseeuw. See [here](#) for more information.



**Warning:** While the scale estimates defined above are much less sensitive to outliers than the standard deviation, they can suffer from an effect called *implosion*. The standard deviation of a window  $W_i^H$  will be zero if and only if all samples in the window are equal. However, it is possible for the MAD of a window to be zero even if all the samples in the window are not equal. For example, if  $K/2 + 1$  or more of the  $K$  samples in the window are equal to some value  $x^*$ , then the window median will be equal to  $x^*$ . Consequently, at least  $K/2 + 1$  of the absolute deviations  $|x_j - x^*|$  will be zero, and so the MAD will be zero. In such a case, the Hampel filter will act like the standard median filter regardless of the value of  $t$ . Caution should also be exercised if dividing by  $S_i$ .

`gsl_filter_impulse_workspace *gsl_filter_impulse_alloc(const size_t K)`

This function initializes a workspace for impulse detection filtering using a symmetric moving window of size  $K$ . Here,  $H = K/2$ . If  $K$  is even, it is rounded up to the next odd integer to ensure a symmetric window. The size of the workspace is  $O(6K)$ .

`void gsl_filter_impulse_free(gsl_filter_impulse_workspace *w)`

This function frees the memory associated with  $w$ .

`int gsl_filter_impulse(const gsl_filter_end_t endtype, const gsl_filter_scale_t scale_type, const double t, const gsl_vector *x, gsl_vector *y, gsl_vector *xmedian, gsl_vector *xsigma, size_t *noutlier, gsl_vector_int *ioutlier, gsl_filter_impulse_workspace *w)`

These functions apply an impulse detection filter to the input vector  $x$ , storing the filtered output in  $y$ . The tuning parameter  $t$  is provided in  $t$ . The window medians  $m_i$  are stored in  $xmedian$  and the  $S_i$  are stored in  $xsigma$  on output. The number of outliers detected is stored in  $noutlier$  on output, while the locations of flagged outliers are stored in the boolean array  $ioutlier$ . The input  $ioutlier$  may be NULL if not desired. It is allowed to have  $x = y$  for an in-place filter.

## 24.5 Examples

### 24.5.1 Gaussian Example 1

This example program illustrates the Gaussian filter applied to smoothing a time series of length  $N = 500$  with a kernel size of  $K = 51$ . Three filters are applied with parameters  $\alpha = 0.5, 3, 10$ . The results are shown in Fig. 24.1.

We see that the filter corresponding to  $\alpha = 0.5$  applies the most smoothing, while  $\alpha = 10$  corresponds to the least amount of smoothing. The program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
    const size_t N = 500;           /* length of time series */
    const size_t K = 51;           /* window size */
    const double alpha[3] = { 0.5, 3.0, 10.0 }; /* alpha values */
    gsl_vector *x = gsl_vector_alloc(N); /* input vector */
    gsl_vector *y1 = gsl_vector_alloc(N); /* filtered output vector for alpha */
}
```

(continues on next page)

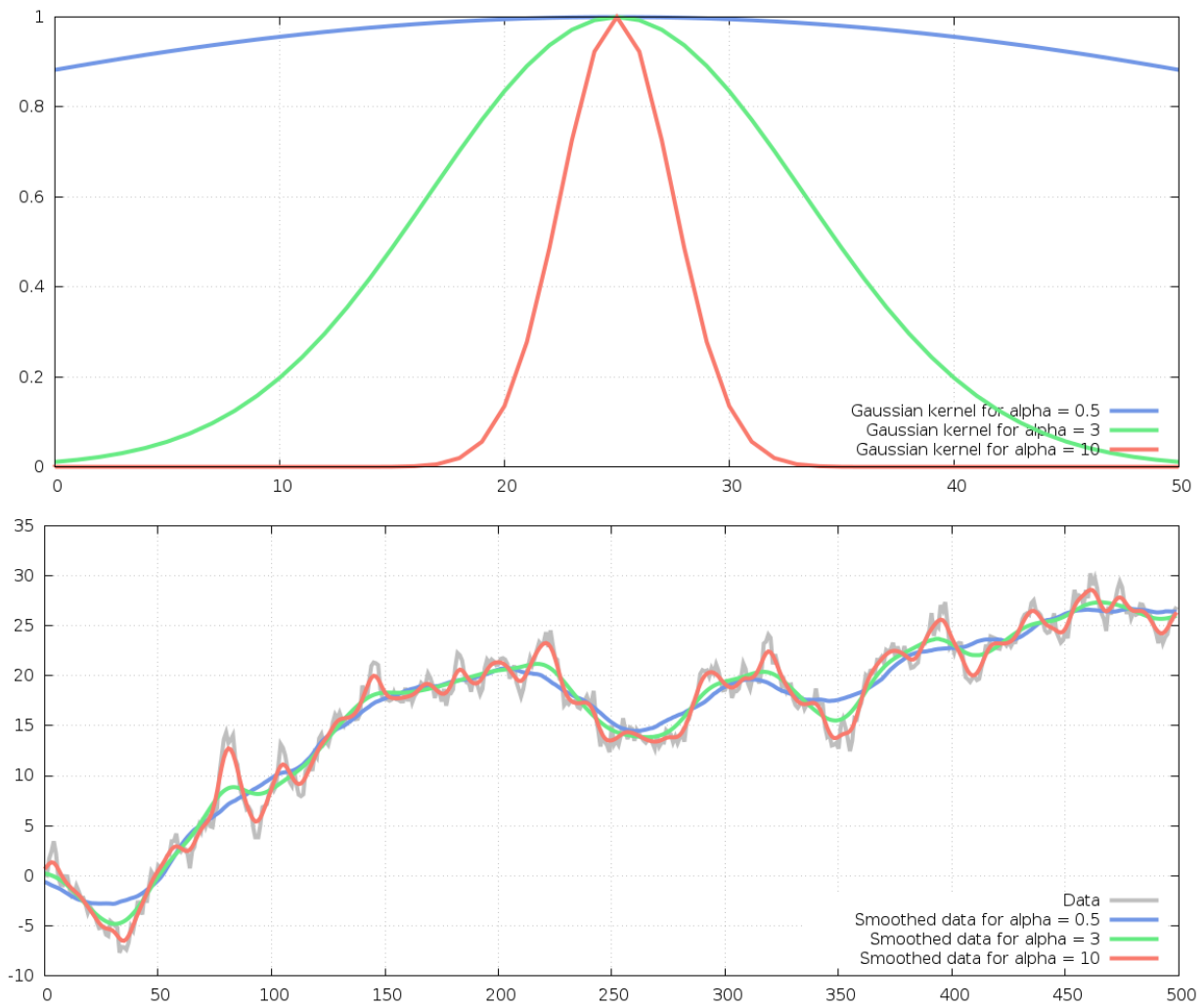


Fig. 24.1: Top panel: Gaussian kernels (unnormalized) for  $\alpha = 0.5, 3, 10$ . Bottom panel: Time series (gray) with Gaussian filter output for same  $\alpha$  values.

(continued from previous page)

```

gsl_vector *y2 = gsl_vector_alloc(N);          /* filtered output vector for alpha2 */
gsl_vector *y3 = gsl_vector_alloc(N);          /* filtered output vector for alpha3 */
gsl_vector *k1 = gsl_vector_alloc(K);          /* Gaussian kernel for alpha1 */
gsl_vector *k2 = gsl_vector_alloc(K);          /* Gaussian kernel for alpha2 */
gsl_vector *k3 = gsl_vector_alloc(K);          /* Gaussian kernel for alpha3 */
gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
gsl_filter_gaussian_workspace *gauss_p = gsl_filter_gaussian_alloc(K);
size_t i;
double sum = 0.0;

/* generate input signal */
for (i = 0; i < N; ++i)
{
    double ui = gsl_ran_gaussian(r, 1.0);
    sum += ui;
    gsl_vector_set(x, i, sum);
}

/* compute kernels without normalization */
gsl_filter_gaussian_kernel(alpha[0], 0, 0, k1);
gsl_filter_gaussian_kernel(alpha[1], 0, 0, k2);
gsl_filter_gaussian_kernel(alpha[2], 0, 0, k3);

/* apply filters */
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha[0], 0, x, y1, gauss_p);
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha[1], 0, x, y2, gauss_p);
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha[2], 0, x, y3, gauss_p);

/* print kernels */
for (i = 0; i < K; ++i)
{
    double k1i = gsl_vector_get(k1, i);
    double k2i = gsl_vector_get(k2, i);
    double k3i = gsl_vector_get(k3, i);

    printf("%e %e %e\n", k1i, k2i, k3i);
}

printf("\n\n");

/* print filter results */
for (i = 0; i < N; ++i)
{
    double xi = gsl_vector_get(x, i);
    double y1i = gsl_vector_get(y1, i);
    double y2i = gsl_vector_get(y2, i);
    double y3i = gsl_vector_get(y3, i);

    printf("%.12e %.12e %.12e %.12e\n", xi, y1i, y2i, y3i);
}

gsl_vector_free(x);

```

(continues on next page)

(continued from previous page)

```

gsl_vector_free(y1);
gsl_vector_free(y2);
gsl_vector_free(y3);
gsl_vector_free(k1);
gsl_vector_free(k2);
gsl_vector_free(k3);
gsl_rng_free(r);
gsl_filter_gaussian_free(gauss_p);

return 0;
}

```

## 24.5.2 Gaussian Example 2

A common application of the Gaussian filter is to detect edges, or sudden jumps, in a noisy input signal. It is used both for 1D edge detection in time series, as well as 2D edge detection in images. Here we will examine a noisy time series of length  $N = 1000$  with a single edge. The input signal is defined as

$$x(n) = e(n) + \begin{cases} 0, & n \leq N/2 \\ 0.5, & n > N/2 \end{cases}$$

where  $e(n)$  is Gaussian random noise. The program smooths the input signal with order 0, 1, and 2 Gaussian filters of length  $K = 61$  with  $\alpha = 3$ . For comparison, the program also computes finite differences of the input signal without smoothing. The results are shown in Fig. 24.2.

The finite difference approximation of the first derivative (second row) shows the common problem with differentiating a noisy signal. The noise is amplified and makes it extremely difficult to detect the sharp gradient at sample 500. The third row shows the first order Gaussian smoothed signal with a clear peak at the location of the edge. Alternatively, one could examine the second order Gaussian smoothed signal (fourth row) and look for zero crossings to determine the edge location.

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
    const size_t N = 1000;          /* length of time series */
    const size_t K = 61;           /* window size */
    const double alpha = 3.0;      /* Gaussian kernel has +/- 3 standard
    deviations */
    gsl_vector *x = gsl_vector_alloc(N); /* input vector */
    gsl_vector *y = gsl_vector_alloc(N); /* filtered output vector */
    gsl_vector *dy = gsl_vector_alloc(N); /* first derivative filtered vector */

```

(continues on next page)

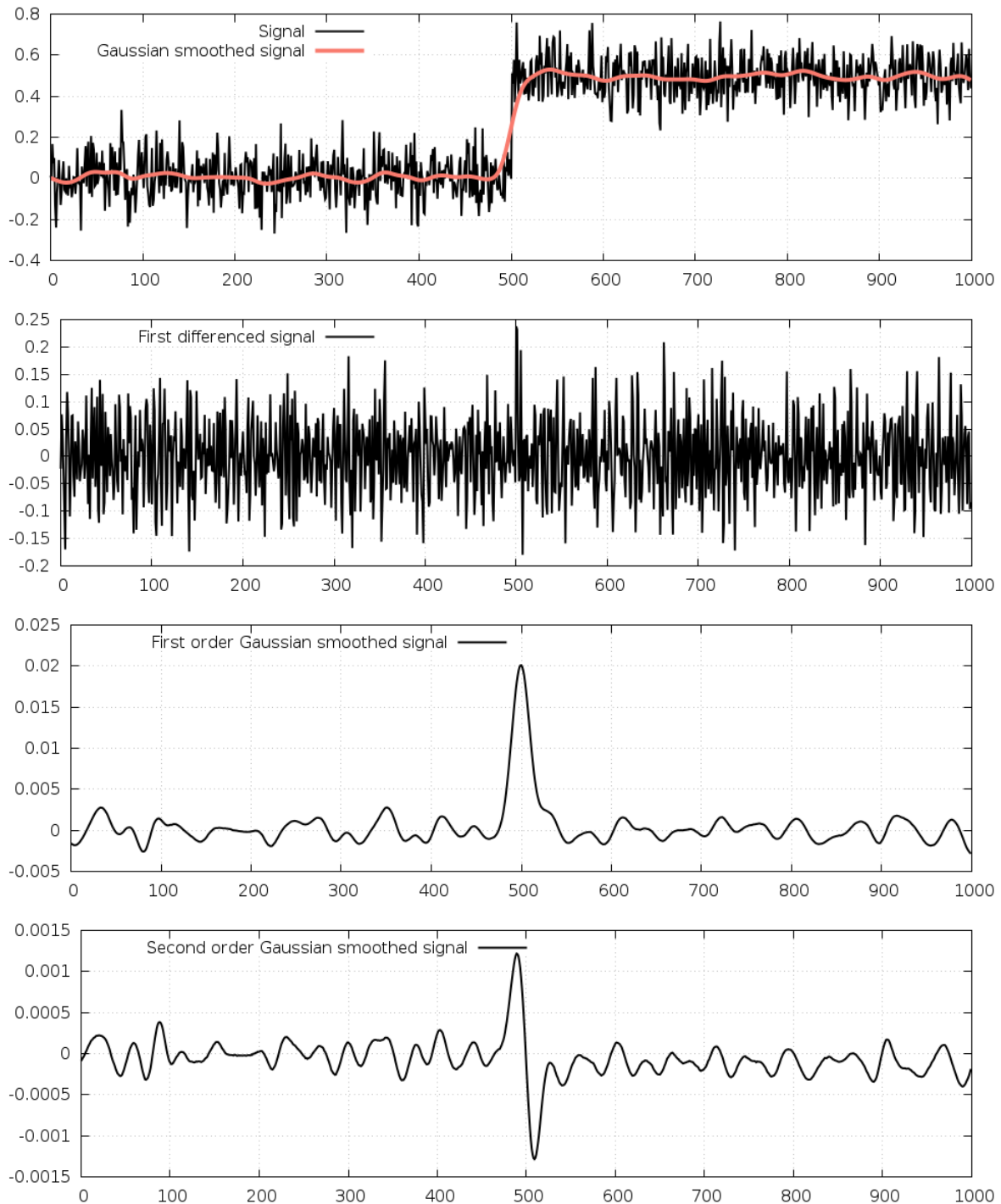


Fig. 24.2: Top row: original input signal  $x(n)$  (black) with Gaussian smoothed signal in red. Second row: First finite differences of input signal. Third row: Input signal smoothed with a first order Gaussian filter. Fourth row: Input signal smoothed with a second order Gaussian filter.

(continued from previous page)

```

gsl_vector *d2y = gsl_vector_alloc(N);          /* second derivative filtered vector */
gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
gsl_filter_gaussian_workspace *gauss_p = gsl_filter_gaussian_alloc(K);
size_t i;

/* generate input signal */
for (i = 0; i < N; ++i)
{
    double xi = (i > N / 2) ? 0.5 : 0.0;
    double ei = gsl_ran_gaussian(r, 0.1);

    gsl_vector_set(x, i, xi + ei);
}

/* apply filters */
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha, 0, x, y, gauss_p);
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha, 1, x, dy, gauss_p);
gsl_filter_gaussian(GSL_FILTER_END_PADVALUE, alpha, 2, x, d2y, gauss_p);

/* print results */
for (i = 0; i < N; ++i)
{
    double xi = gsl_vector_get(x, i);
    double yi = gsl_vector_get(y, i);
    double dyi = gsl_vector_get(dy, i);
    double d2yi = gsl_vector_get(d2y, i);
    double dxi;

    /* compute finite difference of x vector */
    if (i == 0)
        dxi = gsl_vector_get(x, i + 1) - xi;
    else if (i == N - 1)
        dxi = gsl_vector_get(x, i) - gsl_vector_get(x, i - 1);
    else
        dxi = 0.5 * (gsl_vector_get(x, i + 1) - gsl_vector_get(x, i - 1));

    printf("%.12e %.12e %.12e %.12e %.12e\n",
           xi,
           yi,
           dxi,
           dyi,
           d2yi);
}

gsl_vector_free(x);
gsl_vector_free(y);
gsl_vector_free(dy);
gsl_vector_free(d2y);
gsl_rng_free(r);
gsl_filter_gaussian_free(gauss_p);

return 0;

```

(continues on next page)

(continued from previous page)

}

### 24.5.3 Square Wave Signal Example

The following example program illustrates the median filters on a noisy square wave signal. Median filters are well known for preserving sharp edges in the input signal while reducing noise. The program constructs a 5 Hz square wave signal with Gaussian noise added. Then the signal is filtered with a standard median filter and recursive median filter using a symmetric window of length  $K = 7$ . The results are shown in Fig. 24.3.

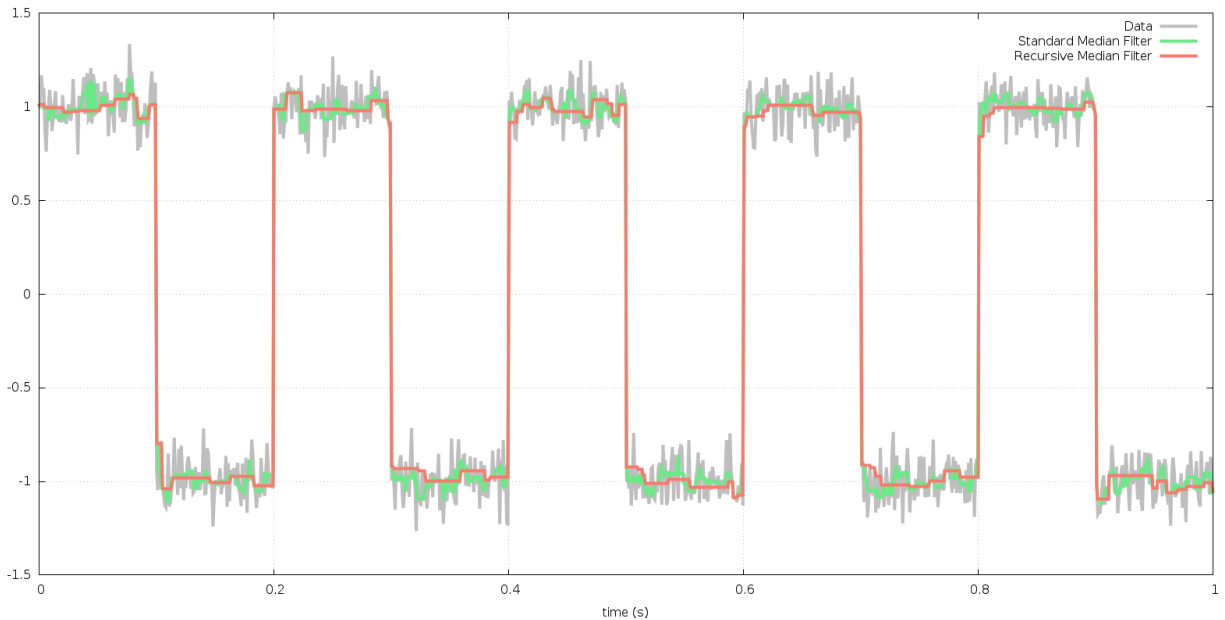


Fig. 24.3: Original time series is in gray. The standard median filter output is in green and the recursive median filter output is in red.

Both filters preserve the sharp signal edges while reducing the noise. The recursive median filter achieves a smoother result than the standard median filter. The “blocky” nature of the output is characteristic of all median filters. The program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
    const size_t N = 1000;           /* length of time series */
    const size_t K = 7;             /* window size */
```

(continues on next page)

(continued from previous page)

```

const double f = 5.0;                               /* frequency of square wave in Hz */
↪ /*
gsl_filter_median_workspace *median_p = gsl_filter_median_alloc(K);
gsl_filter_rmedian_workspace *rmedian_p = gsl_filter_rmedian_alloc(K);
gsl_vector *t = gsl_vector_alloc(N);                /* time */
gsl_vector *x = gsl_vector_alloc(N);                /* input vector */
gsl_vector *y_median = gsl_vector_alloc(N);         /* median filtered output */
gsl_vector *y_rmedian = gsl_vector_alloc(N);        /* recursive median filtered
↪ output */
gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
size_t i;

/* generate input signal */
for (i = 0; i < N; ++i)
{
    double ti = (double) i / (N - 1.0);
    double tmp = sin(2.0 * M_PI * f * ti);
    double xi = (tmp >= 0.0) ? 1.0 : -1.0;
    double ei = gsl_ran_gaussian(r, 0.1);

    gsl_vector_set(t, i, ti);
    gsl_vector_set(x, i, xi + ei);
}

gsl_filter_median(GSL_FILTER_END_PADVALUE, x, y_median, median_p);
gsl_filter_rmedian(GSL_FILTER_END_PADVALUE, x, y_rmedian, rmedian_p);

/* print results */
for (i = 0; i < N; ++i)
{
    double ti = gsl_vector_get(t, i);
    double xi = gsl_vector_get(x, i);
    double medi = gsl_vector_get(y_median, i);
    double rmedi = gsl_vector_get(y_rmedian, i);

    printf("%f %f %f %f\n",
           ti,
           xi,
           medi,
           rmedi);
}

gsl_vector_free(t);
gsl_vector_free(x);
gsl_vector_free(y_median);
gsl_vector_free(y_rmedian);
gsl_rng_free(r);
gsl_filter_median_free(median_p);

return 0;
}

```



## 24.5.4 Impulse Detection Example

The following example program illustrates the impulse detection filter. First, it constructs a sinusoid signal of length  $N = 1000$  with Gaussian noise added. Then, about 1% of the data are perturbed to represent large outliers. An impulse detecting filter is applied with a window size  $K = 25$  and tuning parameter  $t = 4$ , using the  $Q_n$  statistic as the robust measure of scale. The results are plotted in Fig. 24.4.

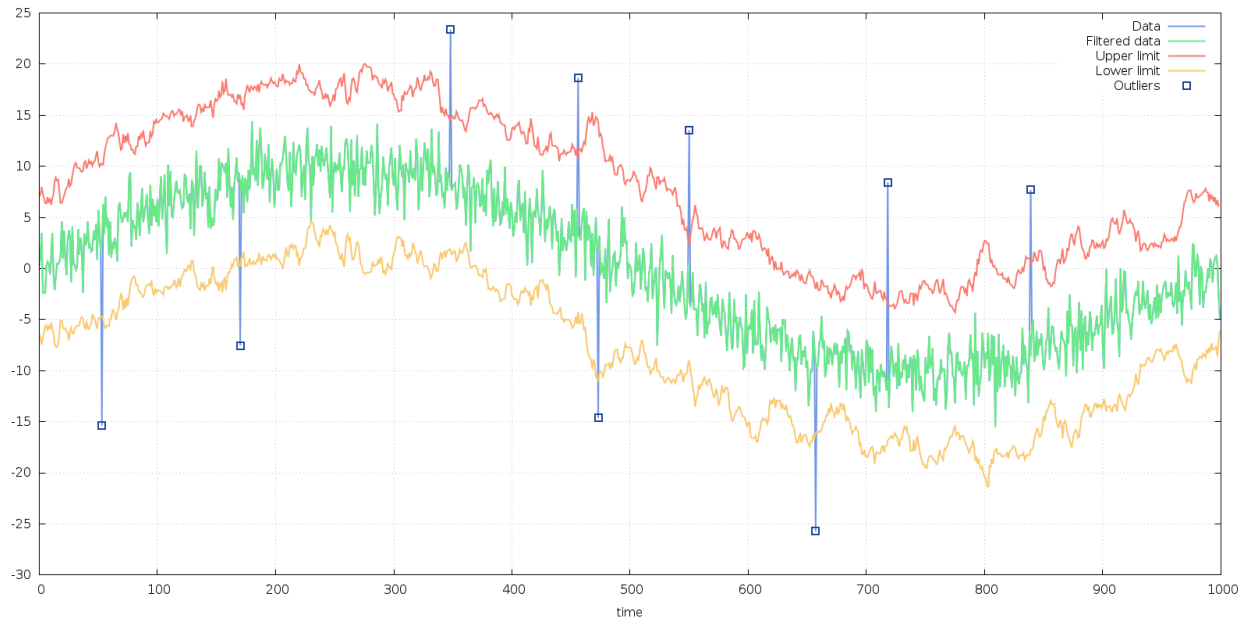


Fig. 24.4: Original time series is in blue, filter output is in green, upper and lower intervals for detecting outliers are in red and yellow respectively. Detected outliers are marked with squares.

The program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_filter.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>

int
main(void)
{
    const size_t N = 1000;          /* length of time series */
    const size_t K = 25;           /* window size */
    const double t = 4.0;          /* number of scale factors for
    ↪ outlier detection */
    gsl_vector *x = gsl_vector_alloc(N); /* input vector */
    gsl_vector *y = gsl_vector_alloc(N); /* output (filtered) vector */
    gsl_vector *xmedian = gsl_vector_alloc(N); /* window medians */
    gsl_vector *xsigma = gsl_vector_alloc(N); /* window scale estimates */
    gsl_vector_int *ioutlier = gsl_vector_int_alloc(N); /* outlier detected? */
```

(continues on next page)

```
gsl_filter_impulse_workspace * w = gsl_filter_impulse_alloc(K);
gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
size_t noutlier;
size_t i;

/* generate input signal */
for (i = 0; i < N; ++i)
{
    double xi = 10.0 * sin(2.0 * M_PI * i / (double) N);
    double ei = gsl_ran_gaussian(r, 2.0);
    double u = gsl_rng_uniform(r);
    double outlier = (u < 0.01) ? 15.0 * GSL_SIGN(ei) : 0.0;

    gsl_vector_set(x, i, xi + ei + outlier);
}

/* apply impulse detection filter */
gsl_filter_impulse(GSL_FILTER_END_TRUNCATE, GSL_FILTER_SCALE_QN, t, x, y,
                  xmedian, xsigma, &noutlier, ioutlier, w);

/* print results */
for (i = 0; i < N; ++i)
{
    double xi = gsl_vector_get(x, i);
    double yi = gsl_vector_get(y, i);
    double xmedi = gsl_vector_get(xmedian, i);
    double xsigmai = gsl_vector_get(xsigma, i);
    int outlier = gsl_vector_int_get(ioutlier, i);

    printf("%zu %f %f %f %f %d\n",
           i,
           xi,
           yi,
           xmedi + t * xsigmai,
           xmedi - t * xsigmai,
           outlier);
}

gsl_vector_free(x);
gsl_vector_free(y);
gsl_vector_free(xmedian);
gsl_vector_free(xsigma);
gsl_vector_int_free(ioutlier);
gsl_filter_impulse_free(w);
gsl_rng_free(r);

return 0;
}
```

## 24.6 References and Further Reading

The following publications are relevant to the algorithms described in this chapter,

- F. J. Harris, *On the use of windows for harmonic analysis with the discrete Fourier transform*, Proceedings of the IEEE, 66 (1), 1978.
- S-J. Ko, Y-H. Lee, and A. T. Fam. *Efficient implementation of one-dimensional recursive median filters*, IEEE transactions on circuits and systems 37.11 (1990): 1447-1450.
- R. K. Pearson and M. Gabbouj, *Nonlinear Digital Filtering with Python: An Introduction*. CRC Press, 2015.



## HISTOGRAMS

This chapter describes functions for creating histograms. Histograms provide a convenient way of summarizing the distribution of a set of data. A histogram consists of a set of *bins* which count the number of events falling into a given range of a continuous variable  $x$ . In GSL the bins of a histogram contain floating-point numbers, so they can be used to record both integer and non-integer distributions. The bins can use arbitrary sets of ranges (uniformly spaced bins are the default). Both one and two-dimensional histograms are supported.

Once a histogram has been created it can also be converted into a probability distribution function. The library provides efficient routines for selecting random samples from probability distributions. This can be useful for generating simulations based on real data.

The functions are declared in the header files `gsl_histogram.h` and `gsl_histogram2d.h`.

### 25.1 The histogram struct

A histogram is defined by the following struct,

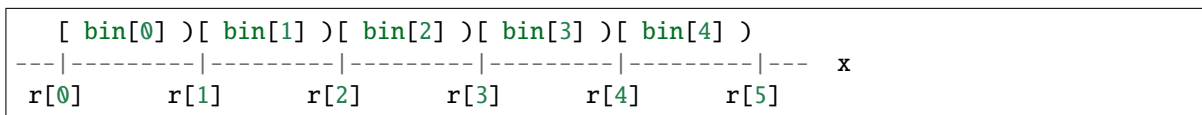
type `gsl_histogram`

size_t n	This is the number of histogram bins
double * range	The ranges of the bins are stored in an array of n+1 elements pointed to by range.
double * bin	The counts for each bin are stored in an array of n elements pointed to by bin. The bins are floating-point numbers, so you can increment them by non-integer values if necessary.

The range for `bin[i]` is given by `range[i]` to `range[i+1]`. For  $n$  bins there are  $n+1$  entries in the array `range`. Each bin is inclusive at the lower end and exclusive at the upper end. Mathematically this means that the bins are defined by the following inequality,

$$\text{bin}[i] \text{ corresponds to } \text{range}[i] \leq x < \text{range}[i+1]$$

Here is a diagram of the correspondence between ranges and bins on the number-line for  $x$ :



In this picture the values of the `range` array are denoted by  $r$ . On the left-hand side of each bin the square bracket `[` denotes an inclusive lower bound ( $r \leq x$ ), and the round parentheses `)` on the right-hand side denote an exclusive upper bound ( $x < r$ ). Thus any samples which fall on the upper end of the histogram are excluded. If you want to include this value for the last bin you will need to add an extra bin to your histogram.

The `gsl_histogram` struct and its associated functions are defined in the header file `gsl_histogram.h`.

## 25.2 Histogram allocation

The functions for allocating memory to a histogram follow the style of `malloc()` and `free()`. In addition they also perform their own error checking. If there is insufficient memory available to allocate a histogram then the functions call the error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every histogram alloc.

`gsl_histogram *gsl_histogram_alloc(size_t n)`

This function allocates memory for a histogram with  $n$  bins, and returns a pointer to a newly created `gsl_histogram` struct. If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`. The bins and ranges are not initialized, and should be prepared using one of the range-setting functions below in order to make the histogram ready for use.

`int gsl_histogram_set_ranges(gsl_histogram *h, const double range[], size_t size)`

This function sets the ranges of the existing histogram  $h$  using the array `range` of size `size`. The values of the histogram bins are reset to zero. The `range` array should contain the desired bin limits. The ranges can be arbitrary, subject to the restriction that they are monotonically increasing.

The following example shows how to create a histogram with logarithmic bins with ranges [1,10), [10,100) and [100,1000):

```
gsl_histogram * h = gsl_histogram_alloc (3);

/* bin[0] covers the range 1 <= x < 10 */
/* bin[1] covers the range 10 <= x < 100 */
/* bin[2] covers the range 100 <= x < 1000 */

double range[4] = { 1.0, 10.0, 100.0, 1000.0 };

gsl_histogram_set_ranges (h, range, 4);
```

Note that the size of the `range` array should be defined to be one element bigger than the number of bins. The additional element is required for the upper value of the final bin.

`int gsl_histogram_set_ranges_uniform(gsl_histogram *h, double xmin, double xmax)`

This function sets the ranges of the existing histogram  $h$  to cover the range  $xmin$  to  $xmax$  uniformly. The values of the histogram bins are reset to zero. The bin ranges are shown in the table below,

bin[0]	corresponds to	$xmin \leq x < xmin + d$
bin[1]	corresponds to	$xmin + d \leq x < xmin + 2d$
...	...	...
bin[n-1]	corresponds to	$xmin + (n - 1)d \leq x < xmax$

where  $d$  is the bin spacing,  $d = (xmax - xmin)/n$ .

`void gsl_histogram_free(gsl_histogram *h)`

This function frees the histogram  $h$  and all of the memory associated with it.

## 25.3 Copying Histograms

int **gsl\_histogram\_memcpy**(*gsl\_histogram* \*dest, const *gsl\_histogram* \*src)

This function copies the histogram *src* into the pre-existing histogram *dest*, making *dest* into an exact copy of *src*. The two histograms must be of the same size.

*gsl\_histogram* \***gsl\_histogram\_clone**(const *gsl\_histogram* \*src)

This function returns a pointer to a newly created histogram which is an exact copy of the histogram *src*.

## 25.4 Updating and accessing histogram elements

There are two ways to access histogram bins, either by specifying an *x* coordinate or by using the bin-index directly. The functions for accessing the histogram through *x* coordinates use a binary search to identify the bin which covers the appropriate range.

int **gsl\_histogram\_increment**(*gsl\_histogram* \*h, double x)

This function updates the histogram *h* by adding one (1.0) to the bin whose range contains the coordinate *x*.

If *x* lies in the valid range of the histogram then the function returns zero to indicate success. If *x* is less than the lower limit of the histogram then the function returns *GSL\_EDOM*, and none of bins are modified. Similarly, if the value of *x* is greater than or equal to the upper limit of the histogram then the function returns *GSL\_EDOM*, and none of the bins are modified. The error handler is not called, however, since it is often necessary to compute histograms for a small range of a larger dataset, ignoring the values outside the range of interest.

int **gsl\_histogram\_accumulate**(*gsl\_histogram* \*h, double x, double weight)

This function is similar to *gsl\_histogram\_increment()* but increases the value of the appropriate bin in the histogram *h* by the floating-point number *weight*.

double **gsl\_histogram\_get**(const *gsl\_histogram* \*h, size\_t i)

This function returns the contents of the *i*-th bin of the histogram *h*. If *i* lies outside the valid range of indices for the histogram then the error handler is called with an error code of *GSL\_EDOM* and the function returns 0.

int **gsl\_histogram\_get\_range**(const *gsl\_histogram* \*h, size\_t i, double \*lower, double \*upper)

This function finds the upper and lower range limits of the *i*-th bin of the histogram *h*. If the index *i* is valid then the corresponding range limits are stored in *lower* and *upper*. The lower limit is inclusive (i.e. events with this coordinate are included in the bin) and the upper limit is exclusive (i.e. events with the coordinate of the upper limit are excluded and fall in the neighboring higher bin, if it exists). The function returns 0 to indicate success. If *i* lies outside the valid range of indices for the histogram then the error handler is called and the function returns an error code of *GSL\_EDOM*.

double **gsl\_histogram\_max**(const *gsl\_histogram* \*h)

double **gsl\_histogram\_min**(const *gsl\_histogram* \*h)

size\_t **gsl\_histogram\_bins**(const *gsl\_histogram* \*h)

These functions return the maximum upper and minimum lower range limits and the number of bins of the histogram *h*. They provide a way of determining these values without accessing the *gsl\_histogram* struct directly.

void **gsl\_histogram\_reset**(*gsl\_histogram* \*h)

This function resets all the bins in the histogram *h* to zero.

## 25.5 Searching histogram ranges

The following functions are used by the access and update routines to locate the bin which corresponds to a given  $x$  coordinate.

int **gsl\_histogram\_find**(const *gsl\_histogram* \*h, double x, size\_t \*i)

This function finds and sets the index  $i$  to the bin number which covers the coordinate  $x$  in the histogram  $h$ . The bin is located using a binary search. The search includes an optimization for histograms with uniform range, and will return the correct bin immediately in this case. If  $x$  is found in the range of the histogram then the function sets the index  $i$  and returns `GSL_SUCCESS`. If  $x$  lies outside the valid range of the histogram then the function returns `GSL_EDOM` and the error handler is invoked.

## 25.6 Histogram Statistics

double **gsl\_histogram\_max\_val**(const *gsl\_histogram* \*h)

This function returns the maximum value contained in the histogram bins.

size\_t **gsl\_histogram\_max\_bin**(const *gsl\_histogram* \*h)

This function returns the index of the bin containing the maximum value. In the case where several bins contain the same maximum value the smallest index is returned.

double **gsl\_histogram\_min\_val**(const *gsl\_histogram* \*h)

This function returns the minimum value contained in the histogram bins.

size\_t **gsl\_histogram\_min\_bin**(const *gsl\_histogram* \*h)

This function returns the index of the bin containing the minimum value. In the case where several bins contain the same minimum value the smallest index is returned.

double **gsl\_histogram\_mean**(const *gsl\_histogram* \*h)

This function returns the mean of the histogrammed variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation. The accuracy of the result is limited by the bin width.

double **gsl\_histogram\_sigma**(const *gsl\_histogram* \*h)

This function returns the standard deviation of the histogrammed variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation. The accuracy of the result is limited by the bin width.

double **gsl\_histogram\_sum**(const *gsl\_histogram* \*h)

This function returns the sum of all bin values. Negative bin values are included in the sum.

## 25.7 Histogram Operations

int **gsl\_histogram\_equal\_bins\_p**(const *gsl\_histogram* \*h1, const *gsl\_histogram* \*h2)

This function returns 1 if the all of the individual bin ranges of the two histograms are identical, and 0 otherwise.

int **gsl\_histogram\_add**(*gsl\_histogram* \*h1, const *gsl\_histogram* \*h2)

This function adds the contents of the bins in histogram  $h2$  to the corresponding bins of histogram  $h1$ , i.e.  $h'_1(i) = h_1(i) + h_2(i)$ . The two histograms must have identical bin ranges.

int **gsl\_histogram\_sub**(*gsl\_histogram* \*h1, const *gsl\_histogram* \*h2)

This function subtracts the contents of the bins in histogram  $h2$  from the corresponding bins of histogram  $h1$ , i.e.  $h'_1(i) = h_1(i) - h_2(i)$ . The two histograms must have identical bin ranges.



int **gsl\_histogram\_mul**(*gsl\_histogram* \*h1, const *gsl\_histogram* \*h2)

This function multiplies the contents of the bins of histogram *h1* by the contents of the corresponding bins in histogram *h2*, i.e.  $h'_1(i) = h_1(i) * h_2(i)$ . The two histograms must have identical bin ranges.

int **gsl\_histogram\_div**(*gsl\_histogram* \*h1, const *gsl\_histogram* \*h2)

This function divides the contents of the bins of histogram *h1* by the contents of the corresponding bins in histogram *h2*, i.e.  $h'_1(i) = h_1(i)/h_2(i)$ . The two histograms must have identical bin ranges.

int **gsl\_histogram\_scale**(*gsl\_histogram* \*h, double scale)

This function multiplies the contents of the bins of histogram *h* by the constant *scale*, i.e.

$$h'_1(i) = h_1(i) * scale$$

int **gsl\_histogram\_shift**(*gsl\_histogram* \*h, double offset)

This function shifts the contents of the bins of histogram *h* by the constant *offset*, i.e.

$$h'_1(i) = h_1(i) + offset$$

## 25.8 Reading and writing histograms

The library provides functions for reading and writing histograms to a file as binary data or formatted text.

int **gsl\_histogram\_fwrite**(FILE \*stream, const *gsl\_histogram* \*h)

This function writes the ranges and bins of the histogram *h* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl\_histogram\_fread**(FILE \*stream, *gsl\_histogram* \*h)

This function reads into the histogram *h* from the open stream *stream* in binary format. The histogram *h* must be preallocated with the correct size since the function uses the number of bins in *h* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl\_histogram\_fprintf**(FILE \*stream, const *gsl\_histogram* \*h, const char \*range\_format, const char \*bin\_format)

This function writes the ranges and bins of the histogram *h* line-by-line to the stream *stream* using the format specifiers *range\_format* and *bin\_format*. These should be one of the `%g`, `%e` or `%f` formats for floating point numbers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file. The histogram output is formatted in three columns, and the columns are separated by spaces, like this:

```
range[0] range[1] bin[0]
range[1] range[2] bin[1]
range[2] range[3] bin[2]
....
range[n-1] range[n] bin[n-1]
```

The values of the ranges are formatted using *range\_format* and the value of the bins are formatted using *bin\_format*. Each line contains the lower and upper limit of the range of the bins and the value of the bin itself. Since the upper limit of one bin is the lower limit of the next there is duplication of these values between lines but this allows the histogram to be manipulated with line-oriented tools.

int **gsl\_histogram\_fscanf**(FILE \*stream, *gsl\_histogram* \*h)

This function reads formatted data from the stream *stream* into the histogram *h*. The data is assumed to be in the three-column format used by *gsl\_histogram\_fprintf*(). The histogram *h* must be preallocated with the correct length since the function uses the size of *h* to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

## 25.9 Resampling from histograms

A histogram made by counting events can be regarded as a measurement of a probability distribution. Allowing for statistical error, the height of each bin represents the probability of an event where the value of  $x$  falls in the range of that bin. The probability distribution function has the one-dimensional form  $p(x)dx$  where,

$$p(x) = n_i / (N w_i)$$

In this equation  $n_i$  is the number of events in the bin which contains  $x$ ,  $w_i$  is the width of the bin and  $N$  is the total number of events. The distribution of events within each bin is assumed to be uniform.

## 25.10 The histogram probability distribution struct

The probability distribution function for a histogram consists of a set of *bins* which measure the probability of an event falling into a given range of a continuous variable  $x$ . A probability distribution function is defined by the following struct, which actually stores the cumulative probability distribution function. This is the natural quantity for generating samples via the inverse transform method, because there is a one-to-one mapping between the cumulative probability distribution and the range  $[0,1]$ . It can be shown that by taking a uniform random number in this range and finding its corresponding coordinate in the cumulative probability distribution we obtain samples with the desired probability distribution.

type **gsl\_histogram\_pdf**

size_t n	This is the number of bins used to approximate the probability distribution function.
double * range	The ranges of the bins are stored in an array of $n + 1$ elements pointed to by <i>range</i> .
double * sum	The cumulative probability for the bins is stored in an array of $n$ elements pointed to by <i>sum</i> .

The following functions allow you to create a *gsl\_histogram\_pdf* struct which represents this probability distribution and generate random samples from it.

*gsl\_histogram\_pdf* \***gsl\_histogram\_pdf\_alloc**(size\_t n)

This function allocates memory for a probability distribution with  $n$  bins and returns a pointer to a newly initialized *gsl\_histogram\_pdf* struct. If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of *GSL\_ENOMEM*.

int **gsl\_histogram\_pdf\_init**(*gsl\_histogram\_pdf* \*p, const *gsl\_histogram* \*h)

This function initializes the probability distribution  $p$  with the contents of the histogram  $h$ . If any of the bins of  $h$  are negative then the error handler is invoked with an error code of *GSL\_EDOM* because a probability distribution cannot contain negative values.

void **gsl\_histogram\_pdf\_free**(*gsl\_histogram\_pdf* \*p)

This function frees the probability distribution function  $p$  and all of the memory associated with it.

double **gsl\_histogram\_pdf\_sample**(const *gsl\_histogram\_pdf* \*p, double r)

This function uses  $r$ , a uniform random number between zero and one, to compute a single random sample from the probability distribution  $p$ . The algorithm used to compute the sample  $s$  is given by the following formula,

$$s = \text{range}[i] + \delta * (\text{range}[i + 1] - \text{range}[i])$$

where  $i$  is the index which satisfies  $\text{sum}[i] \leq r < \text{sum}[i + 1]$  and  $\delta$  is  $(r - \text{sum}[i]) / (\text{sum}[i + 1] - \text{sum}[i])$ .

## 25.11 Example programs for histograms

The following program shows how to make a simple histogram of a column of numerical data supplied on `stdin`. The program takes three arguments, specifying the upper and lower bounds of the histogram and the number of bins. It then reads numbers from `stdin`, one line at a time, and adds them to the histogram. When there is no more data to read it prints out the accumulated histogram using `gsl_histogram_fprintf()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <gsl/gsl_histogram.h>

int
main (int argc, char **argv)
{
    double a, b;
    size_t n;

    if (argc != 4)
    {
        printf ("Usage: gsl-histogram xmin xmax n\n"
                "Computes a histogram of the data "
                "on stdin using n bins from xmin "
                "to xmax\n");
        exit (0);
    }

    a = atof (argv[1]);
    b = atof (argv[2]);
    n = atoi (argv[3]);

    {
        double x;
        gsl_histogram * h = gsl_histogram_alloc (n);
        gsl_histogram_set_ranges_uniform (h, a, b);

        while (fscanf (stdin, "%lg", &x) == 1)
        {
            gsl_histogram_increment (h, x);
        }
        gsl_histogram_fprintf (stdout, h, "%g", "%g");
        gsl_histogram_free (h);
    }
    exit (0);
}
```

Here is an example of the program in use. We generate 10000 random samples from a Cauchy distribution with a width of 30 and histogram them over the range -100 to 100, using 200 bins:

```
$ gsl-randist 0 10000 cauchy 30
| gsl-histogram -- -100 100 200 > histogram.dat
```

Fig. 25.1 shows the familiar shape of the Cauchy distribution and the fluctuations caused by the finite sample size.

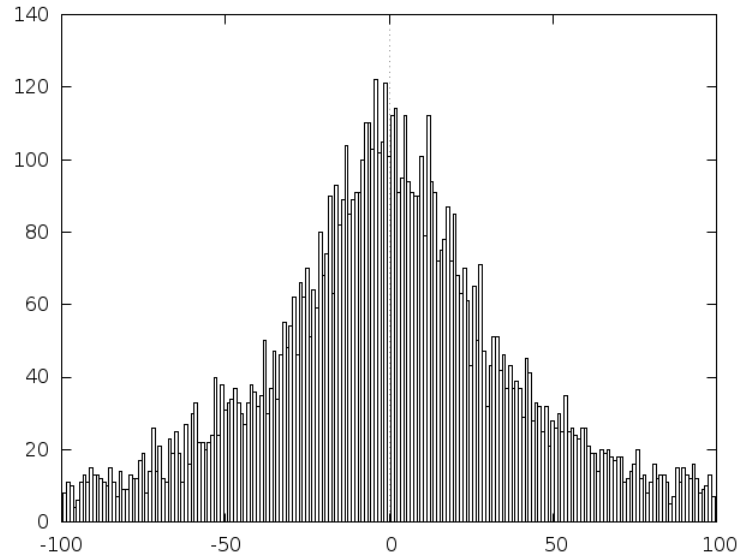


Fig. 25.1: Histogram output from example program

## 25.12 Two dimensional histograms

A two dimensional histogram consists of a set of *bins* which count the number of events falling in a given area of the  $(x, y)$  plane. The simplest way to use a two dimensional histogram is to record two-dimensional position information,  $n(x, y)$ . Another possibility is to form a *joint distribution* by recording related variables. For example a detector might record both the position of an event  $(x)$  and the amount of energy it deposited  $E$ . These could be histogrammed as the joint distribution  $n(x, E)$ .

## 25.13 The 2D histogram struct

Two dimensional histograms are defined by the following struct,

type `gsl_histogram2d`

size_t nx, ny	This is the number of histogram bins in the x and y directions.
double * xrange	The ranges of the bins in the x-direction are stored in an array of <code>nx + 1</code> elements pointed to by <code>xrange</code> .
double * yrange	The ranges of the bins in the y-direction are stored in an array of <code>ny + 1</code> elements pointed to by <code>yrange</code> .
double * bin	The counts for each bin are stored in an array pointed to by <code>bin</code> . The bins are floating-point numbers, so you can increment them by non-integer values if necessary. The array <code>bin</code> stores the two dimensional array of bins in a single block of memory according to the mapping <code>bin(i, j) = bin[i * ny + j]</code> .

The range for  $\text{bin}(i, j)$  is given by  $\text{xrange}[i]$  to  $\text{xrange}[i+1]$  in the x-direction and  $\text{yrange}[j]$  to  $\text{yrange}[j+1]$  in the y-direction. Each bin is inclusive at the lower end and exclusive at the upper end. Mathematically this means that the bins are defined by the following inequality,

$$\begin{aligned} \text{bin}(i,j) \text{ corresponds to } & \text{xrange}[i] \leq x < \text{xrange}[i + 1] \\ \text{and} & \text{yrange}[j] \leq y < \text{yrange}[j + 1] \end{aligned}$$

Note that any samples which fall on the upper sides of the histogram are excluded. If you want to include these values for the side bins you will need to add an extra row or column to your histogram.

The `gsl_histogram2d` struct and its associated functions are defined in the header file `gsl_histogram2d.h`.

## 25.14 2D Histogram allocation

The functions for allocating memory to a 2D histogram follow the style of `malloc()` and `free()`. In addition they also perform their own error checking. If there is insufficient memory available to allocate a histogram then the functions call the error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every 2D histogram alloc.

`gsl_histogram2d *gsl_histogram2d_alloc(size_t nx, size_t ny)`

This function allocates memory for a two-dimensional histogram with *nx* bins in the x direction and *ny* bins in the y direction. The function returns a pointer to a newly created `gsl_histogram2d` struct. If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`. The bins and ranges must be initialized with one of the functions below before the histogram is ready for use.

`int gsl_histogram2d_set_ranges(gsl_histogram2d *h, const double xrange[], size_t xsize, const double yrange[], size_t ysize)`

This function sets the ranges of the existing histogram *h* using the arrays *xrange* and *yrange* of size *xsize* and *ysize* respectively. The values of the histogram bins are reset to zero.

`int gsl_histogram2d_set_ranges_uniform(gsl_histogram2d *h, double xmin, double xmax, double ymin, double ymax)`

This function sets the ranges of the existing histogram *h* to cover the ranges *xmin* to *xmax* and *ymin* to *ymax* uniformly. The values of the histogram bins are reset to zero.

`void gsl_histogram2d_free(gsl_histogram2d *h)`

This function frees the 2D histogram *h* and all of the memory associated with it.

## 25.15 Copying 2D Histograms

`int gsl_histogram2d_memcpy(gsl_histogram2d *dest, const gsl_histogram2d *src)`

This function copies the histogram *src* into the pre-existing histogram *dest*, making *dest* into an exact copy of *src*. The two histograms must be of the same size.

`gsl_histogram2d *gsl_histogram2d_clone(const gsl_histogram2d *src)`

This function returns a pointer to a newly created histogram which is an exact copy of the histogram *src*.

## 25.16 Updating and accessing 2D histogram elements

You can access the bins of a two-dimensional histogram either by specifying a pair of  $(x, y)$  coordinates or by using the bin indices  $(i, j)$  directly. The functions for accessing the histogram through  $(x, y)$  coordinates use binary searches in the  $x$  and  $y$  directions to identify the bin which covers the appropriate range.

int **gsl\_histogram2d\_increment**(*gsl\_histogram2d* \*h, double x, double y)

This function updates the histogram  $h$  by adding one (1.0) to the bin whose  $x$  and  $y$  ranges contain the coordinates  $(x, y)$ .

If the point  $(x, y)$  lies inside the valid ranges of the histogram then the function returns zero to indicate success. If  $(x, y)$  lies outside the limits of the histogram then the function returns *GSL\_EDOM*, and none of the bins are modified. The error handler is not called, since it is often necessary to compute histograms for a small range of a larger dataset, ignoring any coordinates outside the range of interest.

int **gsl\_histogram2d\_accumulate**(*gsl\_histogram2d* \*h, double x, double y, double weight)

This function is similar to *gsl\_histogram2d\_increment()* but increases the value of the appropriate bin in the histogram  $h$  by the floating-point number *weight*.

double **gsl\_histogram2d\_get**(const *gsl\_histogram2d* \*h, size\_t i, size\_t j)

This function returns the contents of the  $(i, j)$ -th bin of the histogram  $h$ . If  $(i, j)$  lies outside the valid range of indices for the histogram then the error handler is called with an error code of *GSL\_EDOM* and the function returns 0.

int **gsl\_histogram2d\_get\_xrange**(const *gsl\_histogram2d* \*h, size\_t i, double \*xlower, double \*xupper)

int **gsl\_histogram2d\_get\_yrange**(const *gsl\_histogram2d* \*h, size\_t j, double \*ylower, double \*yupper)

These functions find the upper and lower range limits of the  $i$ -th and  $j$ -th bins in the  $x$  and  $y$  directions of the histogram  $h$ . The range limits are stored in *xlower* and *xupper* or *ylower* and *yupper*. The lower limits are inclusive (i.e. events with these coordinates are included in the bin) and the upper limits are exclusive (i.e. events with the value of the upper limit are not included and fall in the neighboring higher bin, if it exists). The functions return 0 to indicate success. If  $i$  or  $j$  lies outside the valid range of indices for the histogram then the error handler is called with an error code of *GSL\_EDOM*.

double **gsl\_histogram2d\_xmax**(const *gsl\_histogram2d* \*h)

double **gsl\_histogram2d\_xmin**(const *gsl\_histogram2d* \*h)

size\_t **gsl\_histogram2d\_nx**(const *gsl\_histogram2d* \*h)

double **gsl\_histogram2d\_ymax**(const *gsl\_histogram2d* \*h)

double **gsl\_histogram2d\_ymin**(const *gsl\_histogram2d* \*h)

size\_t **gsl\_histogram2d\_ny**(const *gsl\_histogram2d* \*h)

These functions return the maximum upper and minimum lower range limits and the number of bins for the  $x$  and  $y$  directions of the histogram  $h$ . They provide a way of determining these values without accessing the *gsl\_histogram2d* struct directly.

void **gsl\_histogram2d\_reset**(*gsl\_histogram2d* \*h)

This function resets all the bins of the histogram  $h$  to zero.

## 25.17 Searching 2D histogram ranges

The following functions are used by the access and update routines to locate the bin which corresponds to a given  $(x, y)$  coordinate.

int **gsl\_histogram2d\_find**(const *gsl\_histogram2d* \*h, double x, double y, size\_t \*i, size\_t \*j)

This function finds and sets the indices  $i$  and  $j$  to the bin which covers the coordinates  $(x, y)$ . The bin is located using a binary search. The search includes an optimization for histograms with uniform ranges, and will return the correct bin immediately in this case. If  $(x, y)$  is found then the function sets the indices  $(i, j)$  and returns `GSL_SUCCESS`. If  $(x, y)$  lies outside the valid range of the histogram then the function returns `GSL_EDOM` and the error handler is invoked.

## 25.18 2D Histogram Statistics

double **gsl\_histogram2d\_max\_val**(const *gsl\_histogram2d* \*h)

This function returns the maximum value contained in the histogram bins.

void **gsl\_histogram2d\_max\_bin**(const *gsl\_histogram2d* \*h, size\_t \*i, size\_t \*j)

This function finds the indices of the bin containing the maximum value in the histogram  $h$  and stores the result in  $(i, j)$ . In the case where several bins contain the same maximum value the first bin found is returned.

double **gsl\_histogram2d\_min\_val**(const *gsl\_histogram2d* \*h)

This function returns the minimum value contained in the histogram bins.

void **gsl\_histogram2d\_min\_bin**(const *gsl\_histogram2d* \*h, size\_t \*i, size\_t \*j)

This function finds the indices of the bin containing the minimum value in the histogram  $h$  and stores the result in  $(i, j)$ . In the case where several bins contain the same maximum value the first bin found is returned.

double **gsl\_histogram2d\_xmean**(const *gsl\_histogram2d* \*h)

This function returns the mean of the histogrammed x variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl\_histogram2d\_ymean**(const *gsl\_histogram2d* \*h)

This function returns the mean of the histogrammed y variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl\_histogram2d\_xsigma**(const *gsl\_histogram2d* \*h)

This function returns the standard deviation of the histogrammed x variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl\_histogram2d\_ysigma**(const *gsl\_histogram2d* \*h)

This function returns the standard deviation of the histogrammed y variable, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl\_histogram2d\_cov**(const *gsl\_histogram2d* \*h)

This function returns the covariance of the histogrammed x and y variables, where the histogram is regarded as a probability distribution. Negative bin values are ignored for the purposes of this calculation.

double **gsl\_histogram2d\_sum**(const *gsl\_histogram2d* \*h)

This function returns the sum of all bin values. Negative bin values are included in the sum.



## 25.19 2D Histogram Operations

int `gsl_histogram2d_equal_bins_p`(const *gsl\_histogram2d* \*h1, const *gsl\_histogram2d* \*h2)

This function returns 1 if all the individual bin ranges of the two histograms are identical, and 0 otherwise.

int `gsl_histogram2d_add`(*gsl\_histogram2d* \*h1, const *gsl\_histogram2d* \*h2)

This function adds the contents of the bins in histogram *h2* to the corresponding bins of histogram *h1*, i.e.  $h'_1(i, j) = h_1(i, j) + h_2(i, j)$ . The two histograms must have identical bin ranges.

int `gsl_histogram2d_sub`(*gsl\_histogram2d* \*h1, const *gsl\_histogram2d* \*h2)

This function subtracts the contents of the bins in histogram *h2* from the corresponding bins of histogram *h1*, i.e.  $h'_1(i, j) = h_1(i, j) - h_2(i, j)$ . The two histograms must have identical bin ranges.

int `gsl_histogram2d_mul`(*gsl\_histogram2d* \*h1, const *gsl\_histogram2d* \*h2)

This function multiplies the contents of the bins of histogram *h1* by the contents of the corresponding bins in histogram *h2*, i.e.  $h'_1(i, j) = h_1(i, j) * h_2(i, j)$ . The two histograms must have identical bin ranges.

int `gsl_histogram2d_div`(*gsl\_histogram2d* \*h1, const *gsl\_histogram2d* \*h2)

This function divides the contents of the bins of histogram *h1* by the contents of the corresponding bins in histogram *h2*, i.e.  $h'_1(i, j) = h_1(i, j) / h_2(i, j)$ . The two histograms must have identical bin ranges.

int `gsl_histogram2d_scale`(*gsl\_histogram2d* \*h, double scale)

This function multiplies the contents of the bins of histogram *h* by the constant *scale*, i.e.

$$h'_1(i, j) = h_1(i, j) * scale$$

int `gsl_histogram2d_shift`(*gsl\_histogram2d* \*h, double offset)

This function shifts the contents of the bins of histogram *h* by the constant *offset*, i.e.

$$h'_1(i, j) = h_1(i, j) + offset$$

## 25.20 Reading and writing 2D histograms

The library provides functions for reading and writing two dimensional histograms to a file as binary data or formatted text.

int `gsl_histogram2d_fwrite`(FILE \*stream, const *gsl\_histogram2d* \*h)

This function writes the ranges and bins of the histogram *h* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int `gsl_histogram2d_fread`(FILE \*stream, *gsl\_histogram2d* \*h)

This function reads into the histogram *h* from the stream *stream* in binary format. The histogram *h* must be preallocated with the correct size since the function uses the number of x and y bins in *h* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int `gsl_histogram2d_fprintf`(FILE \*stream, const *gsl\_histogram2d* \*h, const char \*range\_format, const char \*bin\_format)

This function writes the ranges and bins of the histogram *h* line-by-line to the stream *stream* using the format specifiers *range\_format* and *bin\_format*. These should be one of the `%g`, `%e` or `%f` formats for floating point numbers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file. The histogram output is formatted in five columns, and the columns are separated by spaces, like this:



```

xrange[0] xrange[1] yrange[0] yrange[1] bin(0,0)
xrange[0] xrange[1] yrange[1] yrange[2] bin(0,1)
xrange[0] xrange[1] yrange[2] yrange[3] bin(0,2)
....
xrange[0] xrange[1] yrange[ny-1] yrange[ny] bin(0,ny-1)

xrange[1] xrange[2] yrange[0] yrange[1] bin(1,0)
xrange[1] xrange[2] yrange[1] yrange[2] bin(1,1)
xrange[1] xrange[2] yrange[1] yrange[2] bin(1,2)
....
xrange[1] xrange[2] yrange[ny-1] yrange[ny] bin(1,ny-1)

....

xrange[nx-1] xrange[nx] yrange[0] yrange[1] bin(nx-1,0)
xrange[nx-1] xrange[nx] yrange[1] yrange[2] bin(nx-1,1)
xrange[nx-1] xrange[nx] yrange[1] yrange[2] bin(nx-1,2)
....
xrange[nx-1] xrange[nx] yrange[ny-1] yrange[ny] bin(nx-1,ny-1)

```

Each line contains the lower and upper limits of the bin and the contents of the bin. Since the upper limits of the each bin are the lower limits of the neighboring bins there is duplication of these values but this allows the histogram to be manipulated with line-oriented tools.

int **gsl\_histogram2d\_fscanf**(FILE \*stream, *gsl\_histogram2d* \*h)

This function reads formatted data from the stream *stream* into the histogram *h*. The data is assumed to be in the five-column format used by *gsl\_histogram2d\_fprintf()*. The histogram *h* must be preallocated with the correct lengths since the function uses the sizes of *h* to determine how many numbers to read. The function returns 0 for success and GSL\_EFAILED if there was a problem reading from the file.

## 25.21 Resampling from 2D histograms

As in the one-dimensional case, a two-dimensional histogram made by counting events can be regarded as a measurement of a probability distribution. Allowing for statistical error, the height of each bin represents the probability of an event where  $(x, y)$  falls in the range of that bin. For a two-dimensional histogram the probability distribution takes the form  $p(x, y)dx dy$  where,

$$p(x, y) = n_{ij} / (N A_{ij})$$

In this equation  $n_{ij}$  is the number of events in the bin which contains  $(x, y)$ ,  $A_{ij}$  is the area of the bin and  $N$  is the total number of events. The distribution of events within each bin is assumed to be uniform.

type **gsl\_histogram2d\_pdf**

size_t nx, ny	This is the number of histogram bins used to approximate the probability distribution function in the x and y directions.
double * xrange	The ranges of the bins in the x-direction are stored in an array of nx + 1 elements pointed to by xrange.
double * yrange	The ranges of the bins in the y-direction are stored in an array of ny + 1 pointed to by yrange.
double * sum	The cumulative probability for the bins is stored in an array of nx * ny elements pointed to by sum.

The following functions allow you to create a *gsl\_histogram2d\_pdf* struct which represents a two dimensional probability distribution and generate random samples from it.

*gsl\_histogram2d\_pdf* \***gsl\_histogram2d\_pdf\_alloc**(size\_t nx, size\_t ny)

This function allocates memory for a two-dimensional probability distribution of size *nx*-by-*ny* and returns a pointer to a newly initialized *gsl\_histogram2d\_pdf* struct. If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of *GSL\_ENOMEM*.

int **gsl\_histogram2d\_pdf\_init**(*gsl\_histogram2d\_pdf* \*p, const *gsl\_histogram2d* \*h)

This function initializes the two-dimensional probability distribution calculated *p* from the histogram *h*. If any of the bins of *h* are negative then the error handler is invoked with an error code of *GSL\_EDOM* because a probability distribution cannot contain negative values.

void **gsl\_histogram2d\_pdf\_free**(*gsl\_histogram2d\_pdf* \*p)

This function frees the two-dimensional probability distribution function *p* and all of the memory associated with it.

int **gsl\_histogram2d\_pdf\_sample**(const *gsl\_histogram2d\_pdf* \*p, double r1, double r2, double \*x, double \*y)

This function uses two uniform random numbers between zero and one, *r1* and *r2*, to compute a single random sample from the two-dimensional probability distribution *p*.

## 25.22 Example programs for 2D histograms

This program demonstrates two features of two-dimensional histograms. First a 10-by-10 two-dimensional histogram is created with *x* and *y* running from 0 to 1. Then a few sample points are added to the histogram, at (0.3,0.3) with a height of 1, at (0.8,0.1) with a height of 5 and at (0.7,0.9) with a height of 0.5. This histogram with three events is used to generate a random sample of 1000 simulated events, which are printed out.

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_histogram2d.h>

int
main (void)
{
  const gsl_rng_type * T;
  gsl_rng * r;

  gsl_histogram2d * h = gsl_histogram2d_alloc (10, 10);

  gsl_histogram2d_set_ranges_uniform (h,
                                     0.0, 1.0,
                                     0.0, 1.0);

  gsl_histogram2d_accumulate (h, 0.3, 0.3, 1);
  gsl_histogram2d_accumulate (h, 0.8, 0.1, 5);
  gsl_histogram2d_accumulate (h, 0.7, 0.9, 0.5);

  gsl_rng_env_setup ();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  {
```

(continues on next page)

(continued from previous page)

```

int i;
gsl_histogram2d_pdf * p
  = gsl_histogram2d_pdf_alloc (h->nx, h->ny);

gsl_histogram2d_pdf_init (p, h);

for (i = 0; i < 1000; i++) {
  double x, y;
  double u = gsl_rng_uniform (r);
  double v = gsl_rng_uniform (r);

  gsl_histogram2d_pdf_sample (p, u, v, &x, &y);

  printf ("%g %g\n", x, y);
}

gsl_histogram2d_pdf_free (p);
}

gsl_histogram2d_free (h);
gsl_rng_free (r);

return 0;
}

```

The following plot shows the distribution of the simulated events. Using a higher resolution grid we can see the original underlying histogram and also the statistical fluctuations caused by the events being uniformly distributed over the area of the original bins.

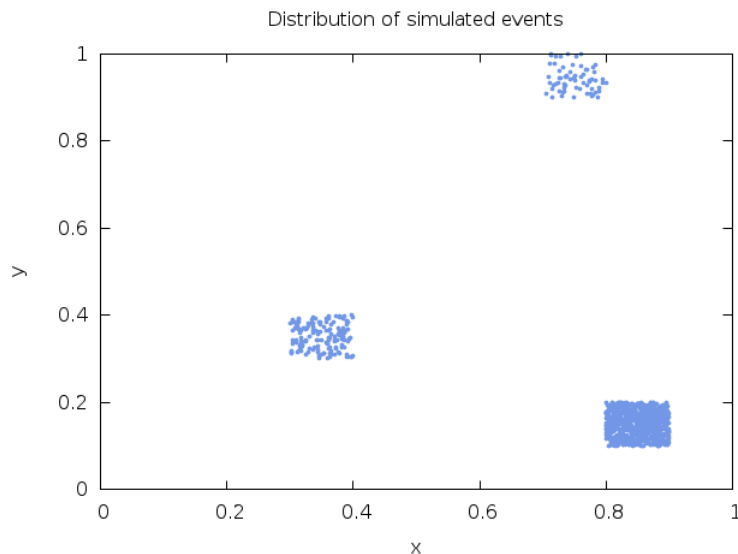


Fig. 25.2: Distribution of simulated events from example program



## N-TUPLES

This chapter describes functions for creating and manipulating *ntuples*, sets of values associated with events. The ntuple data are stored in files. Their values can be extracted in any combination and *booked* in a histogram using a selection function.

The values to be stored are held in a user-defined data structure, and an ntuple is created associating this data structure with a file. The values are then written to the file (normally inside a loop) using the ntuple functions described below.

A histogram can be created from ntuple data by providing a selection function and a value function. The selection function specifies whether an event should be included in the subset to be analyzed or not. The value function computes the entry to be added to the histogram for each event.

All the ntuple functions are defined in the header file `gsl_ntuple.h`.

### 26.1 The ntuple struct

type `gsl_ntuple`

Ntuples are manipulated using the `gsl_ntuple` struct. This struct contains information on the file where the ntuple data is stored, a pointer to the current ntuple data row and the size of the user-defined ntuple data struct:

```
typedef struct
{
    FILE * file;
    void * ntuple_data;
    size_t size;
} gsl_ntuple;
```

### 26.2 Creating ntuples

`gsl_ntuple *gsl_ntuple_create(char *filename, void *ntuple_data, size_t size)`

This function creates a new write-only ntuple file `filename` for ntuples of size `size` and returns a pointer to the newly created ntuple struct. Any existing file with the same name is truncated to zero length and overwritten. A pointer to memory for the current ntuple row `ntuple_data` must be supplied—this is used to copy ntuples in and out of the file.

## 26.3 Opening an existing ntuple file

*gsl\_ntuple* \***gsl\_ntuple\_open**(char \*filename, void \*ntuple\_data, size\_t size)

This function opens an existing ntuple file *filename* for reading and returns a pointer to a corresponding ntuple struct. The ntuple in the file must have size *size*. A pointer to memory for the current ntuple row *ntuple\_data* must be supplied—this is used to copy ntuple in and out of the file.

## 26.4 Writing ntuple

int **gsl\_ntuple\_write**(*gsl\_ntuple* \*ntuple)

This function writes the current ntuple *ntuple->ntuple\_data* of size *ntuple->size* to the corresponding file.

int **gsl\_ntuple\_bookdata**(*gsl\_ntuple* \*ntuple)

This function is a synonym for *gsl\_ntuple\_write()*.

## 26.5 Reading ntuple

int **gsl\_ntuple\_read**(*gsl\_ntuple* \*ntuple)

This function reads the current row of the ntuple file for *ntuple* and stores the values in *ntuple->data*.

## 26.6 Closing an ntuple file

int **gsl\_ntuple\_close**(*gsl\_ntuple* \*ntuple)

This function closes the ntuple file *ntuple* and frees its associated allocated memory.

## 26.7 Histogramming ntuple values

Once an ntuple has been created its contents can be histogrammed in various ways using the function *gsl\_ntuple\_project()*. Two user-defined functions must be provided, a function to select events and a function to compute scalar values. The selection function and the value function both accept the ntuple row as a first argument and other parameters as a second argument.

type **gsl\_ntuple\_select\_fn**

The *selection function* determines which ntuple rows are selected for histogramming. It is defined by the following struct:

```
typedef struct
{
    int (* function) (void * ntuple_data, void * params);
    void * params;
} gsl_ntuple_select_fn;
```

The struct component *function* should return a non-zero value for each ntuple row that is to be included in the histogram.

type **gsl\_ntuple\_value\_fn**

The *value function* computes scalar values for those ntuple rows selected by the selection function:

```
typedef struct
{
    double (* function) (void * ntuple_data, void * params);
    void * params;
} gsl_ntuple_value_fn;
```

In this case the struct component `function` should return the value to be added to the histogram for the ntuple row.

```
int gsl_ntuple_project(gsl_histogram *h, gsl_ntuple *ntuple, gsl_ntuple_value_fn *value_func,
                      gsl_ntuple_select_fn *select_func)
```

This function updates the histogram `h` from the ntuple `ntuple` using the functions `value_func` and `select_func`. For each ntuple row where the selection function `select_func` is non-zero the corresponding value of that row is computed using the function `value_func` and added to the histogram. Those ntuple rows where `select_func` returns zero are ignored. New entries are added to the histogram, so subsequent calls can be used to accumulate further data in the same histogram.

## 26.8 Examples

The following example programs demonstrate the use of ntuples in managing a large dataset. The first program creates a set of 10,000 simulated “events”, each with 3 associated values  $(x, y, z)$ . These are generated from a Gaussian distribution with unit variance, for demonstration purposes, and written to the ntuple file `test.dat`.

```
#include <gsl/gsl_ntuple.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

struct data
{
    double x;
    double y;
    double z;
};

int
main (void)
{
    const gsl_rng_type * T;
    gsl_rng * r;

    struct data ntuple_row;
    int i;

    gsl_ntuple *ntuple
        = gsl_ntuple_create ("test.dat", &ntuple_row,
                             sizeof (ntuple_row));

    gsl_rng_env_setup ();

    T = gsl_rng_default;
    r = gsl_rng_alloc (T);
```

(continues on next page)

(continued from previous page)

```

for (i = 0; i < 10000; i++)
{
    ntuple_row.x = gsl_ran_ugaussian (r);
    ntuple_row.y = gsl_ran_ugaussian (r);
    ntuple_row.z = gsl_ran_ugaussian (r);

    gsl_ntuple_write (ntuple);
}

gsl_ntuple_close (ntuple);
gsl_rng_free (r);

return 0;
}

```

The next program analyses the ntuple data in the file `test.dat`. The analysis procedure is to compute the squared-magnitude of each event,  $E^2 = x^2 + y^2 + z^2$ , and select only those which exceed a lower limit of 1.5. The selected events are then histogrammed using their  $E^2$  values.

```

#include <math.h>
#include <gsl/gsl_ntuple.h>
#include <gsl/gsl_histogram.h>

struct data
{
    double x;
    double y;
    double z;
};

int sel_func (void *ntuple_data, void *params);
double val_func (void *ntuple_data, void *params);

int
main (void)
{
    struct data ntuple_row;

    gsl_ntuple *ntuple
        = gsl_ntuple_open ("test.dat", &ntuple_row,
                           sizeof (ntuple_row));
    double lower = 1.5;

    gsl_ntuple_select_fn S;
    gsl_ntuple_value_fn V;

    gsl_histogram *h = gsl_histogram_alloc (100);
    gsl_histogram_set_ranges_uniform(h, 0.0, 10.0);

    S.function = &sel_func;
    S.params = &lower;
}

```

(continues on next page)



(continued from previous page)

```
V.function = &val_func;
V.params = 0;

gsl_ntuple_project (h, ntuple, &V, &S);
gsl_histogram_fprintf (stdout, h, "%f", "%f");
gsl_histogram_free (h);
gsl_ntuple_close (ntuple);

return 0;
}

int
sel_func (void *ntuple_data, void *params)
{
    struct data * data = (struct data *) ntuple_data;
    double x, y, z, E2, scale;
    scale = *(double *) params;

    x = data->x;
    y = data->y;
    z = data->z;

    E2 = x * x + y * y + z * z;

    return E2 > scale;
}

double
val_func (void *ntuple_data, void *params)
{
    (void)(params); /* avoid unused parameter warning */
    struct data * data = (struct data *) ntuple_data;
    double x, y, z;

    x = data->x;
    y = data->y;
    z = data->z;

    return x * x + y * y + z * z;
}
```

Fig. 26.1 shows the distribution of the selected events. Note the cut-off at the lower bound.

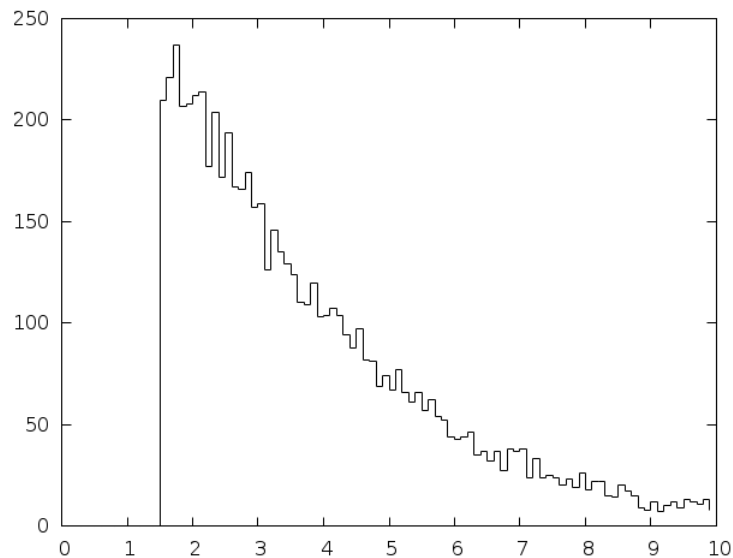


Fig. 26.1: Distribution of selected events

## 26.9 References and Further Reading

Further information on the use of ntuples can be found in the documentation for the CERN packages PAW and HBOOK (available online).

## MONTE CARLO INTEGRATION

This chapter describes routines for multidimensional Monte Carlo integration. These include the traditional Monte Carlo method and adaptive algorithms such as VEGAS and MISER which use importance sampling and stratified sampling techniques. Each algorithm computes an estimate of a multidimensional definite integral of the form,

$$I = \int_{x_l}^{x_u} dx \int_{y_l}^{y_u} dy \dots f(x, y, \dots)$$

over a hypercubic region  $((x_l, x_u), (y_l, y_u), \dots)$  using a fixed number of function calls. The routines also provide a statistical estimate of the error on the result. This error estimate should be taken as a guide rather than as a strict error bound—random sampling of the region may not uncover all the important features of the function, resulting in an underestimate of the error.

The functions are defined in separate header files for each routine, `gsl_monte_plain.h`, `gsl_monte_miser.h` and `gsl_monte_vegas.h`.

### 27.1 Interface

All of the Monte Carlo integration routines use the same general form of interface. There is an allocator to allocate memory for control variables and workspace, a routine to initialize those control variables, the integrator itself, and a function to free the space when done.

Each integration function requires a random number generator to be supplied, and returns an estimate of the integral and its standard deviation. The accuracy of the result is determined by the number of function calls specified by the user. If a known level of accuracy is required this can be achieved by calling the integrator several times and averaging the individual results until the desired accuracy is obtained.

Random sample points used within the Monte Carlo routines are always chosen strictly within the integration region, so that endpoint singularities are automatically avoided.

The function to be integrated has its own datatype, defined in the header file `gsl_monte.h`.

#### type `gsl_monte_function`

This data type defines a general function with parameters for Monte Carlo integration.

<code>double (* f) (double * x, size_t dim, void * params)</code>	this function should return the value $f(x, params)$ for the argument <code>x</code> and parameters <code>params</code> , where <code>x</code> is an array of size <code>dim</code> giving the coordinates of the point where the function is to be evaluated.
<code>size_t dim</code>	the number of dimensions for <code>x</code> .
<code>void * params</code>	a pointer to the parameters of the function.

Here is an example for a quadratic function in two dimensions,

$$f(x, y) = ax^2 + bxy + cy^2$$

with  $a = 3, b = 2, c = 1$ . The following code defines a *gsl\_monte\_function*  $F$  which you could pass to an integrator:

```

struct my_f_params { double a; double b; double c; };

double
my_f (double x[], size_t dim, void * p) {
    struct my_f_params * fp = (struct my_f_params *)p;

    if (dim != 2)
    {
        fprintf (stderr, "error: dim != 2");
        abort ();
    }

    return fp->a * x[0] * x[0]
        + fp->b * x[0] * x[1]
        + fp->c * x[1] * x[1];
}

gsl_monte_function F;
struct my_f_params params = { 3.0, 2.0, 1.0 };

F.f = &my_f;
F.dim = 2;
F.params = &params;

```

The function  $f(x)$  can be evaluated using the following macro:

```

#define GSL_MONTE_FN_EVAL(F,x)
    (*(F->f))(x, (F->dim), (F->params))

```

## 27.2 PLAIN Monte Carlo

The plain Monte Carlo algorithm samples points randomly from the integration region to estimate the integral and its error. Using this algorithm the estimate of the integral  $E(f; N)$  for  $N$  randomly distributed points  $x_i$  is given by,

$$E(f; N) = V\langle f \rangle = \frac{V}{N} \sum_i^N f(x_i)$$

where  $V$  is the volume of the integration region. The error on this estimate  $\sigma(E; N)$  is calculated from the estimated variance of the mean,

$$\sigma^2(E; N) = \frac{V^2}{N^2} \sum_i^N (f(x_i) - \langle f \rangle)^2.$$

For large  $N$  this variance decreases asymptotically as  $\text{Var}(f)/N$ , where  $\text{Var}(f)$  is the true variance of the function over the integration region. The error estimate itself should decrease as  $\sigma(f)/\sqrt{N}$ . The familiar law of errors decreasing as  $1/\sqrt{N}$  applies—to reduce the error by a factor of 10 requires a 100-fold increase in the number of sample points.

The functions described in this section are declared in the header file `gsl_monte_plain.h`.

type **gsl\_monte\_plain\_state**

This is a workspace for plain Monte Carlo integration

`gsl_monte_plain_state *gsl_monte_plain_alloc(size_t dim)`

This function allocates and initializes a workspace for Monte Carlo integration in *dim* dimensions.

`int gsl_monte_plain_init(gsl_monte_plain_state *s)`

This function initializes a previously allocated integration state. This allows an existing workspace to be reused for different integrations.

`int gsl_monte_plain_integrate(gsl_monte_function *f, const double xl[], const double xu[], size_t dim, size_t calls, gsl_rng *r, gsl_monte_plain_state *s, double *result, double *abserr)`

This routine uses the plain Monte Carlo algorithm to integrate the function *f* over the *dim*-dimensional hyper-cubic region defined by the lower and upper limits in the arrays *xl* and *xu*, each of size *dim*. The integration uses a fixed number of function calls *calls*, and obtains random sampling points using the random number generator *r*. A previously allocated workspace *s* must be supplied. The result of the integration is returned in *result*, with an estimated absolute error *abserr*.

`void gsl_monte_plain_free(gsl_monte_plain_state *s)`

This function frees the memory associated with the integrator state *s*.

## 27.3 MISER

The MISER algorithm of Press and Farrar is based on recursive stratified sampling. This technique aims to reduce the overall integration error by concentrating integration points in the regions of highest variance.

The idea of stratified sampling begins with the observation that for two disjoint regions *a* and *b* with Monte Carlo estimates of the integral  $E_a(f)$  and  $E_b(f)$  and variances  $\sigma_a^2(f)$  and  $\sigma_b^2(f)$ , the variance  $\text{Var}(f)$  of the combined estimate  $E(f) = \frac{1}{2}(E_a(f) + E_b(f))$  is given by,

$$\text{Var}(f) = \frac{\sigma_a^2(f)}{4N_a} + \frac{\sigma_b^2(f)}{4N_b}.$$

It can be shown that this variance is minimized by distributing the points such that,

$$\frac{N_a}{N_a + N_b} = \frac{\sigma_a}{\sigma_a + \sigma_b}.$$

Hence the smallest error estimate is obtained by allocating sample points in proportion to the standard deviation of the function in each sub-region.

The MISER algorithm proceeds by bisecting the integration region along one coordinate axis to give two sub-regions at each step. The direction is chosen by examining all *d* possible bisections and selecting the one which will minimize the combined variance of the two sub-regions. The variance in the sub-regions is estimated by sampling with a fraction of the total number of points available to the current step. The same procedure is then repeated recursively for each of the two half-spaces from the best bisection. The remaining sample points are allocated to the sub-regions using the formula for  $N_a$  and  $N_b$ . This recursive allocation of integration points continues down to a user-specified depth where each sub-region is integrated using a plain Monte Carlo estimate. These individual values and their error estimates are then combined upwards to give an overall result and an estimate of its error.

The functions described in this section are declared in the header file `gsl_monte_miser.h`.

`type gsl_monte_miser_state`

This workspace is used for MISER Monte Carlo integration

`gsl_monte_miser_state *gsl_monte_miser_alloc(size_t dim)`

This function allocates and initializes a workspace for Monte Carlo integration in *dim* dimensions. The workspace is used to maintain the state of the integration.

`int gsl_monte_miser_init(gsl_monte_miser_state *s)`

This function initializes a previously allocated integration state. This allows an existing workspace to be reused for different integrations.

int **gsl\_monte\_miser\_integrate**(*gsl\_monte\_function* \*f, const double xl[], const double xu[], size\_t dim, size\_t calls, *gsl\_rng* \*r, *gsl\_monte\_miser\_state* \*s, double \*result, double \*abserr)

This routine uses the MISER Monte Carlo algorithm to integrate the function  $f$  over the  $dim$ -dimensional hypercubic region defined by the lower and upper limits in the arrays  $xl$  and  $xu$ , each of size  $dim$ . The integration uses a fixed number of function calls  $calls$ , and obtains random sampling points using the random number generator  $r$ . A previously allocated workspace  $s$  must be supplied. The result of the integration is returned in  $result$ , with an estimated absolute error  $abserr$ .

void **gsl\_monte\_miser\_free**(*gsl\_monte\_miser\_state* \*s)

This function frees the memory associated with the integrator state  $s$ .

The MISER algorithm has several configurable parameters which can be changed using the following two functions<sup>1</sup>.

void **gsl\_monte\_miser\_params\_get**(const *gsl\_monte\_miser\_state* \*s, *gsl\_monte\_miser\_params* \*params)

This function copies the parameters of the integrator state into the user-supplied  $params$  structure.

void **gsl\_monte\_miser\_params\_set**(*gsl\_monte\_miser\_state* \*s, const *gsl\_monte\_miser\_params* \*params)

This function sets the integrator parameters based on values provided in the  $params$  structure.

Typically the values of the parameters are first read using `gsl_monte_miser_params_get()`, the necessary changes are made to the fields of the  $params$  structure, and the values are copied back into the integrator state using `gsl_monte_miser_params_set()`. The functions use the *gsl\_monte\_miser\_params* structure which contains the following fields:

type **gsl\_monte\_miser\_params**

double **estimate\_frac**

This parameter specifies the fraction of the currently available number of function calls which are allocated to estimating the variance at each recursive step. The default value is 0.1.

size\_t **min\_calls**

This parameter specifies the minimum number of function calls required for each estimate of the variance. If the number of function calls allocated to the estimate using *estimate\_frac* falls below *min\_calls* then *min\_calls* are used instead. This ensures that each estimate maintains a reasonable level of accuracy. The default value of *min\_calls* is  $16 * dim$ .

size\_t **min\_calls\_per\_bisection**

This parameter specifies the minimum number of function calls required to proceed with a bisection step. When a recursive step has fewer calls available than *min\_calls\_per\_bisection* it performs a plain Monte Carlo estimate of the current sub-region and terminates its branch of the recursion. The default value of this parameter is  $32 * min\_calls$ .

double **alpha**

This parameter controls how the estimated variances for the two sub-regions of a bisection are combined when allocating points. With recursive sampling the overall variance should scale better than  $1/N$ , since the values from the sub-regions will be obtained using a procedure which explicitly minimizes their variance. To accommodate this behavior the MISER algorithm allows the total variance to depend on a scaling parameter  $\alpha$ ,

$$\text{Var}(f) = \frac{\sigma_a}{N_a^\alpha} + \frac{\sigma_b}{N_b^\alpha}.$$

The authors of the original paper describing MISER recommend the value  $\alpha = 2$  as a good choice, obtained from numerical experiments, and this is used as the default value in this implementation.

double **dither**

This parameter introduces a random fractional variation of size *dither* into each bisection, which can be used to break the symmetry of integrands which are concentrated near the exact center of the hypercubic

<sup>1</sup> The previous method of accessing these fields directly through the *gsl\_monte\_miser\_state* struct is now deprecated.

integration region. The default value of `dither` is zero, so no variation is introduced. If needed, a typical value of `dither` is 0.1.

## 27.4 VEGAS

The VEGAS algorithm of Lepage is based on importance sampling. It samples points from the probability distribution described by the function  $|f|$ , so that the points are concentrated in the regions that make the largest contribution to the integral.

In general, if the Monte Carlo integral of  $f$  is sampled with points distributed according to a probability distribution described by the function  $g$ , we obtain an estimate  $E_g(f; N)$ ,

$$E_g(f; N) = E(f/g; N)$$

with a corresponding variance,

$$\text{Var}_g(f; N) = \text{Var}(f/g; N)$$

If the probability distribution is chosen as  $g = |f|/I(|f|)$  then it can be shown that the variance  $V_g(f; N)$  vanishes, and the error in the estimate will be zero. In practice it is not possible to sample from the exact distribution  $g$  for an arbitrary function, so importance sampling algorithms aim to produce efficient approximations to the desired distribution.

The VEGAS algorithm approximates the exact distribution by making a number of passes over the integration region while histogramming the function  $f$ . Each histogram is used to define a sampling distribution for the next pass. Asymptotically this procedure converges to the desired distribution. In order to avoid the number of histogram bins growing like  $K^d$  the probability distribution is approximated by a separable function:  $g(x_1, x_2, \dots) = g_1(x_1)g_2(x_2) \dots$  so that the number of bins required is only  $Kd$ . This is equivalent to locating the peaks of the function from the projections of the integrand onto the coordinate axes. The efficiency of VEGAS depends on the validity of this assumption. It is most efficient when the peaks of the integrand are well-localized. If an integrand can be rewritten in a form which is approximately separable this will increase the efficiency of integration with VEGAS.

VEGAS incorporates a number of additional features, and combines both stratified sampling and importance sampling. The integration region is divided into a number of “boxes”, with each box getting a fixed number of points (the goal is 2). Each box can then have a fractional number of bins, but if the ratio of bins-per-box is less than two, Vegas switches to a kind variance reduction (rather than importance sampling).

type `gsl_monte_vegas_state`

This workspace is used for VEGAS Monte Carlo integration

`gsl_monte_vegas_state *gsl_monte_vegas_alloc(size_t dim)`

This function allocates and initializes a workspace for Monte Carlo integration in `dim` dimensions. The workspace is used to maintain the state of the integration.

int `gsl_monte_vegas_init(gsl_monte_vegas_state *s)`

This function initializes a previously allocated integration state. This allows an existing workspace to be reused for different integrations.

int `gsl_monte_vegas_integrate(gsl_monte_function *f, double xl[], double xu[], size_t dim, size_t calls, gsl_rng *r, gsl_monte_vegas_state *s, double *result, double *abserr)`

This routine uses the VEGAS Monte Carlo algorithm to integrate the function  $f$  over the `dim`-dimensional hypercubic region defined by the lower and upper limits in the arrays `xl` and `xu`, each of size `dim`. The integration uses a fixed number of function calls `calls`, and obtains random sampling points using the random number generator `r`. A previously allocated workspace `s` must be supplied. The result of the integration is returned in `result`, with an estimated absolute error `abserr`. The result and its error estimate are based on a weighted average of independent samples. The chi-squared per degree of freedom for the weighted average is returned via the state struct component, `s->chi_sq`, and must be consistent with 1 for the weighted average to be reliable.

void **gsl\_monte\_vegas\_free**(*gsl\_monte\_vegas\_state* \*s)

This function frees the memory associated with the integrator state *s*.

The VEGAS algorithm computes a number of independent estimates of the integral internally, according to the `iterations` parameter described below, and returns their weighted average. Random sampling of the integrand can occasionally produce an estimate where the error is zero, particularly if the function is constant in some regions. An estimate with zero error causes the weighted average to break down and must be handled separately. In the original Fortran implementations of VEGAS the error estimate is made non-zero by substituting a small value (typically  $1e-30$ ). The implementation in GSL differs from this and avoids the use of an arbitrary constant—it either assigns the value a weight which is the average weight of the preceding estimates or discards it according to the following procedure,

- current estimate has zero error, weighted average has finite error

The current estimate is assigned a weight which is the average weight of the preceding estimates.

- current estimate has finite error, previous estimates had zero error

The previous estimates are discarded and the weighted averaging procedure begins with the current estimate.

- current estimate has zero error, previous estimates had zero error

The estimates are averaged using the arithmetic mean, but no error is computed.

The convergence of the algorithm can be tested using the overall chi-squared value of the results, which is available from the following function:

double **gsl\_monte\_vegas\_chisq**(const *gsl\_monte\_vegas\_state* \*s)

This function returns the chi-squared per degree of freedom for the weighted estimate of the integral. The returned value should be close to 1. A value which differs significantly from 1 indicates that the values from different iterations are inconsistent. In this case the weighted error will be under-estimated, and further iterations of the algorithm are needed to obtain reliable results.

void **gsl\_monte\_vegas\_runval**(const *gsl\_monte\_vegas\_state* \*s, double \*result, double \*sigma)

This function returns the raw (unaveraged) values of the integral *result* and its error *sigma* from the most recent iteration of the algorithm.

The VEGAS algorithm is highly configurable. Several parameters can be changed using the following two functions.

void **gsl\_monte\_vegas\_params\_get**(const *gsl\_monte\_vegas\_state* \*s, *gsl\_monte\_vegas\_params* \*params)

This function copies the parameters of the integrator state into the user-supplied *params* structure.

void **gsl\_monte\_vegas\_params\_set**(*gsl\_monte\_vegas\_state* \*s, const *gsl\_monte\_vegas\_params* \*params)

This function sets the integrator parameters based on values provided in the *params* structure.

Typically the values of the parameters are first read using `gsl_monte_vegas_params_get()`, the necessary changes are made to the fields of the *params* structure, and the values are copied back into the integrator state using `gsl_monte_vegas_params_set()`. The functions use the *gsl\_monte\_vegas\_params* structure which contains the following fields:

type **gsl\_monte\_vegas\_params**

double **alpha**

The parameter *alpha* controls the stiffness of the rebinning algorithm. It is typically set between one and two. A value of zero prevents rebinning of the grid. The default value is 1.5.

size\_t **iterations**

The number of iterations to perform for each call to the routine. The default value is 5 iterations.

int **stage**

Setting this determines the *stage* of the calculation. Normally, `stage = 0` which begins with a new uniform grid and empty weighted average. Calling VEGAS with `stage = 1` retains the grid from the previous run but discards the weighted average, so that one can “tune” the grid using a relatively small number of points



and then do a large run with `stage = 1` on the optimized grid. Setting `stage = 2` keeps the grid and the weighted average from the previous run, but may increase (or decrease) the number of histogram bins in the grid depending on the number of calls available. Choosing `stage = 3` enters at the main loop, so that nothing is changed, and is equivalent to performing additional iterations in a previous call.

`int mode`

The possible choices are `GSL_VEGAS_MODE_IMPORTANCE`, `GSL_VEGAS_MODE_STRATIFIED`, `GSL_VEGAS_MODE_IMPORTANCE_ONLY`. This determines whether VEGAS will use importance sampling or stratified sampling, or whether it can pick on its own. In low dimensions VEGAS uses strict stratified sampling (more precisely, stratified sampling is chosen if there are fewer than 2 bins per box).

`int verbose`

`FILE *ostream`

These parameters set the level of information printed by VEGAS. All information is written to the stream `ostream`. The default setting of `verbose` is `-1`, which turns off all output. A `verbose` value of `0` prints summary information about the weighted average and final result, while a value of `1` also displays the grid coordinates. A value of `2` prints information from the rebinning procedure for each iteration.

The above fields and the `chisq` value can also be accessed directly in the `gsl_monte_vegas_state` but such use is deprecated.

## 27.5 Examples

The example program below uses the Monte Carlo routines to estimate the value of the following 3-dimensional integral from the theory of random walks,

$$I = \int_{-\pi}^{+\pi} \frac{dk_x}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_y}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_z}{2\pi} \frac{1}{(1 - \cos(k_x) \cos(k_y) \cos(k_z))}.$$

The analytic value of this integral can be shown to be  $I = \Gamma(1/4)^4 / (4\pi^3) = 1.393203929685676859\dots$ . The integral gives the mean time spent at the origin by a random walk on a body-centered cubic lattice in three dimensions.

For simplicity we will compute the integral over the region  $(0, 0, 0)$  to  $(\pi, \pi, \pi)$  and multiply by 8 to obtain the full result. The integral is slowly varying in the middle of the region but has integrable singularities at the corners  $(0, 0, 0)$ ,  $(0, \pi, \pi)$ ,  $(\pi, 0, \pi)$  and  $(\pi, \pi, 0)$ . The Monte Carlo routines only select points which are strictly within the integration region and so no special measures are needed to avoid these singularities.

```
#include <stdlib.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_monte.h>
#include <gsl/gsl_monte_plain.h>
#include <gsl/gsl_monte_miser.h>
#include <gsl/gsl_monte_vegas.h>

/* Computation of the integral,

   I = int (dx dy dz)/(2pi)^3  1/(1-cos(x)cos(y)cos(z))

   over (-pi,-pi,-pi) to (+pi, +pi, +pi). The exact answer
   is Gamma(1/4)^4/(4 pi^3). This example is taken from
   C.Itzykson, J.M.Drouffe, "Statistical Field Theory -
   Volume 1", Section 1.1, p21, which cites the original
   paper M.L.Glasser, I.J.Zucker, Proc.Natl.Acad.Sci.USA 74
   1800 (1977) */
```

(continues on next page)

(continued from previous page)

```

/* For simplicity we compute the integral over the region
   (0,0,0) -> (pi,pi,pi) and multiply by 8 */

double exact = 1.3932039296856768591842462603255;

double
g (double *k, size_t dim, void *params)
{
  (void)(dim); /* avoid unused parameter warnings */
  (void)(params);
  double A = 1.0 / (M_PI * M_PI * M_PI);
  return A / (1.0 - cos (k[0]) * cos (k[1]) * cos (k[2]));
}

void
display_results (char *title, double result, double error)
{
  printf ("%s =====\n", title);
  printf ("result = % .6f\n", result);
  printf ("sigma = % .6f\n", error);
  printf ("exact = % .6f\n", exact);
  printf ("error = % .6f = %.2g sigma\n", result - exact,
          fabs (result - exact) / error);
}

int
main (void)
{
  double res, err;

  double xl[3] = { 0, 0, 0 };
  double xu[3] = { M_PI, M_PI, M_PI };

  const gsl_rng_type *T;
  gsl_rng *r;

  gsl_monte_function G = { &g, 3, 0 };

  size_t calls = 500000;

  gsl_rng_env_setup ();

  T = gsl_rng_default;
  r = gsl_rng_alloc (T);

  {
    gsl_monte_plain_state *s = gsl_monte_plain_alloc (3);
    gsl_monte_plain_integrate (&G, xl, xu, 3, calls, r, s,
                              &res, &err);
    gsl_monte_plain_free (s);
  }
}

```

(continues on next page)

(continued from previous page)

```

display_results ("plain", res, err);
}

{
gsl_monte_miser_state *s = gsl_monte_miser_alloc (3);
gsl_monte_miser_integrate (&G, xl, xu, 3, calls, r, s,
                           &res, &err);
gsl_monte_miser_free (s);

display_results ("miser", res, err);
}

{
gsl_monte_vegas_state *s = gsl_monte_vegas_alloc (3);

gsl_monte_vegas_integrate (&G, xl, xu, 3, 10000, r, s,
                           &res, &err);
display_results ("vegas warm-up", res, err);

printf ("converging...\n");

do
  {
    gsl_monte_vegas_integrate (&G, xl, xu, 3, calls/5, r, s,
                              &res, &err);
    printf ("result = % .6f sigma = % .6f "
           "chisq/dof = %.1f\n", res, err, gsl_monte_vegas_chisq (s));
  }
while (fabs (gsl_monte_vegas_chisq (s) - 1.0) > 0.5);

display_results ("vegas final", res, err);

gsl_monte_vegas_free (s);
}

gsl_rng_free (r);

return 0;
}

```

With 500,000 function calls the plain Monte Carlo algorithm achieves a fractional error of 1%. The estimated error  $\sigma$  is roughly consistent with the actual error—the computed result differs from the true result by about 1.4 standard deviations:

```

plain =====
result = 1.412209
sigma  = 0.013436
exact  = 1.393204
error  = 0.019005 = 1.4 sigma

```

The MISER algorithm reduces the error by a factor of four, and also correctly estimates the error:

```
miser =====  
result = 1.391322  
sigma = 0.003461  
exact = 1.393204  
error = -0.001882 = 0.54 sigma
```

In the case of the VEGAS algorithm the program uses an initial warm-up run of 10,000 function calls to prepare, or “warm up”, the grid. This is followed by a main run with five iterations of 100,000 function calls. The chi-squared per degree of freedom for the five iterations are checked for consistency with 1, and the run is repeated if the results have not converged. In this case the estimates are consistent on the first pass:

```
vegas warm-up =====  
result = 1.392673  
sigma = 0.003410  
exact = 1.393204  
error = -0.000531 = 0.16 sigma  
converging...  
result = 1.393281 sigma = 0.000362 chisq/dof = 1.5  
vegas final =====  
result = 1.393281  
sigma = 0.000362  
exact = 1.393204  
error = 0.000077 = 0.21 sigma
```

If the value of `chisq` had differed significantly from 1 it would indicate inconsistent results, with a correspondingly underestimated error. The final estimate from VEGAS (using a similar number of function calls) is significantly more accurate than the other two algorithms.

## 27.6 References and Further Reading

The MISER algorithm is described in the following article by Press and Farrar,

- W.H. Press, G.R. Farrar, *Recursive Stratified Sampling for Multidimensional Monte Carlo Integration*, Computers in Physics, v4 (1990), pp190–195.

The VEGAS algorithm is described in the following papers,

- G.P. Lepage, *A New Algorithm for Adaptive Multidimensional Integration*, Journal of Computational Physics 27, 192–203, (1978)
- G.P. Lepage, *VEGAS: An Adaptive Multi-dimensional Integration Program*, Cornell preprint CLNS 80-447, March 1980

## SIMULATED ANNEALING

Stochastic search techniques are used when the structure of a space is not well understood or is not smooth, so that techniques like Newton's method (which requires calculating Jacobian derivative matrices) cannot be used. In particular, these techniques are frequently used to solve combinatorial optimization problems, such as the traveling salesman problem.

The goal is to find a point in the space at which a real valued *energy function* (or *cost function*) is minimized. Simulated annealing is a minimization technique which has given good results in avoiding local minima; it is based on the idea of taking a random walk through the space at successively lower temperatures, where the probability of taking a step is given by a Boltzmann distribution.

The functions described in this chapter are declared in the header file `gsl_siman.h`.

### 28.1 Simulated Annealing algorithm

The simulated annealing algorithm takes random walks through the problem space, looking for points with low energies; in these random walks, the probability of taking a step is determined by the Boltzmann distribution,

$$p = e^{-(E_{i+1}-E_i)/(kT)}$$

if  $E_{i+1} > E_i$ , and  $p = 1$  when  $E_{i+1} \leq E_i$ .

In other words, a step will occur if the new energy is lower. If the new energy is higher, the transition can still occur, and its likelihood is proportional to the temperature  $T$  and inversely proportional to the energy difference  $E_{i+1} - E_i$ .

The temperature  $T$  is initially set to a high value, and a random walk is carried out at that temperature. Then the temperature is lowered very slightly according to a *cooling schedule*, for example:  $T \rightarrow T/\mu_T$  where  $\mu_T$  is slightly greater than 1.

The slight probability of taking a step that gives higher energy is what allows simulated annealing to frequently get out of local minima.

### 28.2 Simulated Annealing functions

```
void gsl_siman_solve(const gsl_rng *r, void *x0_p, gsl_siman_Efunc_t Ef, gsl_siman_step_t take_step,  
                    gsl_siman_metric_t distance, gsl_siman_print_t print_position, gsl_siman_copy_t  
                    copyfunc, gsl_siman_copy_construct_t copy_constructor, gsl_siman_destroy_t destructor,  
                    size_t element_size, gsl_siman_params_t params)
```

This function performs a simulated annealing search through a given space. The space is specified by providing the functions *Ef* and *distance*. The simulated annealing steps are generated using the random number generator *r* and the function *take\_step*.

The starting configuration of the system should be given by *x0\_p*. The routine offers two modes for updating configurations, a fixed-size mode and a variable-size mode. In the fixed-size mode the configuration is stored as a single block of memory of size *element\_size*. Copies of this configuration are created, copied and destroyed internally using the standard library functions `malloc()`, `memcpy()` and `free()`. The function pointers *copyfunc*, *copy\_constructor* and *destructor* should be null pointers in fixed-size mode. In the variable-size mode the functions *copyfunc*, *copy\_constructor* and *destructor* are used to create, copy and destroy configurations internally. The variable *element\_size* should be zero in the variable-size mode.

The *params* structure (described below) controls the run by providing the temperature schedule and other tunable parameters to the algorithm.

On exit the best result achieved during the search is placed in *x0\_p*. If the annealing process has been successful this should be a good approximation to the optimal point in the space.

If the function pointer *print\_position* is not null, a debugging log will be printed to `stdout` with the following columns:

```
#-iter #-evals temperature position energy best_energy
```

and the output of the function *print\_position* itself. If *print\_position* is null then no information is printed.

The simulated annealing routines require several user-specified functions to define the configuration space and energy function. The prototypes for these functions are given below.

type **gsl\_siman\_Efunc\_t**

This function type should return the energy of a configuration *xp*:

```
double (*gsl_siman_Efunc_t) (void *xp)
```

type **gsl\_siman\_step\_t**

This function type should modify the configuration *xp* using a random step taken from the generator *r*, up to a maximum distance of *step\_size*:

```
void (*gsl_siman_step_t) (const gsl_rng *r, void *xp,  
                        double step_size)
```

type **gsl\_siman\_metric\_t**

This function type should return the distance between two configurations *xp* and *yp*:

```
double (*gsl_siman_metric_t) (void *xp, void *yp)
```

type **gsl\_siman\_print\_t**

This function type should print the contents of the configuration *xp*:

```
void (*gsl_siman_print_t) (void *xp)
```

type **gsl\_siman\_copy\_t**

This function type should copy the configuration source into *dest*:

```
void (*gsl_siman_copy_t) (void *source, void *dest)
```

type **gsl\_siman\_copy\_construct\_t**

This function type should create a new copy of the configuration *xp*:

```
void * (*gsl_siman_copy_construct_t) (void *xp)
```

type `gsl_siman_destroy_t`

This function type should destroy the configuration `xp`, freeing its memory:

```
void (*gsl_siman_destroy_t) (void *xp)
```

type `gsl_siman_params_t`

These are the parameters that control a run of `gsl_siman_solve()`. This structure contains all the information needed to control the search, beyond the energy function, the step function and the initial guess.

<code>int n_tries</code>	The number of points to try for each step.
<code>int iters_fixed_T</code>	The number of iterations at each temperature.
<code>double step_size</code>	The maximum step size in the random walk.
<code>double k, t_initial, mu_t, t_min</code>	The parameters of the Boltzmann distribution and cooling schedule.

## 28.3 Examples

The simulated annealing package is clumsy, and it has to be because it is written in C, for C callers, and tries to be polymorphic at the same time. But here we provide some examples which can be pasted into your application with little change and should make things easier.

### 28.3.1 Trivial example

The first example, in one dimensional Cartesian space, sets up an energy function which is a damped sine wave; this has many local minima, but only one global minimum, somewhere between 1.0 and 1.5. The initial guess given is 15.5, which is several local minima away from the global minimum.

```
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <gsl/gsl_siman.h>

/* set up parameters for this simulated annealing run */

/* how many points do we try before stepping */
#define N_TRIES 200

/* how many iterations for each T? */
#define ITERS_FIXED_T 1000

/* max step size in random walk */
#define STEP_SIZE 1.0

/* Boltzmann constant */
#define K 1.0

/* initial temperature */
#define T_INITIAL 0.008

/* damping factor for temperature */
```

(continues on next page)

(continued from previous page)

```
#define MU_T 1.003
#define T_MIN 2.0e-6

gsl_siman_params_t params
= {N_TRIES, ITERS_FIXED_T, STEP_SIZE,
   K, T_INITIAL, MU_T, T_MIN};

/* now some functions to test in one dimension */
double E1(void *xp)
{
    double x = *(double *) xp;

    return exp(-pow((x-1.0),2.0))*sin(8*x);
}

double M1(void *xp, void *yp)
{
    double x = *((double *) xp);
    double y = *((double *) yp);

    return fabs(x - y);
}

void S1(const gsl_rng * r, void *xp, double step_size)
{
    double old_x = *((double *) xp);
    double new_x;

    double u = gsl_rng_uniform(r);
    new_x = u * 2 * step_size - step_size + old_x;

    memcpy(xp, &new_x, sizeof(new_x));
}

void P1(void *xp)
{
    printf ("%12g", *((double *) xp));
}

int
main(void)
{
    const gsl_rng_type * T;
    gsl_rng * r;

    double x_initial = 15.5;

    gsl_rng_env_setup();

    T = gsl_rng_default;
    r = gsl_rng_alloc(T);
}
```

(continues on next page)



(continued from previous page)

```

gsl_siman_solve(r, &x_initial, E1, S1, M1, P1,
                NULL, NULL, NULL,
                sizeof(double), params);

gsl_rng_free (r);
return 0;
}

```

Fig. 28.1 is generated by running `siman_test` in the following way:

```

$ ./siman_test | awk '!/^#/ {print $1, $4}'
| graph -y 1.34 1.4 -W0 -X generation -Y position
| plot -Tps > siman-test.eps

```

Fig. 28.2 is generated by running `siman_test` in the following way:

```

$ ./siman_test | awk '!/^#/ {print $1, $5}'
| graph -y -0.88 -0.83 -W0 -X generation -Y energy
| plot -Tps > siman-energy.eps

```

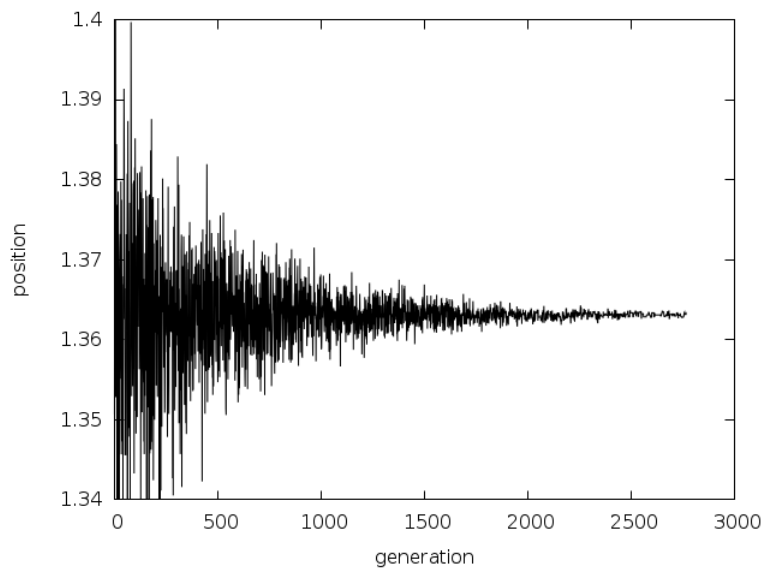


Fig. 28.1: Example of a simulated annealing run: at higher temperatures (early in the plot) you see that the solution can fluctuate, but at lower temperatures it converges.

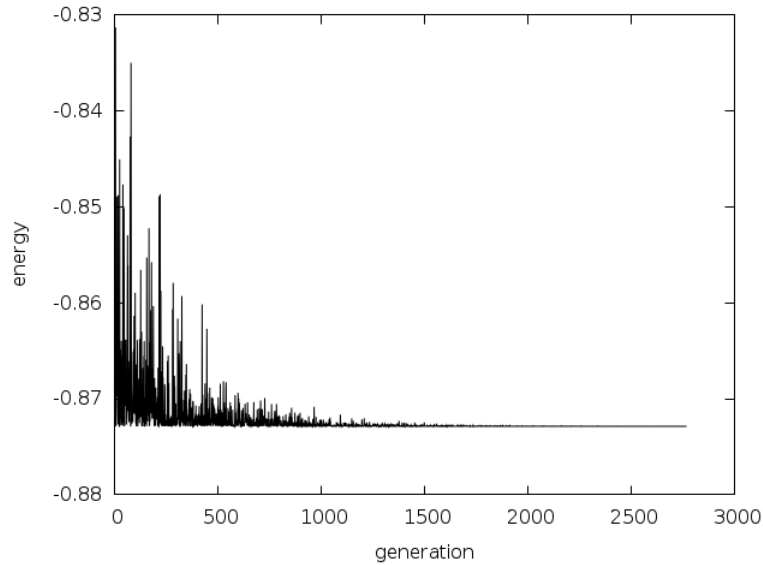


Fig. 28.2: Simulated annealing energy vs generation

### 28.3.2 Traveling Salesman Problem

The TSP (*Traveling Salesman Problem*) is the classic combinatorial optimization problem. I have provided a very simple version of it, based on the coordinates of twelve cities in the southwestern United States. This should maybe be called the *Flying Salesman Problem*, since I am using the great-circle distance between cities, rather than the driving distance. Also: I assume the earth is a sphere, so I don't use geoid distances.

The `gsl_siman_solve()` routine finds a route which is 3490.62 Kilometers long; this is confirmed by an exhaustive search of all possible routes with the same initial city.

The full code is given below.

```

/* siman/siman_tsp.c
 *
 * Copyright (C) 1996, 1997, 1998, 1999, 2000 Mark Galassi
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
 */

#include <config.h>
#include <math.h>

```

(continues on next page)

(continued from previous page)

```

#include <string.h>
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_siman.h>
#include <gsl/gsl_ieee_utils.h>

/* set up parameters for this simulated annealing run */

#define N_TRIES 200          /* how many points do we try before stepping */
#define ITERS_FIXED_T 2000  /* how many iterations for each T? */
#define STEP_SIZE 1.0       /* max step size in random walk */
#define K 1.0               /* Boltzmann constant */
#define T_INITIAL 5000.0    /* initial temperature */
#define MU_T 1.002         /* damping factor for temperature */
#define T_MIN 5.0e-1

gsl_siman_params_t params = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,
                             K, T_INITIAL, MU_T, T_MIN};

struct s_tsp_city {
    const char * name;
    double lat, longitude;    /* coordinates */
};
typedef struct s_tsp_city Stsp_city;

void prepare_distance_matrix(void);
void exhaustive_search(void);
void print_distance_matrix(void);
double city_distance(Stsp_city c1, Stsp_city c2);
double E_tsp(void *xp);
double M_tsp(void *xp, void *yp);
void Stsp(const gsl_rng * r, void *xp, double step_size);
void Ptsp(void *xp);

/* in this table, latitude and longitude are obtained from the US
   Census Bureau, at http://www.census.gov/cgi-bin/gazetteer */

Stsp_city cities[] = {"Santa Fe",    35.68,  105.95},
                     {"Phoenix",    33.54,  112.07},
                     {"Albuquerque", 35.12,  106.62},
                     {"Clovis",     34.41,  103.20},
                     {"Durango",    37.29,  107.87},
                     {"Dallas",     32.79,   96.77},
                     {"Tesuque",    35.77,  105.92},
                     {"Grants",     35.15,  107.84},
                     {"Los Alamos", 35.89,  106.28},
                     {"Las Cruces", 32.34,  106.76},
                     {"Cortez",     37.35,  108.58},
                     {"Gallup",     35.52,  108.74}};

#define N_CITIES (sizeof(cities)/sizeof(Stsp_city))

```

(continues on next page)

(continued from previous page)

```
double distance_matrix[N_CITIES][N_CITIES];

/* distance between two cities */
double city_distance(Stsp_city c1, Stsp_city c2)
{
    const double earth_radius = 6375.000; /* 6000KM approximately */
    /* sin and cos of lat and long; must convert to radians */
    double sla1 = sin(c1.lat*M_PI/180), cla1 = cos(c1.lat*M_PI/180),
        slo1 = sin(c1.longitude*M_PI/180), clo1 = cos(c1.longitude*M_PI/180);
    double sla2 = sin(c2.lat*M_PI/180), cla2 = cos(c2.lat*M_PI/180),
        slo2 = sin(c2.longitude*M_PI/180), clo2 = cos(c2.longitude*M_PI/180);

    double x1 = cla1*clo1;
    double x2 = cla2*clo2;

    double y1 = cla1*slo1;
    double y2 = cla2*slo2;

    double z1 = sla1;
    double z2 = sla2;

    double dot_product = x1*x2 + y1*y2 + z1*z2;

    double angle = acos(dot_product);

    /* distance is the angle (in radians) times the earth radius */
    return angle*earth_radius;
}

/* energy for the travelling salesman problem */
double Etsp(void *xp)
{
    /* an array of N_CITIES integers describing the order */
    int *route = (int *) xp;
    double E = 0;
    unsigned int i;

    for (i = 0; i < N_CITIES; ++i) {
        /* use the distance_matrix to optimize this calculation; it had
           better be allocated!! */
        E += distance_matrix[route[i]][route[(i + 1) % N_CITIES]];
    }

    return E;
}

double Mtsp(void *xp, void *yp)
{
    int *route1 = (int *) xp, *route2 = (int *) yp;
    double distance = 0;
    unsigned int i;
```

(continues on next page)

(continued from previous page)

```

for (i = 0; i < N_CITIES; ++i) {
    distance += ((route1[i] == route2[i]) ? 0 : 1);
}

return distance;
}

/* take a step through the TSP space */
void Stsp(const gsl_rng * r, void *xp, double step_size)
{
    int x1, x2, dummy;
    int *route = (int *) xp;

    step_size = 0 ; /* prevent warnings about unused parameter */

    /* pick the two cities to swap in the matrix; we leave the first
       city fixed */
    x1 = (gsl_rng_get (r) % (N_CITIES-1)) + 1;
    do {
        x2 = (gsl_rng_get (r) % (N_CITIES-1)) + 1;
    } while (x2 == x1);

    dummy = route[x1];
    route[x1] = route[x2];
    route[x2] = dummy;
}

void Ptsp(void *xp)
{
    unsigned int i;
    int *route = (int *) xp;
    printf("  ");
    for (i = 0; i < N_CITIES; ++i) {
        printf(" %d ", route[i]);
    }
    printf("] ");
}

int main(void)
{
    int x_initial[N_CITIES];
    unsigned int i;

    const gsl_rng * r = gsl_rng_alloc (gsl_rng_env_setup());

    gsl_ieee_env_setup ();

    prepare_distance_matrix();

    /* set up a trivial initial route */
    printf("# initial order of cities:\n");

```

(continues on next page)

(continued from previous page)

```

for (i = 0; i < N_CITIES; ++i) {
    printf("# \"%s\"\n", cities[i].name);
    x_initial[i] = i;
}

printf("# distance matrix is:\n");
print_distance_matrix();

printf("# initial coordinates of cities (longitude and latitude)\n");
/* this can be plotted with */
/* ./siman_tsp > hhh ; grep city_coord hhh | awk '{print $2 " " $3}' | xyplot -ps -d
↪ "xy" > c.eps */
for (i = 0; i < N_CITIES+1; ++i) {
    printf("###initial_city_coord: %g %g \"%s\"\n",
        -cities[x_initial[i % N_CITIES]].longitude,
        cities[x_initial[i % N_CITIES]].lat,
        cities[x_initial[i % N_CITIES]].name);
}

/* exhaustive_search(); */

gs1_siman_solve(r, x_initial, Etsp, Stsp, Mtsp, Ptsp, NULL, NULL, NULL,
    N_CITIES*sizeof(int), params);

printf("# final order of cities:\n");
for (i = 0; i < N_CITIES; ++i) {
    printf("# \"%s\"\n", cities[x_initial[i]].name);
}

printf("# final coordinates of cities (longitude and latitude)\n");
/* this can be plotted with */
/* ./siman_tsp > hhh ; grep city_coord hhh | awk '{print $2 " " $3}' | xyplot -ps -d
↪ "xy" > c.eps */
for (i = 0; i < N_CITIES+1; ++i) {
    printf("###final_city_coord: %g %g %s\n",
        -cities[x_initial[i % N_CITIES]].longitude,
        cities[x_initial[i % N_CITIES]].lat,
        cities[x_initial[i % N_CITIES]].name);
}

printf("# ");
fflush(stdout);
#if 0
    system("date");
#endif /* 0 */
fflush(stdout);

return 0;
}

void prepare_distance_matrix()
{

```

(continues on next page)

(continued from previous page)

```

unsigned int i, j;
double dist;

for (i = 0; i < N_CITIES; ++i) {
    for (j = 0; j < N_CITIES; ++j) {
        if (i == j) {
            dist = 0;
        } else {
            dist = city_distance(cities[i], cities[j]);
        }
        distance_matrix[i][j] = dist;
    }
}

void print_distance_matrix()
{
    unsigned int i, j;

    for (i = 0; i < N_CITIES; ++i) {
        printf("# ");
        for (j = 0; j < N_CITIES; ++j) {
            printf("%15.8f  ", distance_matrix[i][j]);
        }
        printf("\n");
    }
}

/* [only works for 12] search the entire space for solutions */
static double best_E = 1.0e100, second_E = 1.0e100, third_E = 1.0e100;
static int best_route[N_CITIES];
static int second_route[N_CITIES];
static int third_route[N_CITIES];
static void do_all_perms(int *route, int n);

void exhaustive_search()
{
    static int initial_route[N_CITIES] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    printf("\n# ");
    fflush(stdout);
#ifdef 0
    system("date");
#endif
    fflush(stdout);
    do_all_perms(initial_route, 1);
    printf("\n# ");
    fflush(stdout);
#ifdef 0
    system("date");
#endif /* 0 */
    fflush(stdout);
}

```

(continues on next page)

(continued from previous page)

```

printf("# exhaustive best route: ");
Ptsp(best_route);
printf("\n# its energy is: %g\n", best_E);

printf("# exhaustive second_best route: ");
Ptsp(second_route);
printf("\n# its energy is: %g\n", second_E);

printf("# exhaustive third_best route: ");
Ptsp(third_route);
printf("\n# its energy is: %g\n", third_E);
}

/* James Theiler's recursive algorithm for generating all routes */
static void do_all_perms(int *route, int n)
{
  if (n == (N_CITIES-1)) {
    /* do it! calculate the energy/cost for that route */
    double E;
    E = Etsp(route);          /* TSP energy function */
    /* now save the best 3 energies and routes */
    if (E < best_E) {
      third_E = second_E;
      memcpy(third_route, second_route, N_CITIES*sizeof(*route));
      second_E = best_E;
      memcpy(second_route, best_route, N_CITIES*sizeof(*route));
      best_E = E;
      memcpy(best_route, route, N_CITIES*sizeof(*route));
    } else if (E < second_E) {
      third_E = second_E;
      memcpy(third_route, second_route, N_CITIES*sizeof(*route));
      second_E = E;
      memcpy(second_route, route, N_CITIES*sizeof(*route));
    } else if (E < third_E) {
      third_E = E;
      memcpy(route, third_route, N_CITIES*sizeof(*route));
    }
  } else {
    int new_route[N_CITIES];
    unsigned int j;
    int swap_tmp;
    memcpy(new_route, route, N_CITIES*sizeof(*route));
    for (j = n; j < N_CITIES; ++j) {
      swap_tmp = new_route[j];
      new_route[j] = new_route[n];
      new_route[n] = swap_tmp;
      do_all_perms(new_route, n+1);
    }
  }
}

```

Below are some plots generated in the following way:



```

$ ./siman_tsp > tsp.output
$ grep -v "^#" tsp.output
| awk '{print $1, $NF}'
| graph -y 3300 6500 -W0 -X generation -Y distance
  -L "TSP - 12 southwest cities"
| plot -Tps > 12-cities.eps
$ grep initial_city_coord tsp.output
| awk '{print $2, $3}'
| graph -X "longitude (- means west)" -Y "latitude"
  -L "TSP - initial-order" -f 0.03 -S 1 0.1
| plot -Tps > initial-route.eps
$ grep final_city_coord tsp.output
| awk '{print $2, $3}'
| graph -X "longitude (- means west)" -Y "latitude"
  -L "TSP - final-order" -f 0.03 -S 1 0.1
| plot -Tps > final-route.eps

```

This is the output showing the initial order of the cities; longitude is negative, since it is west and I want the plot to look like a map:

```

# initial coordinates of cities (longitude and latitude)
###initial_city_coord: -105.95 35.68 Santa Fe
###initial_city_coord: -112.07 33.54 Phoenix
###initial_city_coord: -106.62 35.12 Albuquerque
###initial_city_coord: -103.2 34.41 Clovis
###initial_city_coord: -107.87 37.29 Durango
###initial_city_coord: -96.77 32.79 Dallas
###initial_city_coord: -105.92 35.77 Tesuque
###initial_city_coord: -107.84 35.15 Grants
###initial_city_coord: -106.28 35.89 Los Alamos
###initial_city_coord: -106.76 32.34 Las Cruces
###initial_city_coord: -108.58 37.35 Cortez
###initial_city_coord: -108.74 35.52 Gallup
###initial_city_coord: -105.95 35.68 Santa Fe

```

The optimal route turns out to be:

```

# final coordinates of cities (longitude and latitude)
###final_city_coord: -105.95 35.68 Santa Fe
###final_city_coord: -103.2 34.41 Clovis
###final_city_coord: -96.77 32.79 Dallas
###final_city_coord: -106.76 32.34 Las Cruces
###final_city_coord: -112.07 33.54 Phoenix
###final_city_coord: -108.74 35.52 Gallup
###final_city_coord: -108.58 37.35 Cortez
###final_city_coord: -107.87 37.29 Durango
###final_city_coord: -107.84 35.15 Grants
###final_city_coord: -106.62 35.12 Albuquerque
###final_city_coord: -106.28 35.89 Los Alamos
###final_city_coord: -105.92 35.77 Tesuque
###final_city_coord: -105.95 35.68 Santa Fe

```

Here's a plot of the cost function (energy) versus generation (point in the calculation at which a new temperature is set) for this problem:

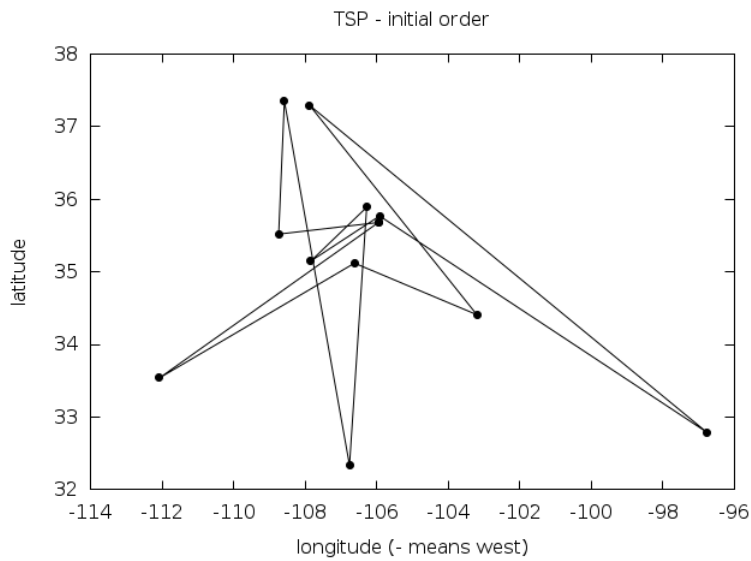


Fig. 28.3: Initial route for the 12 southwestern cities Flying Salesman Problem.

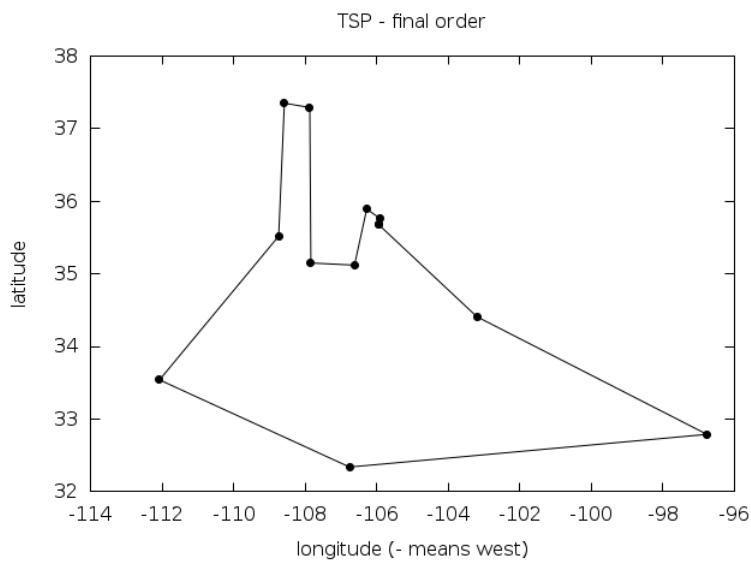


Fig. 28.4: Final (optimal) route for the 12 southwestern cities Flying Salesman Problem.

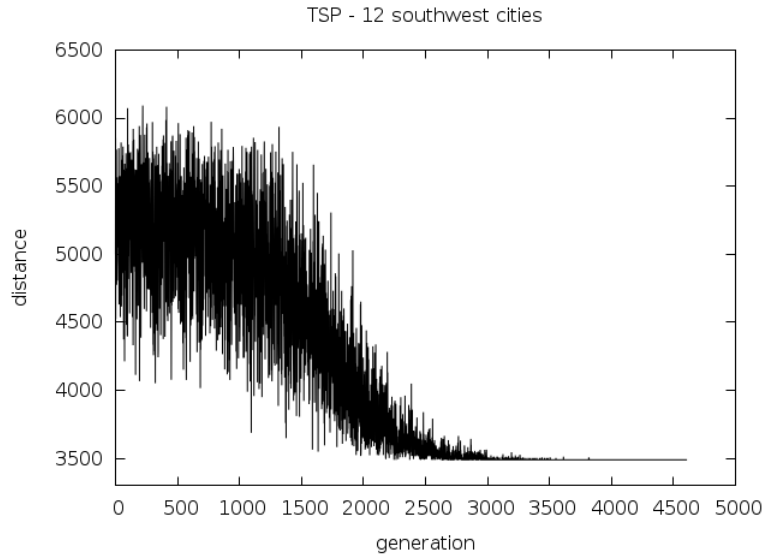


Fig. 28.5: Example of a simulated annealing run for the 12 southwestern cities Flying Salesman Problem.

## 28.4 References and Further Reading

Further information is available in the following book,

- *Modern Heuristic Techniques for Combinatorial Problems*, Colin R. Reeves (ed.), McGraw-Hill, 1995 (ISBN 0-07-709239-2).



## ORDINARY DIFFERENTIAL EQUATIONS

This chapter describes functions for solving ordinary differential equation (ODE) initial value problems. The library provides a variety of low-level methods, such as Runge-Kutta and Bulirsch-Stoer routines, and higher-level components for adaptive step-size control. The components can be combined by the user to achieve the desired solution, with full access to any intermediate steps. A driver object can be used as a high level wrapper for easy use of low level functions.

These functions are declared in the header file `gsl_odeiv2.h`. This is a new interface in version 1.15 and uses the prefix `gsl_odeiv2` for all functions. It is recommended over the previous `gsl_odeiv` implementation defined in `gsl_odeiv.h`. The old interface has been retained under the original name for backwards compatibility.

### 29.1 Defining the ODE System

The routines solve the general  $n$ -dimensional first-order system,

$$\frac{dy_i(t)}{dt} = f_i(t, y_1(t), \dots, y_n(t))$$

for  $i = 1, \dots, n$ . The stepping functions rely on the vector of derivatives  $f_i$  and the Jacobian matrix,

$$J_{ij} = \partial f_i(t, y(t)) / \partial y_j$$

A system of equations is defined using the `gsl_odeiv2_system` datatype.

#### type `gsl_odeiv2_system`

This data type defines a general ODE system with arbitrary parameters.

```
int (* function) (double t, const double y[], double dydt[], void * params)
```

This function should store the vector elements  $f_i(t, y, params)$  in the array `dydt`, for arguments (`t`, `y`) and parameters `params`.

The function should return `GSL_SUCCESS` if the calculation was completed successfully. Any other return value indicates an error. A special return value `GSL_EBADFUNC` causes `gsl_odeiv2` routines to immediately stop and return. If `function` is modified (for example contents of `params`), the user must call an appropriate reset function (`gsl_odeiv2_driver_reset()`, `gsl_odeiv2_evolve_reset()` or `gsl_odeiv2_step_reset()`) before continuing. Use return values distinct from standard GSL error codes to distinguish your function as the source of the error.

```
int (* jacobian) (double t, const double y[], double * dfdy, double dfdt[], void * params)
```

This function should store the vector of derivative elements

$$\partial f_i(t, y, params) / \partial t$$

in the array `dfdt` and the Jacobian matrix  $J_{ij}$  in the array `dfdy`, regarded as a row-ordered matrix  $J(i, j) = \text{dfdy}[i * \text{dimension} + j]$  where `dimension` is the dimension of the system.

Not all of the stepper algorithms of `gsl_odeiv2` make use of the Jacobian matrix, so it may not be necessary to provide this function (the `jacobian` element of the struct can be replaced by a null pointer for those algorithms).

The function should return `GSL_SUCCESS` if the calculation was completed successfully. Any other return value indicates an error. A special return value `GSL_EBADFUNC` causes `gsl_odeiv2` routines to immediately stop and return. If `jacobian` is modified (for example contents of `params`), the user must call an appropriate reset function (`gsl_odeiv2_driver_reset()`, `gsl_odeiv2_evolve_reset()` or `gsl_odeiv2_step_reset()`) before continuing. Use return values distinct from standard GSL error codes to distinguish your function as the source of the error.

`size_t dimension`

This is the dimension of the system of equations.

`void * params`

This is a pointer to the arbitrary parameters of the system.

## 29.2 Stepping Functions

The lowest level components are the *stepping functions* which advance a solution from time  $t$  to  $t + h$  for a fixed step-size  $h$  and estimate the resulting local error.

type `gsl_odeiv2_step`

This contains internal parameters for a stepping function.

`gsl_odeiv2_step *gsl_odeiv2_step_alloc(const gsl_odeiv2_step_type *T, size_t dim)`

This function returns a pointer to a newly allocated instance of a stepping function of type  $T$  for a system of  $dim$  dimensions. Please note that if you use a stepper method that requires access to a driver object, it is advisable to use a driver allocation method, which automatically allocates a stepper, too.

`int gsl_odeiv2_step_reset(gsl_odeiv2_step *s)`

This function resets the stepping function  $s$ . It should be used whenever the next use of  $s$  will not be a continuation of a previous step.

`void gsl_odeiv2_step_free(gsl_odeiv2_step *s)`

This function frees all the memory associated with the stepping function  $s$ .

`const char *gsl_odeiv2_step_name(const gsl_odeiv2_step *s)`

This function returns a pointer to the name of the stepping function. For example:

```
printf ("step method is '%s'\n", gsl_odeiv2_step_name (s));
```

would print something like `step method is 'rkf45'`.

`unsigned int gsl_odeiv2_step_order(const gsl_odeiv2_step *s)`

This function returns the order of the stepping function on the previous step. The order can vary if the stepping function itself is adaptive.

`int gsl_odeiv2_step_set_driver(gsl_odeiv2_step *s, const gsl_odeiv2_driver *d)`

This function sets a pointer of the driver object  $d$  for stepper  $s$ , to allow the stepper to access control (and evolve) object through the driver object. This is a requirement for some steppers, to get the desired error level for internal iteration of stepper. Allocation of a driver object calls this function automatically.

```
int gsl_odeiv2_step_apply(gsl_odeiv2_step *s, double t, double h, double y[], double yerr[], const double
dydt_in[], double dydt_out[], const gsl_odeiv2_system *sys)
```

This function applies the stepping function *s* to the system of equations defined by *sys*, using the step-size *h* to advance the system from time *t* and state *y* to time *t + h*. The new state of the system is stored in *y* on output, with an estimate of the absolute error in each component stored in *yerr*. If the argument *dydt\_in* is not null it should point an array containing the derivatives for the system at time *t* on input. This is optional as the derivatives will be computed internally if they are not provided, but allows the reuse of existing derivative information. On output the new derivatives of the system at time *t + h* will be stored in *dydt\_out* if it is not null.

The stepping function returns `GSL_FAILURE` if it is unable to compute the requested step. Also, if the user-supplied functions defined in the system *sys* return a status other than `GSL_SUCCESS` the step will be aborted. In that case, the elements of *y* will be restored to their pre-step values and the error code from the user-supplied function will be returned. Failure may be due to a singularity in the system or too large step-size *h*. In that case the step should be attempted again with a smaller step-size, e.g.  $h / 2$ .

If the driver object is not appropriately set via *gsl\_odeiv2\_step\_set\_driver()* for those steppers that need it, the stepping function returns `GSL_EFAULT`. If the user-supplied functions defined in the system *sys* returns `GSL_EBADFUNC`, the function returns immediately with the same return code. In this case the user must call *gsl\_odeiv2\_step\_reset()* before calling this function again.

The following algorithms are available. Please note that algorithms which use step doubling for error estimation apply the more accurate values from two half steps instead of values from a single step for the new state *y*.

type **gsl\_odeiv2\_step\_type**

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rk2**

Explicit embedded Runge-Kutta (2, 3) method.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rk4**

Explicit 4th order (classical) Runge-Kutta. Error estimation is carried out by the step doubling method. For more efficient estimate of the error, use the embedded methods described below.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rkf45**

Explicit embedded Runge-Kutta-Fehlberg (4, 5) method. This method is a good general-purpose integrator.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rkck**

Explicit embedded Runge-Kutta Cash-Karp (4, 5) method.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rk8pd**

Explicit embedded Runge-Kutta Prince-Dormand (8, 9) method.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rk1imp**

Implicit Gaussian first order Runge-Kutta. Also known as implicit Euler or backward Euler method. Error estimation is carried out by the step doubling method. This algorithm requires the Jacobian and access to the driver object via *gsl\_odeiv2\_step\_set\_driver()*.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rk2imp**

Implicit Gaussian second order Runge-Kutta. Also known as implicit mid-point rule. Error estimation is carried out by the step doubling method. This stepper requires the Jacobian and access to the driver object via *gsl\_odeiv2\_step\_set\_driver()*.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_rk4imp**

Implicit Gaussian 4th order Runge-Kutta. Error estimation is carried out by the step doubling method. This algorithm requires the Jacobian and access to the driver object via *gsl\_odeiv2\_step\_set\_driver()*.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_bsimp**

Implicit Bulirsch-Stoer method of Bader and Deuffhard. The method is generally suitable for stiff problems. This stepper requires the Jacobian.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_msadams**

A variable-coefficient linear multistep Adams method in Nordsieck form. This stepper uses explicit Adams-Bashforth (predictor) and implicit Adams-Moulton (corrector) methods in  $P(EC)^m$  functional iteration mode. Method order varies dynamically between 1 and 12. This stepper requires the access to the driver object via *gsl\_odeiv2\_step\_set\_driver()*.

*gsl\_odeiv2\_step\_type* \***gsl\_odeiv2\_step\_msbdf**

A variable-coefficient linear multistep backward differentiation formula (BDF) method in Nordsieck form. This stepper uses the explicit BDF formula as predictor and implicit BDF formula as corrector. A modified Newton iteration method is used to solve the system of non-linear equations. Method order varies dynamically between 1 and 5. The method is generally suitable for stiff problems. This stepper requires the Jacobian and the access to the driver object via *gsl\_odeiv2\_step\_set\_driver()*.

## 29.3 Adaptive Step-size Control

The control function examines the proposed change to the solution produced by a stepping function and attempts to determine the optimal step-size for a user-specified level of error.

type **gsl\_odeiv2\_control**

This is a workspace for controlling step size.

type **gsl\_odeiv2\_control\_type**

This specifies the control type.

*gsl\_odeiv2\_control* \***gsl\_odeiv2\_control\_standard\_new**(double eps\_abs, double eps\_rel, double a\_y, double a\_dydt)

The standard control object is a four parameter heuristic based on absolute and relative errors *eps\_abs* and *eps\_rel*, and scaling factors *a\_y* and *a\_dydt* for the system state  $y(t)$  and derivatives  $y'(t)$  respectively.

The step-size adjustment procedure for this method begins by computing the desired error level  $D_i$  for each component,

$$D_i = \epsilon_{abs} + \epsilon_{rel} * (a_y |y_i| + a_{dydt} h |y'_i|)$$

and comparing it with the observed error  $E_i = |yerr_i|$ . If the observed error  $E$  exceeds the desired error level  $D$  by more than 10% for any component then the method reduces the step-size by an appropriate factor,

$$h_{new} = h_{old} * S * (E/D)^{-1/q}$$

where  $q$  is the consistency order of the method (e.g.  $q = 4$  for 4(5) embedded RK), and  $S$  is a safety factor of 0.9. The ratio  $E/D$  is taken to be the maximum of the ratios  $E_i/D_i$ .

If the observed error  $E$  is less than 50% of the desired error level  $D$  for the maximum ratio  $E_i/D_i$  then the algorithm takes the opportunity to increase the step-size to bring the error in line with the desired level,

$$h_{new} = h_{old} * S * (E/D)^{-1/(q+1)}$$

This encompasses all the standard error scaling methods. To avoid uncontrolled changes in the stepsize, the overall scaling factor is limited to the range 1/5 to 5.

*gsl\_odeiv2\_control* \***gsl\_odeiv2\_control\_y\_new**(double eps\_abs, double eps\_rel)

This function creates a new control object which will keep the local error on each step within an absolute error of *eps\_abs* and relative error of *eps\_rel* with respect to the solution  $y_i(t)$ . This is equivalent to the standard control object with  $a_y = 1$  and  $a_{dydt} = 0$ .



`gsl_odeiv2_control *gsl_odeiv2_control_yp_new(double eps_abs, double eps_rel)`

This function creates a new control object which will keep the local error on each step within an absolute error of `eps_abs` and relative error of `eps_rel` with respect to the derivatives of the solution  $y'_i(t)$ . This is equivalent to the standard control object with `a_y = 0` and `a_dydt = 1`.

`gsl_odeiv2_control *gsl_odeiv2_control_scaled_new(double eps_abs, double eps_rel, double a_y, double a_dydt, const double scale_abs[], size_t dim)`

This function creates a new control object which uses the same algorithm as `gsl_odeiv2_control_standard_new()` but with an absolute error which is scaled for each component by the array `scale_abs`. The formula for  $D_i$  for this control object is,

$$D_i = \epsilon_{abs} s_i + \epsilon_{rel} * (a_y |y_i| + a_{dydt} h |y'_i|)$$

where  $s_i$  is the  $i$ -th component of the array `scale_abs`. The same error control heuristic is used by the Matlab ODE suite.

`gsl_odeiv2_control *gsl_odeiv2_control_alloc(const gsl_odeiv2_control_type *T)`

This function returns a pointer to a newly allocated instance of a control function of type  $T$ . This function is only needed for defining new types of control functions. For most purposes the standard control functions described above should be sufficient.

`int gsl_odeiv2_control_init(gsl_odeiv2_control *c, double eps_abs, double eps_rel, double a_y, double a_dydt)`

This function initializes the control function `c` with the parameters `eps_abs` (absolute error), `eps_rel` (relative error), `a_y` (scaling factor for  $y$ ) and `a_dydt` (scaling factor for derivatives).

`void gsl_odeiv2_control_free(gsl_odeiv2_control *c)`

This function frees all the memory associated with the control function `c`.

`int gsl_odeiv2_control_hadjust(gsl_odeiv2_control *c, gsl_odeiv2_step *s, const double y[], const double yerr[], const double dydt[], double *h)`

This function adjusts the step-size  $h$  using the control function `c`, and the current values of  $y$ , `yerr` and `dydt`. The stepping function `step` is also needed to determine the order of the method. If the error in the  $y$ -values `yerr` is found to be too large then the step-size  $h$  is reduced and the function returns `GSL_ODEIV_HADJ_DEC`. If the error is sufficiently small then  $h$  may be increased and `GSL_ODEIV_HADJ_INC` is returned. The function returns `GSL_ODEIV_HADJ_NIL` if the step-size is unchanged. The goal of the function is to estimate the largest step-size which satisfies the user-specified accuracy requirements for the current point.

`const char *gsl_odeiv2_control_name(const gsl_odeiv2_control *c)`

This function returns a pointer to the name of the control function. For example:

```
printf ("control method is '%s'\n", gsl_odeiv2_control_name (c));
```

would print something like control method is 'standard'

`int gsl_odeiv2_control_errlevel(gsl_odeiv2_control *c, const double y, const double dydt, const double h, const size_t ind, double *errlev)`

This function calculates the desired error level of the `ind`-th component to `errlev`. It requires the value ( $y$ ) and value of the derivative (`dydt`) of the component, and the current step size  $h$ .

`int gsl_odeiv2_control_set_driver(gsl_odeiv2_control *c, const gsl_odeiv2_driver *d)`

This function sets a pointer of the driver object `d` for control object `c`.

## 29.4 Evolution

The evolution function combines the results of a stepping function and control function to reliably advance the solution forward one step using an acceptable step-size.

type **gsl\_odeiv2\_evolve**

This workspace contains parameters for controlling the evolution function

*gsl\_odeiv2\_evolve* \***gsl\_odeiv2\_evolve\_alloc**(size\_t dim)

This function returns a pointer to a newly allocated instance of an evolution function for a system of *dim* dimensions.

int **gsl\_odeiv2\_evolve\_apply**(*gsl\_odeiv2\_evolve* \*e, *gsl\_odeiv2\_control* \*con, *gsl\_odeiv2\_step* \*step, const *gsl\_odeiv2\_system* \*sys, double \*t, double t1, double \*h, double y[])

This function advances the system (*e*, *sys*) from time *t* and position *y* using the stepping function *step*. The new time and position are stored in *t* and *y* on output.

The initial step-size is taken as *h*. The control function *con* is applied to check whether the local error estimated by the stepping function *step* using step-size *h* exceeds the required error tolerance. If the error is too high, the step is retried by calling *step* with a decreased step-size. This process is continued until an acceptable step-size is found. An estimate of the local error for the step can be obtained from the components of the array *e->yerr* [].

If the user-supplied functions defined in the system *sys* returns `GSL_EBADFUNC`, the function returns immediately with the same return code. In this case the user must call *gsl\_odeiv2\_step\_reset()* and *gsl\_odeiv2\_evolve\_reset()* before calling this function again.

Otherwise, if the user-supplied functions defined in the system *sys* or the stepping function *step* return a status other than `GSL_SUCCESS`, the step is retried with a decreased step-size. If the step-size decreases below machine precision, a status of `GSL_FAILURE` is returned if the user functions returned `GSL_SUCCESS`. Otherwise the value returned by user function is returned. If no acceptable step can be made, *t* and *y* will be restored to their pre-step values and *h* contains the final attempted step-size.

If the step is successful the function returns a suggested step-size for the next step in *h*. The maximum time *t1* is guaranteed not to be exceeded by the time-step. On the final time-step the value of *t* will be set to *t1* exactly.

int **gsl\_odeiv2\_evolve\_apply\_fixed\_step**(*gsl\_odeiv2\_evolve* \*e, *gsl\_odeiv2\_control* \*con, *gsl\_odeiv2\_step* \*step, const *gsl\_odeiv2\_system* \*sys, double \*t, const double h, double y[])

This function advances the ODE-system (*e*, *sys*, *con*) from time *t* and position *y* using the stepping function *step* by a specified step size *h*. If the local error estimated by the stepping function exceeds the desired error level, the step is not taken and the function returns `GSL_FAILURE`. Otherwise the value returned by user function is returned.

int **gsl\_odeiv2\_evolve\_reset**(*gsl\_odeiv2\_evolve* \*e)

This function resets the evolution function *e*. It should be used whenever the next use of *e* will not be a continuation of a previous step.

void **gsl\_odeiv2\_evolve\_free**(*gsl\_odeiv2\_evolve* \*e)

This function frees all the memory associated with the evolution function *e*.

int **gsl\_odeiv2\_evolve\_set\_driver**(*gsl\_odeiv2\_evolve* \*e, const *gsl\_odeiv2\_driver* \*d)

This function sets a pointer of the driver object *d* for evolve object *e*.

If a system has discontinuous changes in the derivatives at known points, it is advisable to evolve the system between each discontinuity in sequence. For example, if a step-change in an external driving force occurs at times  $t_a$ ,  $t_b$  and  $t_c$  then evolution should be carried out over the ranges  $(t_0, t_a)$ ,  $(t_a, t_b)$ ,  $(t_b, t_c)$ , and  $(t_c, t_1)$  separately and not directly over the range  $(t_0, t_1)$ .

## 29.5 Driver

The driver object is a high level wrapper that combines the evolution, control and stepper objects for easy use.

```
gsl_odeiv2_driver *gsl_odeiv2_driver_alloc_y_new(const gsl_odeiv2_system *sys, const gsl_odeiv2_step_type
                                               *T, const double hstart, const double epsabs, const double
                                               epsrel)
```

```
gsl_odeiv2_driver *gsl_odeiv2_driver_alloc_yp_new(const gsl_odeiv2_system *sys, const
                                                  gsl_odeiv2_step_type *T, const double hstart, const
                                                  double epsabs, const double epsrel)
```

```
gsl_odeiv2_driver *gsl_odeiv2_driver_alloc_standard_new(const gsl_odeiv2_system *sys, const
                                                         gsl_odeiv2_step_type *T, const double hstart,
                                                         const double epsabs, const double epsrel, const
                                                         double a_y, const double a_dydt)
```

```
gsl_odeiv2_driver *gsl_odeiv2_driver_alloc_scaled_new(const gsl_odeiv2_system *sys, const
                                                       gsl_odeiv2_step_type *T, const double hstart, const
                                                       double epsabs, const double epsrel, const double
                                                       a_y, const double a_dydt, const double
                                                       scale_abs[])
```

These functions return a pointer to a newly allocated instance of a driver object. The functions automatically allocate and initialise the evolve, control and stepper objects for ODE system *sys* using stepper type *T*. The initial step size is given in *hstart*. The rest of the arguments follow the syntax and semantics of the control functions with same name (*gsl\_odeiv2\_control\_\*\_new*).

```
int gsl_odeiv2_driver_set_hmin(gsl_odeiv2_driver *d, const double hmin)
    The function sets a minimum for allowed step size hmin for driver d. Default value is 0.
```

```
int gsl_odeiv2_driver_set_hmax(gsl_odeiv2_driver *d, const double hmax)
    The function sets a maximum for allowed step size hmax for driver d. Default value is GSL_DBL_MAX.
```

```
int gsl_odeiv2_driver_set_nmax(gsl_odeiv2_driver *d, const unsigned long int nmax)
    The function sets a maximum for allowed number of steps nmax for driver d. Default value of 0 sets no limit for
    steps.
```

```
int gsl_odeiv2_driver_apply(gsl_odeiv2_driver *d, double *t, const double t1, double y[])
    This function evolves the driver system d from t to t1. Initially vector y should contain the values of
    dependent variables at point t. If the function is unable to complete the calculation, an error code from
    gsl_odeiv2_evolve_apply() is returned, and t and y contain the values from last successful step.
```

If maximum number of steps is reached, a value of `GSL_EMAXITER` is returned. If the step size drops below minimum value, the function returns with `GSL_ENOPROG`. If the user-supplied functions defined in the system *sys* returns `GSL_EBADFUNC`, the function returns immediately with the same return code. In this case the user must call *gsl\_odeiv2\_driver\_reset()* before calling this function again.

```
int gsl_odeiv2_driver_apply_fixed_step(gsl_odeiv2_driver *d, double *t, const double h, const unsigned
                                       long int n, double y[])
    This function evolves the driver system d from t with n steps of size h. If the function is unable to complete the
    calculation, an error code from gsl_odeiv2_evolve_apply_fixed_step() is returned, and t and y contain
    the values from last successful step.
```

```
int gsl_odeiv2_driver_reset(gsl_odeiv2_driver *d)
    This function resets the evolution and stepper objects.
```

int `gsl_odeiv2_driver_reset_hstart`(gsl\_odeiv2\_driver \*d, const double hstart)

The routine resets the evolution and stepper objects and sets new initial step size to `hstart`. This function can be used e.g. to change the direction of integration.

int `gsl_odeiv2_driver_free`(gsl\_odeiv2\_driver \*d)

This function frees the driver object, and the related evolution, stepper and control objects.

## 29.6 Examples

The following program solves the second-order nonlinear Van der Pol oscillator equation,

$$u''(t) + \mu u'(t)(u(t)^2 - 1) + u(t) = 0$$

This can be converted into a first order system suitable for use with the routines described in this chapter by introducing a separate variable for the velocity,  $v = u'(t)$ ,

$$\begin{aligned} u' &= v \\ v' &= -u + \mu v(1 - u^2) \end{aligned}$$

The program begins by defining functions for these derivatives and their Jacobian. The main function uses driver level functions to solve the problem. The program evolves the solution from  $(u, v) = (1, 0)$  at  $t = 0$  to  $t = 100$ . The step-size  $h$  is automatically adjusted by the controller to maintain an absolute accuracy of  $10^{-6}$  in the function values  $(u, v)$ . The loop in the example prints the solution at the points  $t_i = 1, 2, \dots, 100$ .

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv2.h>

int
func (double t, const double y[], double f[],
      void *params)
{
    (void)(t); /* avoid unused parameter warning */
    double mu = *(double *)params;
    f[0] = y[1];
    f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
    return GSL_SUCCESS;
}

int
jac (double t, const double y[], double *dfdy,
     double dfdt[], void *params)
{
    (void)(t); /* avoid unused parameter warning */
    double mu = *(double *)params;
    gsl_matrix_view dfdy_mat
        = gsl_matrix_view_array (dfdy, 2, 2);
    gsl_matrix * m = &dfdy_mat.matrix;
    gsl_matrix_set (m, 0, 0, 0.0);
    gsl_matrix_set (m, 0, 1, 1.0);
    gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
    gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
}
```

(continues on next page)

(continued from previous page)

```

dfdt[0] = 0.0;
dfdt[1] = 0.0;
return GSL_SUCCESS;
}

int
main (void)
{
    double mu = 10;
    gsl_odeiv2_system sys = {func, jac, 2, &mu};

    gsl_odeiv2_driver * d =
        gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd,
                                       1e-6, 1e-6, 0.0);

    int i;
    double t = 0.0, t1 = 100.0;
    double y[2] = { 1.0, 0.0 };

    for (i = 1; i <= 100; i++)
    {
        double ti = i * t1 / 100.0;
        int status = gsl_odeiv2_driver_apply (d, &t, ti, y);

        if (status != GSL_SUCCESS)
        {
            printf ("error, return value=%d\n", status);
            break;
        }

        printf (".5e %.5e %.5e\n", t, y[0], y[1]);
    }

    gsl_odeiv2_driver_free (d);
    return 0;
}

```

The user can work with the lower level functions directly, as in the following example. In this case an intermediate result is printed after each successful step instead of equidistant time points.

```

int
main (void)
{
    const gsl_odeiv2_step_type * T
        = gsl_odeiv2_step_rk8pd;

    gsl_odeiv2_step * s
        = gsl_odeiv2_step_alloc (T, 2);
    gsl_odeiv2_control * c
        = gsl_odeiv2_control_y_new (1e-6, 0.0);
    gsl_odeiv2_evolve * e
        = gsl_odeiv2_evolve_alloc (2);

```

(continues on next page)

(continued from previous page)

```

double mu = 10;
gsl_odeiv2_system sys = {func, jac, 2, &mu};

double t = 0.0, t1 = 100.0;
double h = 1e-6;
double y[2] = { 1.0, 0.0 };

while (t < t1)
{
    int status = gsl_odeiv2_evolve_apply (e, c, s,
                                         &sys,
                                         &t, t1,
                                         &h, y);

    if (status != GSL_SUCCESS)
        break;

    printf ("%5e %5e %5e\n", t, y[0], y[1]);
}

gsl_odeiv2_evolve_free (e);
gsl_odeiv2_control_free (c);
gsl_odeiv2_step_free (s);
return 0;
}

```

For functions with multiple parameters, the appropriate information can be passed in through the `params` argument in `gsl_odeiv2_system` definition (`mu` in this example) by using a pointer to a struct.

It is also possible to work with a non-adaptive integrator, using only the stepping function itself, `gsl_odeiv2_driver_apply_fixed_step()` or `gsl_odeiv2_evolve_apply_fixed_step()`. The following program uses the driver level function, with fourth-order Runge-Kutta stepping function with a fixed stepsize of 0.001.

```

int
main (void)
{
    double mu = 10;
    gsl_odeiv2_system sys = { func, jac, 2, &mu };

    gsl_odeiv2_driver *d =
        gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk4,
                                       1e-3, 1e-8, 1e-8);

    double t = 0.0;
    double y[2] = { 1.0, 0.0 };
    int i, s;

    for (i = 0; i < 100; i++)
    {
        s = gsl_odeiv2_driver_apply_fixed_step (d, &t, 1e-3, 1000, y);

        if (s != GSL_SUCCESS)

```

(continues on next page)

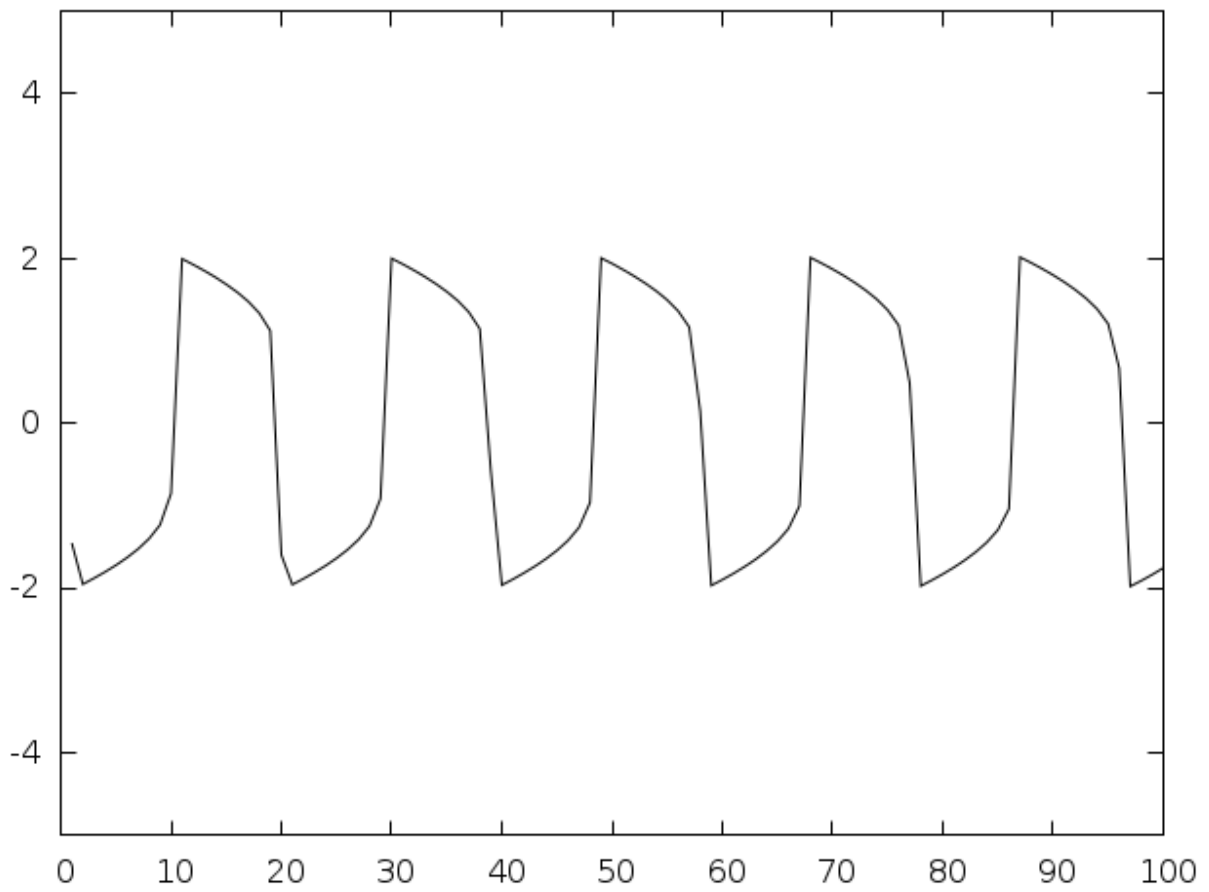


Fig. 29.1: Numerical solution of the Van der Pol oscillator equation using Prince-Dormand 8th order Runge-Kutta.

(continued from previous page)

```
    {
        printf ("error: driver returned %d\n", s);
        break;
    }

    printf ("%5e %5e %5e\n", t, y[0], y[1]);
}

gsl_odeiv2_driver_free (d);
return s;
}
```

## 29.7 References and Further Reading

- Ascher, U.M., Petzold, L.R., *Computer Methods for Ordinary Differential and Differential-Algebraic Equations*, SIAM, Philadelphia, 1998.
- Hairer, E., Norsett, S. P., Wanner, G., *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer, Berlin, 1993.
- Hairer, E., Wanner, G., *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, Springer, Berlin, 1996.

Many of the basic Runge-Kutta formulas can be found in the Handbook of Mathematical Functions,

- Abramowitz & Stegun (eds.), *Handbook of Mathematical Functions*, Section 25.5.

The implicit Bulirsch-Stoer algorithm `bsimp` is described in the following paper,

- G. Bader and P. Deuffhard, “A Semi-Implicit Mid-Point Rule for Stiff Systems of Ordinary Differential Equations.”, *Numer. Math.* 41, 373–398, 1983.

The Adams and BDF multistep methods `msadams` and `msbdf` are based on the following articles,

- G. D. Byrne and A. C. Hindmarsh, “A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations.”, *ACM Trans. Math. Software*, 1, 71–96, 1975.
- P. N. Brown, G. D. Byrne and A. C. Hindmarsh, “VODE: A Variable-coefficient ODE Solver.”, *SIAM J. Sci. Stat. Comput.* 10, 1038–1051, 1989.
- A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker and C. S. Woodward, “SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers.”, *ACM Trans. Math. Software* 31, 363–396, 2005.



## INTERPOLATION

This chapter describes functions for performing interpolation. The library provides a variety of interpolation methods, including Cubic, Akima, and Steffen splines. The interpolation types are interchangeable, allowing different methods to be used without recompiling. Interpolations can be defined for both normal and periodic boundary conditions. Additional functions are available for computing derivatives and integrals of interpolating functions. Routines are provided for interpolating both one and two dimensional datasets.

These interpolation methods produce curves that pass through each datapoint. To interpolate noisy data with a smoothing curve see *Basis Splines*.

The functions described in this section are declared in the header files `gsl_interp.h` and `gsl_spline.h`.

### 30.1 Introduction to 1D Interpolation

Given a set of data points  $(x_1, y_1) \dots (x_n, y_n)$  the routines described in this section compute a continuous interpolating function  $y(x)$  such that  $y(x_i) = y_i$ . The interpolation is piecewise smooth, and its behavior at the end-points is determined by the type of interpolation used.

### 30.2 1D Interpolation Functions

The interpolation function for a given dataset is stored in a `gsl_interp` object. These are created by the following functions.

type **gsl\_interp**

Workspace for 1D interpolation

`gsl_interp *gsl_interp_alloc(const gsl_interp_type *T, size_t size)`

This function returns a pointer to a newly allocated interpolation object of type *T* for *size* data-points.

int **gsl\_interp\_init**(`gsl_interp *interp`, const double xa[], const double ya[], size\_t size)

This function initializes the interpolation object *interp* for the data  $(xa, ya)$  where *xa* and *ya* are arrays of size *size*. The interpolation object (`gsl_interp`) does not save the data arrays *xa* and *ya* and only stores the static state computed from the data. The *xa* data array is always assumed to be strictly ordered, with increasing *x* values; the behavior for other arrangements is not defined.

void **gsl\_interp\_free**(`gsl_interp *interp`)

This function frees the interpolation object *interp*.

## 30.3 1D Interpolation Types

The interpolation library provides the following interpolation types:

type **`gsl_interp_type`**

*`gsl_interp_type`* \***`gsl_interp_linear`**

Linear interpolation. This interpolation method does not require any additional memory.

*`gsl_interp_type`* \***`gsl_interp_polynomial`**

Polynomial interpolation. This method should only be used for interpolating small numbers of points because polynomial interpolation introduces large oscillations, even for well-behaved datasets. The number of terms in the interpolating polynomial is equal to the number of points.

*`gsl_interp_type`* \***`gsl_interp_cspline`**

Cubic spline with natural boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The second derivative is chosen to be zero at the first point and last point.

*`gsl_interp_type`* \***`gsl_interp_cspline_periodic`**

Cubic spline with periodic boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The derivatives at the first and last points are also matched. Note that the last point in the data must have the same y-value as the first point, otherwise the resulting periodic interpolation will have a discontinuity at the boundary.

*`gsl_interp_type`* \***`gsl_interp_akima`**

Non-rounded Akima spline with natural boundary conditions. This method uses the non-rounded corner algorithm of Wodicka.

*`gsl_interp_type`* \***`gsl_interp_akima_periodic`**

Non-rounded Akima spline with periodic boundary conditions. This method uses the non-rounded corner algorithm of Wodicka.

*`gsl_interp_type`* \***`gsl_interp_steffen`**

Steffen's method guarantees the monotonicity of the interpolating function between the given data points. Therefore, minima and maxima can only occur exactly at the data points, and there can never be spurious oscillations between data points. The interpolated function is piecewise cubic in each interval. The resulting curve and its first derivative are guaranteed to be continuous, but the second derivative may be discontinuous.

The following related functions are available:

const char \***`gsl_interp_name`**(const *`gsl_interp`* \*interp)

This function returns the name of the interpolation type used by *interp*. For example:

```
printf ("interp uses '%s' interpolation.\n", gsl_interp_name (interp));
```

would print something like:

```
interp uses 'cspline' interpolation.
```

unsigned int **`gsl_interp_min_size`**(const *`gsl_interp`* \*interp)

unsigned int **`gsl_interp_type_min_size`**(const *`gsl_interp_type`* \*T)

These functions return the minimum number of points required by the interpolation object *interp* or interpolation type *T*. For example, Akima spline interpolation requires a minimum of 5 points.

## 30.4 1D Index Look-up and Acceleration

The state of searches can be stored in a *gsl\_interp\_accel* object, which is a kind of iterator for interpolation lookups.

type **gsl\_interp\_accel**

This workspace stores state variables for interpolation lookups. It caches the previous value of an index lookup. When the subsequent interpolation point falls in the same interval its index value can be returned immediately.

size\_t **gsl\_interp\_bsearch**(const double *x\_array*[], double *x*, size\_t *index\_lo*, size\_t *index\_hi*)

This function returns the index *i* of the array *x\_array* such that  $x\_array[i] \leq x < x\_array[i+1]$ . The index is searched for in the range [*index\_lo*, *index\_hi*]. An inline version of this function is used when HAVE\_INLINE is defined.

*gsl\_interp\_accel* \***gsl\_interp\_accel\_alloc**(void)

This function returns a pointer to an accelerator object, which is a kind of iterator for interpolation lookups. It tracks the state of lookups, thus allowing for application of various acceleration strategies. When multiple interpolants are in use, the same accelerator object may be used for all datasets with the same domain (*x\_array*), but different accelerators should be used for data defined on different domains.

size\_t **gsl\_interp\_accel\_find**(*gsl\_interp\_accel* \**a*, const double *x\_array*[], size\_t *size*, double *x*)

This function performs a lookup action on the data array *x\_array* of size *size*, using the given accelerator *a*. This is how lookups are performed during evaluation of an interpolation. The function returns an index *i* such that  $x\_array[i] \leq x < x\_array[i+1]$ . An inline version of this function is used when HAVE\_INLINE is defined.

int **gsl\_interp\_accel\_reset**(*gsl\_interp\_accel* \**acc*);

This function reinitializes the accelerator object *acc*. It should be used when the cached information is no longer applicable—for example, when switching to a new dataset.

void **gsl\_interp\_accel\_free**(*gsl\_interp\_accel* \**acc*)

This function frees the accelerator object *acc*.

## 30.5 1D Evaluation of Interpolating Functions

double **gsl\_interp\_eval**(const *gsl\_interp* \**interp*, const double *xa*[], const double *ya*[], double *x*, *gsl\_interp\_accel* \**acc*)

int **gsl\_interp\_eval\_e**(const *gsl\_interp* \**interp*, const double *xa*[], const double *ya*[], double *x*, *gsl\_interp\_accel* \**acc*, double \**y*)

These functions return the interpolated value of *y* for a given point *x*, using the interpolation object *interp*, data arrays *xa* and *ya* and the accelerator *acc*. When *x* is outside the range of *xa*, the error code *GSL\_EDOM* is returned with a value of *GSL\_NAN* for *y*.

double **gsl\_interp\_eval\_deriv**(const *gsl\_interp* \**interp*, const double *xa*[], const double *ya*[], double *x*, *gsl\_interp\_accel* \**acc*)

int **gsl\_interp\_eval\_deriv\_e**(const *gsl\_interp* \**interp*, const double *xa*[], const double *ya*[], double *x*, *gsl\_interp\_accel* \**acc*, double \**d*)

These functions return the derivative *d* of an interpolated function for a given point *x*, using the interpolation object *interp*, data arrays *xa* and *ya* and the accelerator *acc*.

double **gsl\_interp\_eval\_deriv2**(const *gsl\_interp* \**interp*, const double *xa*[], const double *ya*[], double *x*, *gsl\_interp\_accel* \**acc*)

```
int gsl_interp_eval_deriv2_e(const gsl_interp *interp, const double xa[], const double ya[], double x,
                             gsl_interp_accel *acc, double *d2)
```

These functions return the second derivative *d2* of an interpolated function for a given point *x*, using the interpolation object *interp*, data arrays *xa* and *ya* and the accelerator *acc*.

```
double gsl_interp_eval_integ(const gsl_interp *interp, const double xa[], const double ya[], double a, double b,
                              gsl_interp_accel *acc)
```

```
int gsl_interp_eval_integ_e(const gsl_interp *interp, const double xa[], const double ya[], double a, double b,
                             gsl_interp_accel *acc, double *result)
```

These functions return the numerical integral *result* of an interpolated function over the range [*a*, *b*], using the interpolation object *interp*, data arrays *xa* and *ya* and the accelerator *acc*.

## 30.6 1D Higher-level Interface

The functions described in the previous sections required the user to supply pointers to the *x* and *y* arrays on each call. The following functions are equivalent to the corresponding *gsl\_interp* functions but maintain a copy of this data in the *gsl\_spline* object. This removes the need to pass both *xa* and *ya* as arguments on each evaluation. These functions are defined in the header file *gsl\_spline.h*.

```
type gsl_spline
```

This workspace provides a higher level interface for the *gsl\_interp* object

```
gsl_spline *gsl_spline_alloc(const gsl_interp_type *T, size_t size)
```

```
int gsl_spline_init(gsl_spline *spline, const double xa[], const double ya[], size_t size)
```

```
void gsl_spline_free(gsl_spline *spline)
```

```
const char *gsl_spline_name(const gsl_spline *spline)
```

```
unsigned int gsl_spline_min_size(const gsl_spline *spline)
```

```
double gsl_spline_eval(const gsl_spline *spline, double x, gsl_interp_accel *acc)
```

```
int gsl_spline_eval_e(const gsl_spline *spline, double x, gsl_interp_accel *acc, double *y)
```

```
double gsl_spline_eval_deriv(const gsl_spline *spline, double x, gsl_interp_accel *acc)
```

```
int gsl_spline_eval_deriv_e(const gsl_spline *spline, double x, gsl_interp_accel *acc, double *d)
```

```
double gsl_spline_eval_deriv2(const gsl_spline *spline, double x, gsl_interp_accel *acc)
```

```
int gsl_spline_eval_deriv2_e(const gsl_spline *spline, double x, gsl_interp_accel *acc, double *d2)
```

```
double gsl_spline_eval_integ(const gsl_spline *spline, double a, double b, gsl_interp_accel *acc)
```

```
int gsl_spline_eval_integ_e(const gsl_spline *spline, double a, double b, gsl_interp_accel *acc, double
                           *result)
```

## 30.7 1D Interpolation Example Programs

The following program demonstrates the use of the interpolation and spline functions. It computes a cubic spline interpolation of the 10-point dataset  $(x_i, y_i)$  where  $x_i = i + \sin(i)/2$  and  $y_i = i + \cos(i^2)$  for  $i = 0 \dots 9$ .

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

int
main (void)
{
    int i;
    double xi, yi, x[10], y[10];

    printf ("#m=0,S=17\n");

    for (i = 0; i < 10; i++)
    {
        x[i] = i + 0.5 * sin (i);
        y[i] = i + cos (i * i);
        printf ("%g %g\n", x[i], y[i]);
    }

    printf ("#m=1,S=0\n");

    {
        gsl_interp_accel *acc
            = gsl_interp_accel_alloc ();
        gsl_spline *spline
            = gsl_spline_alloc (gsl_interp_cspline, 10);

        gsl_spline_init (spline, x, y, 10);

        for (xi = x[0]; xi < x[9]; xi += 0.01)
        {
            yi = gsl_spline_eval (spline, xi, acc);
            printf ("%g %g\n", xi, yi);
        }
        gsl_spline_free (spline);
        gsl_interp_accel_free (acc);
    }
    return 0;
}
```

The output is designed to be used with the GNU plotutils graph program:

```
$ ./a.out > interp.dat
$ graph -T ps < interp.dat > interp.ps
```

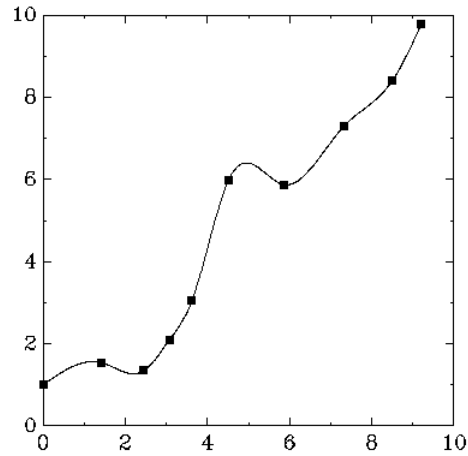


Fig. 30.1: Cubic spline interpolation

Fig. 30.1 shows a smooth interpolation of the original points. The interpolation method can be changed simply by varying the first argument of `gsl_spline_alloc()`.

The next program demonstrates a periodic cubic spline with 4 data points. Note that the first and last points must be supplied with the same y-value for a periodic spline.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

int
main (void)
{
    int N = 4;
    double x[4] = {0.00, 0.10, 0.27, 0.30};
    double y[4] = {0.15, 0.70, -0.10, 0.15};
    /* Note: y[0] == y[3] for periodic data */

    gsl_interp_accel *acc = gsl_interp_accel_alloc ();
    const gsl_interp_type *t = gsl_interp_cspline_periodic;
    gsl_spline *spline = gsl_spline_alloc (t, N);

    int i; double xi, yi;
```

(continues on next page)

(continued from previous page)

```

printf ("#m=0,S=5\n");
for (i = 0; i < N; i++)
{
    printf ("%g %g\n", x[i], y[i]);
}

printf ("#m=1,S=0\n");
gsl_spline_init (spline, x, y, N);

for (i = 0; i <= 100; i++)
{
    xi = (1 - i / 100.0) * x[0] + (i / 100.0) * x[N-1];
    yi = gsl_spline_eval (spline, xi, acc);
    printf ("%g %g\n", xi, yi);
}

gsl_spline_free (spline);
gsl_interp_accel_free (acc);
return 0;
}

```

The output can be plotted with GNU graph:

```

$ ./a.out > interp.dat
$ graph -T ps < interp.dat > interp.ps

```

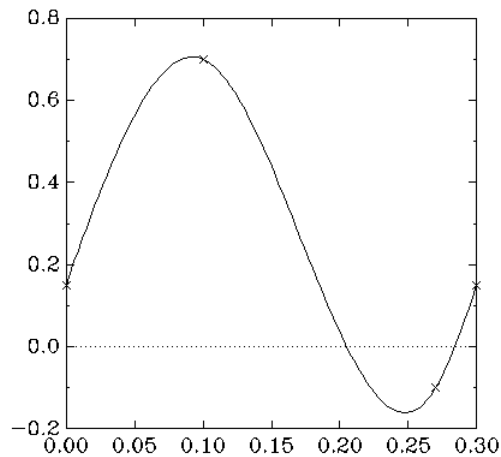


Fig. 30.2: Periodic cubic spline interpolation

Fig. 30.2 shows a periodic interpolation of the original points. The slope of the fitted curve is the same at the beginning and end of the data, and the second derivative is also.

The next program illustrates the difference between the cubic spline, Akima, and Steffen interpolation types on a difficult dataset.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_spline.h>

int
main(void)
{
    size_t i;
    const size_t N = 9;

    /* this dataset is taken from
     * J. M. Hyman, Accurate Monotonicity preserving cubic interpolation,
     * SIAM J. Sci. Stat. Comput. 4, 4, 1983. */
    const double x[] = { 7.99, 8.09, 8.19, 8.7, 9.2,
                        10.0, 12.0, 15.0, 20.0 };
    const double y[] = { 0.0, 2.76429e-5, 4.37498e-2,
                        0.169183, 0.469428, 0.943740,
                        0.998636, 0.999919, 0.999994 };

    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    gsl_spline *spline_cubic = gsl_spline_alloc(gsl_interp_cspline, N);
    gsl_spline *spline_akima = gsl_spline_alloc(gsl_interp_akima, N);
    gsl_spline *spline_steffen = gsl_spline_alloc(gsl_interp_steffen, N);

    gsl_spline_init(spline_cubic, x, y, N);
    gsl_spline_init(spline_akima, x, y, N);
    gsl_spline_init(spline_steffen, x, y, N);

    for (i = 0; i < N; ++i)
        printf("%g %g\n", x[i], y[i]);

    printf("\n\n");

    for (i = 0; i <= 100; ++i)
    {
        double xi = (1 - i / 100.0) * x[0] + (i / 100.0) * x[N-1];
        double yi_cubic = gsl_spline_eval(spline_cubic, xi, acc);
        double yi_akima = gsl_spline_eval(spline_akima, xi, acc);
        double yi_steffen = gsl_spline_eval(spline_steffen, xi, acc);

        printf("%g %g %g %g\n", xi, yi_cubic, yi_akima, yi_steffen);
    }

    gsl_spline_free(spline_cubic);
    gsl_spline_free(spline_akima);
    gsl_spline_free(spline_steffen);
    gsl_interp_accel_free(acc);

```

(continues on next page)



(continued from previous page)

```

return 0;
}

```

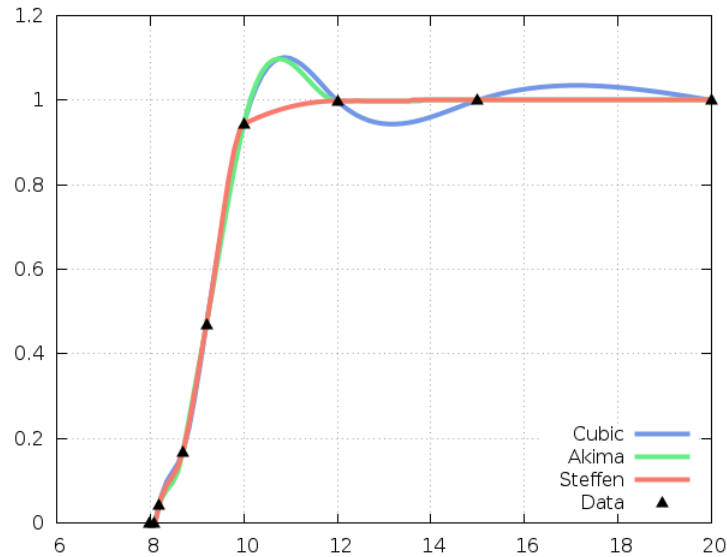


Fig. 30.3: Comparison of different 1D interpolation methods

The output is shown in Fig. 30.3. The cubic method exhibits a local maxima between the 6th and 7th data points and continues oscillating for the rest of the data. Akima also shows a local maxima but recovers and follows the data well after the 7th grid point. Steffen preserves monotonicity in all intervals and does not exhibit oscillations, at the expense of having a discontinuous second derivative.

## 30.8 Introduction to 2D Interpolation

Given a set of  $x$  coordinates  $x_1, \dots, x_m$  and a set of  $y$  coordinates  $y_1, \dots, y_n$ , each in increasing order, plus a set of function values  $z_{ij}$  for each grid point  $(x_i, y_j)$ , the routines described in this section compute a continuous interpolation function  $z(x, y)$  such that  $z(x_i, y_j) = z_{ij}$ .

## 30.9 2D Interpolation Functions

The interpolation function for a given dataset is stored in a *gsl\_interp2d* object. These are created by the following functions.

type **gsl\_interp2d**

Workspace for 2D interpolation

*gsl\_interp2d* \***gsl\_interp2d\_alloc**(const *gsl\_interp2d\_type* \*T, const size\_t xsize, const size\_t ysize)

This function returns a pointer to a newly allocated interpolation object of type *T* for *xsize* grid points in the  $x$  direction and *ysize* grid points in the  $y$  direction.

int **gsl\_interp2d\_init**(*gsl\_interp2d* \*interp, const double xa[], const double ya[], const double za[], const size\_t xsize, const size\_t ysize)

This function initializes the interpolation object *interp* for the data  $(xa, ya, za)$  where *xa* and *ya* are arrays

of the  $x$  and  $y$  grid points of size  $xsize$  and  $ysize$  respectively, and  $za$  is an array of function values of size  $xsize * ysize$ . The interpolation object (*gsl\_interp2d*) does not save the data arrays  $xa$ ,  $ya$ , and  $za$  and only stores the static state computed from the data. The  $xa$  and  $ya$  data arrays are always assumed to be strictly ordered, with increasing  $x, y$  values; the behavior for other arrangements is not defined.

void **gsl\_interp2d\_free**(*gsl\_interp2d* \*interp)

This function frees the interpolation object *interp*.

## 30.10 2D Interpolation Grids

The 2D interpolation routines access the function values  $z_{ij}$  with the following ordering:

$$z_{ij} = za[j * xsize + i]$$

with  $i = 0, \dots, xsize - 1$  and  $j = 0, \dots, ysize - 1$ . However, for ease of use, the following functions are provided to add and retrieve elements from the function grid without requiring knowledge of the internal ordering.

int **gsl\_interp2d\_set**(const *gsl\_interp2d* \*interp, double za[], const size\_t i, const size\_t j, const double z)

This function sets the value  $z_{ij}$  for grid point  $(i, j)$  of the array  $za$  to  $z$ .

double **gsl\_interp2d\_get**(const *gsl\_interp2d* \*interp, const double za[], const size\_t i, const size\_t j)

This function returns the value  $z_{ij}$  for grid point  $(i, j)$  stored in the array  $za$ .

size\_t **gsl\_interp2d\_idx**(const *gsl\_interp2d* \*interp, const size\_t i, const size\_t j)

This function returns the index corresponding to the grid point  $(i, j)$ . The index is given by  $j * xsize + i$ .

## 30.11 2D Interpolation Types

type **gsl\_interp2d\_type**

The interpolation library provides the following 2D interpolation types:

*gsl\_interp2d\_type* \***gsl\_interp2d\_bilinear**

Bilinear interpolation. This interpolation method does not require any additional memory.

*gsl\_interp2d\_type* \***gsl\_interp2d\_bicubic**

Bicubic interpolation.

const char \***gsl\_interp2d\_name**(const *gsl\_interp2d* \*interp)

This function returns the name of the interpolation type used by *interp*. For example:

```
printf ("interp uses '%s' interpolation.\n", gsl_interp2d_name (interp));
```

would print something like:

```
interp uses 'bilinear' interpolation.
```

unsigned int **gsl\_interp2d\_min\_size**(const *gsl\_interp2d* \*interp)

unsigned int **gsl\_interp2d\_type\_min\_size**(const *gsl\_interp2d\_type* \*T)

These functions return the minimum number of points required by the interpolation object *interp* or interpolation type  $T$ . For example, bicubic interpolation requires a minimum of 4 points.

## 30.12 2D Evaluation of Interpolating Functions

```
double gsl_interp2d_eval(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[],
                        const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc)
```

```
int gsl_interp2d_eval_e(const gsl_interp2d *interp, const double xa[], const double ya[], const double za[], const
                        double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel *yacc, double *z)
```

These functions return the interpolated value of  $z$  for a given point  $(x, y)$ , using the interpolation object *interp*, data arrays *xa*, *ya*, and *za* and the accelerators *xacc* and *yacc*. When  $x$  is outside the range of *xa* or  $y$  is outside the range of *ya*, the error code *GSL\_EDOM* is returned.

```
double gsl_interp2d_eval_extrap(const gsl_interp2d *interp, const double xa[], const double ya[], const double
                                za[], const double x, const double y, gsl_interp_accel *xacc,
                                gsl_interp_accel *yacc)
```

```
int gsl_interp2d_eval_extrap_e(const gsl_interp2d *interp, const double xa[], const double ya[], const double
                                za[], const double x, const double y, gsl_interp_accel *xacc, gsl_interp_accel
                                *yacc, double *z)
```

These functions return the interpolated value of  $z$  for a given point  $(x, y)$ , using the interpolation object *interp*, data arrays *xa*, *ya*, and *za* and the accelerators *xacc* and *yacc*. The functions perform no bounds checking, so when  $x$  is outside the range of *xa* or  $y$  is outside the range of *ya*, extrapolation is performed.

```
double gsl_interp2d_eval_deriv_x(const gsl_interp2d *interp, const double xa[], const double ya[], const
                                double za[], const double x, const double y, gsl_interp_accel *xacc,
                                gsl_interp_accel *yacc)
```

```
int gsl_interp2d_eval_deriv_x_e(const gsl_interp2d *interp, const double xa[], const double ya[], const double
                                za[], const double x, const double y, gsl_interp_accel *xacc,
                                gsl_interp_accel *yacc, double *d)
```

These functions return the interpolated value  $d = \partial z / \partial x$  for a given point  $(x, y)$ , using the interpolation object *interp*, data arrays *xa*, *ya*, and *za* and the accelerators *xacc* and *yacc*. When  $x$  is outside the range of *xa* or  $y$  is outside the range of *ya*, the error code *GSL\_EDOM* is returned.

```
double gsl_interp2d_eval_deriv_y(const gsl_interp2d *interp, const double xa[], const double ya[], const
                                double za[], const double x, const double y, gsl_interp_accel *xacc,
                                gsl_interp_accel *yacc)
```

```
int gsl_interp2d_eval_deriv_y_e(const gsl_interp2d *interp, const double xa[], const double ya[], const double
                                za[], const double x, const double y, gsl_interp_accel *xacc,
                                gsl_interp_accel *yacc, double *d)
```

These functions return the interpolated value  $d = \partial z / \partial y$  for a given point  $(x, y)$ , using the interpolation object *interp*, data arrays *xa*, *ya*, and *za* and the accelerators *xacc* and *yacc*. When  $x$  is outside the range of *xa* or  $y$  is outside the range of *ya*, the error code *GSL\_EDOM* is returned.

```
double gsl_interp2d_eval_deriv_xx(const gsl_interp2d *interp, const double xa[], const double ya[], const
                                   double za[], const double x, const double y, gsl_interp_accel *xacc,
                                   gsl_interp_accel *yacc)
```

```
int gsl_interp2d_eval_deriv_xx_e(const gsl_interp2d *interp, const double xa[], const double ya[], const
                                   double za[], const double x, const double y, gsl_interp_accel *xacc,
                                   gsl_interp_accel *yacc, double *d)
```

These functions return the interpolated value  $d = \partial^2 z / \partial x^2$  for a given point  $(x, y)$ , using the interpolation object *interp*, data arrays *xa*, *ya*, and *za* and the accelerators *xacc* and *yacc*. When  $x$  is outside the range of *xa* or  $y$  is outside the range of *ya*, the error code *GSL\_EDOM* is returned.

```
double gsl_interp2d_eval_deriv_yy(const gsl_interp2d *interp, const double xa[], const double ya[], const
double za[], const double x, const double y, gsl_interp_accel *xacc,
gsl_interp_accel *yacc)
```

```
int gsl_interp2d_eval_deriv_yy_e(const gsl_interp2d *interp, const double xa[], const double ya[], const
double za[], const double x, const double y, gsl_interp_accel *xacc,
gsl_interp_accel *yacc, double *d)
```

These functions return the interpolated value  $d = \partial^2 z / \partial y^2$  for a given point  $(x, y)$ , using the interpolation object *interp*, data arrays *xa*, *ya*, and *za* and the accelerators *xacc* and *yacc*. When *x* is outside the range of *xa* or *y* is outside the range of *ya*, the error code *GSL\_EDOM* is returned.

```
double gsl_interp2d_eval_deriv_xy(const gsl_interp2d *interp, const double xa[], const double ya[], const
double za[], const double x, const double y, gsl_interp_accel *xacc,
gsl_interp_accel *yacc)
```

```
int gsl_interp2d_eval_deriv_xy_e(const gsl_interp2d *interp, const double xa[], const double ya[], const
double za[], const double x, const double y, gsl_interp_accel *xacc,
gsl_interp_accel *yacc, double *d)
```

These functions return the interpolated value  $d = \partial^2 z / \partial x \partial y$  for a given point  $(x, y)$ , using the interpolation object *interp*, data arrays *xa*, *ya*, and *za* and the accelerators *xacc* and *yacc*. When *x* is outside the range of *xa* or *y* is outside the range of *ya*, the error code *GSL\_EDOM* is returned.

## 30.13 2D Higher-level Interface

The functions described in the previous sections required the user to supply pointers to the *x*, *y*, and *z* arrays on each call. The following functions are equivalent to the corresponding *gsl\_interp2d* functions but maintain a copy of this data in the *gsl\_spline2d* object. This removes the need to pass *xa*, *ya*, and *za* as arguments on each evaluation. These functions are defined in the header file *gsl\_spline2d.h*.

```
type gsl_spline2d
```

This workspace provides a higher level interface for the *gsl\_interp2d* object

```
gsl_spline2d *gsl_spline2d_alloc(const gsl_interp2d_type *T, size_t xsize, size_t ysize)
```

```
int gsl_spline2d_init(gsl_spline2d *spline, const double xa[], const double ya[], const double za[], size_t xsize,
size_t ysize)
```

```
void gsl_spline2d_free(gsl_spline2d *spline)
```

```
const char *gsl_spline2d_name(const gsl_spline2d *spline)
```

```
unsigned int gsl_spline2d_min_size(const gsl_spline2d *spline)
```

```
double gsl_spline2d_eval(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc,
gsl_interp_accel *yacc)
```

```
int gsl_spline2d_eval_e(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel *xacc,
gsl_interp_accel *yacc, double *z)
```

---

```

double gsl_spline2d_eval_extrap(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                *xacc, gsl_interp_accel *yacc)

int gsl_spline2d_eval_extrap_e(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                               *xacc, gsl_interp_accel *yacc, double *z)

double gsl_spline2d_eval_deriv_x(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                  *xacc, gsl_interp_accel *yacc)

int gsl_spline2d_eval_deriv_x_e(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                 *xacc, gsl_interp_accel *yacc, double *d)

double gsl_spline2d_eval_deriv_y(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                  *xacc, gsl_interp_accel *yacc)

int gsl_spline2d_eval_deriv_y_e(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                 *xacc, gsl_interp_accel *yacc, double *d)

double gsl_spline2d_eval_deriv_xx(const gsl_spline2d *spline, const double x, const double y,
                                   gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int gsl_spline2d_eval_deriv_xx_e(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                  *xacc, gsl_interp_accel *yacc, double *d)

double gsl_spline2d_eval_deriv_yy(const gsl_spline2d *spline, const double x, const double y,
                                   gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int gsl_spline2d_eval_deriv_yy_e(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                  *xacc, gsl_interp_accel *yacc, double *d)

double gsl_spline2d_eval_deriv_xy(const gsl_spline2d *spline, const double x, const double y,
                                   gsl_interp_accel *xacc, gsl_interp_accel *yacc)

int gsl_spline2d_eval_deriv_xy_e(const gsl_spline2d *spline, const double x, const double y, gsl_interp_accel
                                  *xacc, gsl_interp_accel *yacc, double *d)

int gsl_spline2d_set(const gsl_spline2d *spline, double za[], const size_t i, const size_t j, const double z)

double gsl_spline2d_get(const gsl_spline2d *spline, const double za[], const size_t i, const size_t j)
    This function returns the value  $z_{ij}$  for grid point  $(i, j)$  stored in the array za.

```

## 30.14 2D Interpolation Example programs

The following example performs bilinear interpolation on the unit square, using  $z$  values of (0, 1, 0.5, 1) going clockwise around the square.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_interp2d.h>
#include <gsl/gsl_spline2d.h>

int
main()
{
    const gsl_interp2d_type *T = gsl_interp2d_bilinear;
    const size_t N = 100;          /* number of points to interpolate */
    const double xa[] = { 0.0, 1.0 }; /* define unit square */
    const double ya[] = { 0.0, 1.0 };
    const size_t nx = sizeof(xa) / sizeof(double); /* x grid points */
    const size_t ny = sizeof(ya) / sizeof(double); /* y grid points */
    double *za = malloc(nx * ny * sizeof(double));
    gsl_spline2d *spline = gsl_spline2d_alloc(T, nx, ny);
    gsl_interp_accel *xacc = gsl_interp_accel_alloc();
    gsl_interp_accel *yacc = gsl_interp_accel_alloc();
    size_t i, j;

    /* set z grid values */
    gsl_spline2d_set(spline, za, 0, 0, 0.0);
    gsl_spline2d_set(spline, za, 0, 1, 1.0);
    gsl_spline2d_set(spline, za, 1, 1, 0.5);
    gsl_spline2d_set(spline, za, 1, 0, 1.0);

    /* initialize interpolation */
    gsl_spline2d_init(spline, xa, ya, za, nx, ny);

    /* interpolate N values in x and y and print out grid for plotting */
    for (i = 0; i < N; ++i)
    {
        double xi = i / (N - 1.0);

        for (j = 0; j < N; ++j)
        {
            double yj = j / (N - 1.0);
            double zij = gsl_spline2d_eval(spline, xi, yj, xacc, yacc);

            printf("%f %f %f\n", xi, yj, zij);
        }
        printf("\n");
    }

    gsl_spline2d_free(spline);
    gsl_interp_accel_free(xacc);
}
```

(continues on next page)

(continued from previous page)

```
gsl_interp_accel_free(yacc);  
free(za);  
  
return 0;  
}
```

The results of the interpolation are shown in Fig. 30.4, where the corners are labeled with their fixed  $z$  values.

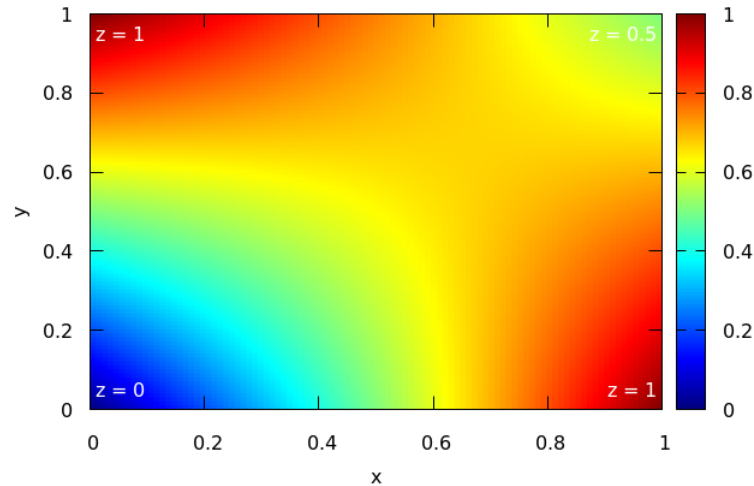


Fig. 30.4: 2D interpolation example

## 30.15 References and Further Reading

Descriptions of the interpolation algorithms and further references can be found in the following publications:

- C.W. Ueberhuber, *Numerical Computation (Volume 1), Chapter 9 “Interpolation”*, Springer (1997), ISBN 3-540-62058-3.
- D.M. Young, R.T. Gregory, *A Survey of Numerical Mathematics (Volume 1), Chapter 6.8*, Dover (1988), ISBN 0-486-65691-8.
- M. Steffen, *A simple method for monotonic interpolation in one dimension*, *Astron. Astrophys.* 239, 443-450, 1990.





## NUMERICAL DIFFERENTIATION

The functions described in this chapter compute numerical derivatives by finite differencing. An adaptive algorithm is used to find the best choice of finite difference and to estimate the error in the derivative. These functions are declared in the header file `gsl_deriv.h`.

### 31.1 Functions

int `gsl_deriv_central`(const *gsl\_function* \*f, double x, double h, double \*result, double \*abserr)

This function computes the numerical derivative of the function  $f$  at the point  $x$  using an adaptive central difference algorithm with a step-size of  $h$ . The derivative is returned in *result* and an estimate of its absolute error is returned in *abserr*.

The initial value of  $h$  is used to estimate an optimal step-size, based on the scaling of the truncation error and round-off error in the derivative calculation. The derivative is computed using a 5-point rule for equally spaced abscissae at  $x - h$ ,  $x - h/2$ ,  $x$ ,  $x + h/2$ ,  $x + h$ , with an error estimate taken from the difference between the 5-point rule and the corresponding 3-point rule  $x - h$ ,  $x$ ,  $x + h$ . Note that the value of the function at  $x$  does not contribute to the derivative calculation, so only 4-points are actually used.

int `gsl_deriv_forward`(const *gsl\_function* \*f, double x, double h, double \*result, double \*abserr)

This function computes the numerical derivative of the function  $f$  at the point  $x$  using an adaptive forward difference algorithm with a step-size of  $h$ . The function is evaluated only at points greater than  $x$ , and never at  $x$  itself. The derivative is returned in *result* and an estimate of its absolute error is returned in *abserr*. This function should be used if  $f(x)$  has a discontinuity at  $x$ , or is undefined for values less than  $x$ .

The initial value of  $h$  is used to estimate an optimal step-size, based on the scaling of the truncation error and round-off error in the derivative calculation. The derivative at  $x$  is computed using an “open” 4-point rule for equally spaced abscissae at  $x + h/4$ ,  $x + h/2$ ,  $x + 3h/4$ ,  $x + h$ , with an error estimate taken from the difference between the 4-point rule and the corresponding 2-point rule  $x + h/2$ ,  $x + h$ .

int `gsl_deriv_backward`(const *gsl\_function* \*f, double x, double h, double \*result, double \*abserr)

This function computes the numerical derivative of the function  $f$  at the point  $x$  using an adaptive backward difference algorithm with a step-size of  $h$ . The function is evaluated only at points less than  $x$ , and never at  $x$  itself. The derivative is returned in *result* and an estimate of its absolute error is returned in *abserr*. This function should be used if  $f(x)$  has a discontinuity at  $x$ , or is undefined for values greater than  $x$ .

This function is equivalent to calling `gsl_deriv_forward()` with a negative step-size.

## 31.2 Examples

The following code estimates the derivative of the function  $f(x) = x^{3/2}$  at  $x = 2$  and at  $x = 0$ . The function  $f(x)$  is undefined for  $x < 0$  so the derivative at  $x = 0$  is computed using `gsl_deriv_forward()`.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_deriv.h>

double f (double x, void * params)
{
    (void)(params); /* avoid unused parameter warning */
    return pow (x, 1.5);
}

int
main (void)
{
    gsl_function F;
    double result, abserr;

    F.function = &f;
    F.params = 0;

    printf ("f(x) = x^(3/2)\n");

    gsl_deriv_central (&F, 2.0, 1e-8, &result, &abserr);
    printf ("x = 2.0\n");
    printf ("f'(x) = %.10f +/- %.10f\n", result, abserr);
    printf ("exact = %.10f\n\n", 1.5 * sqrt(2.0));

    gsl_deriv_forward (&F, 0.0, 1e-8, &result, &abserr);
    printf ("x = 0.0\n");
    printf ("f'(x) = %.10f +/- %.10f\n", result, abserr);
    printf ("exact = %.10f\n", 0.0);

    return 0;
}
```

Here is the output of the program,

```
f(x) = x^(3/2)
x = 2.0
f'(x) = 2.1213203120 +/- 0.00000005006
exact = 2.1213203436

x = 0.0
f'(x) = 0.00000000160 +/- 0.00000000339
exact = 0.00000000000
```

## 31.3 References and Further Reading

The algorithms used by these functions are described in the following sources:

- Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 25.3.4, and Table 25.5 (Coefficients for Differentiation).
- S.D. Conte and Carl de Boor, *Elementary Numerical Analysis: An Algorithmic Approach*, McGraw-Hill, 1972.



## CHEBYSHEV APPROXIMATIONS

This chapter describes routines for computing Chebyshev approximations to univariate functions. A Chebyshev approximation is a truncation of the series  $f(x) = \sum c_n T_n(x)$ , where the Chebyshev polynomials  $T_n(x) = \cos(n \arccos x)$  provide an orthogonal basis of polynomials on the interval  $[-1, 1]$  with the weight function  $1/\sqrt{1-x^2}$ . The first few Chebyshev polynomials are,  $T_0(x) = 1$ ,  $T_1(x) = x$ ,  $T_2(x) = 2x^2 - 1$ . For further information see Abramowitz & Stegun, Chapter 22.

The functions described in this chapter are declared in the header file `gsl_chebyshev.h`.

### 32.1 Definitions

type `gsl_cheb_series`

A Chebyshev series is stored using the following structure:

```
typedef struct
{
  double * c; /* coefficients c[0] .. c[order] */
  int order; /* order of expansion */
  double a; /* lower interval point */
  double b; /* upper interval point */
  ...
} gsl_cheb_series
```

The approximation is made over the range  $[a, b]$  using `order + 1` terms, including the coefficient  $c[0]$ . The series is computed using the following convention,

$$f(x) = \frac{c_0}{2} + \sum_{n=1} c_n T_n(x)$$

which is needed when accessing the coefficients directly.

### 32.2 Creation and Calculation of Chebyshev Series

`gsl_cheb_series *gsl_cheb_alloc(const size_t n)`

This function allocates space for a Chebyshev series of order  $n$  and returns a pointer to a new `gsl_cheb_series` struct.

void `gsl_cheb_free(gsl_cheb_series *cs)`

This function frees a previously allocated Chebyshev series `cs`.

int **gsl\_cheb\_init**(*gsl\_cheb\_series* \*cs, const *gsl\_function* \*f, const double a, const double b)

This function computes the Chebyshev approximation *cs* for the function *f* over the range  $(a, b)$  to the previously specified order. The computation of the Chebyshev approximation is an  $O(n^2)$  process, and requires  $n$  function evaluations.

## 32.3 Auxiliary Functions

The following functions provide information about an existing Chebyshev series.

size\_t **gsl\_cheb\_order**(const *gsl\_cheb\_series* \*cs)

This function returns the order of Chebyshev series *cs*.

size\_t **gsl\_cheb\_size**(const *gsl\_cheb\_series* \*cs)

double \***gsl\_cheb\_coeffs**(const *gsl\_cheb\_series* \*cs)

These functions return the size of the Chebyshev coefficient array *c*[] and a pointer to its location in memory for the Chebyshev series *cs*.

## 32.4 Chebyshev Series Evaluation

double **gsl\_cheb\_eval**(const *gsl\_cheb\_series* \*cs, double x)

This function evaluates the Chebyshev series *cs* at a given point *x*.

int **gsl\_cheb\_eval\_err**(const *gsl\_cheb\_series* \*cs, const double x, double \*result, double \*abserr)

This function computes the Chebyshev series *cs* at a given point *x*, estimating both the series *result* and its absolute error *abserr*. The error estimate is made from the first neglected term in the series.

double **gsl\_cheb\_eval\_n**(const *gsl\_cheb\_series* \*cs, size\_t order, double x)

This function evaluates the Chebyshev series *cs* at a given point *x*, to (at most) the given order *order*.

int **gsl\_cheb\_eval\_n\_err**(const *gsl\_cheb\_series* \*cs, const size\_t order, const double x, double \*result, double \*abserr)

This function evaluates a Chebyshev series *cs* at a given point *x*, estimating both the series *result* and its absolute error *abserr*, to (at most) the given order *order*. The error estimate is made from the first neglected term in the series.

## 32.5 Derivatives and Integrals

The following functions allow a Chebyshev series to be differentiated or integrated, producing a new Chebyshev series. Note that the error estimate produced by evaluating the derivative series will be underestimated due to the contribution of higher order terms being neglected.

int **gsl\_cheb\_calc\_deriv**(*gsl\_cheb\_series* \*deriv, const *gsl\_cheb\_series* \*cs)

This function computes the derivative of the series *cs*, storing the derivative coefficients in the previously allocated *deriv*. The two series *cs* and *deriv* must have been allocated with the same order.

int **gsl\_cheb\_calc\_integ**(*gsl\_cheb\_series* \*integ, const *gsl\_cheb\_series* \*cs)

This function computes the integral of the series *cs*, storing the integral coefficients in the previously allocated *integ*. The two series *cs* and *integ* must have been allocated with the same order. The lower limit of the integration is taken to be the left hand end of the range *a*.

## 32.6 Examples

The following example program computes Chebyshev approximations to a step function. This is an extremely difficult approximation to make, due to the discontinuity, and was chosen as an example where approximation error is visible. For smooth functions the Chebyshev approximation converges extremely rapidly and errors would not be visible.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_chebyshev.h>

double
f (double x, void *p)
{
    (void)(p); /* avoid unused parameter warning */

    if (x < 0.5)
        return 0.25;
    else
        return 0.75;
}

int
main (void)
{
    int i, n = 10000;

    gsl_cheb_series *cs = gsl_cheb_alloc (40);

    gsl_function F;

    F.function = f;
    F.params = 0;

    gsl_cheb_init (cs, &F, 0.0, 1.0);

    for (i = 0; i < n; i++)
    {
        double x = i / (double)n;
        double r10 = gsl_cheb_eval_n (cs, 10, x);
        double r40 = gsl_cheb_eval (cs, x);
        printf ("%g %g %g %g\n",
                x, GSL_FN_EVAL (&F, x), r10, r40);
    }

    gsl_cheb_free (cs);

    return 0;
}
```

Fig. 32.1 shows output from the program with the original function, 10-th order approximation and 40-th order approximation, all sampled at intervals of 0.001 in  $x$ .

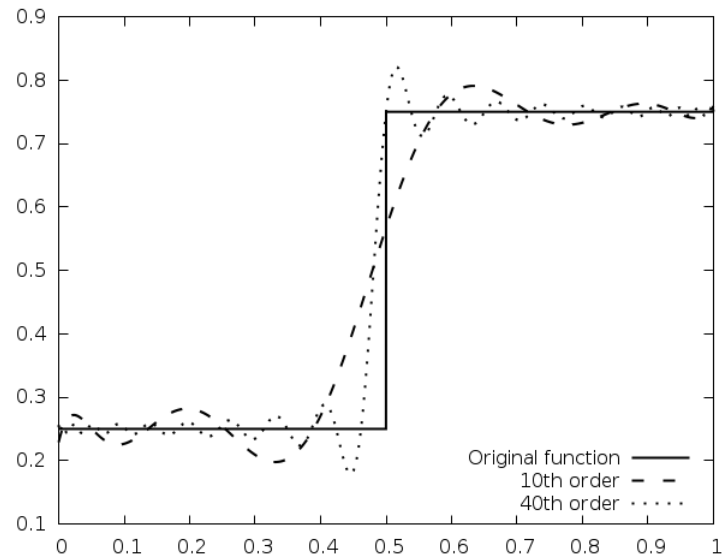


Fig. 32.1: Chebyshev approximations to a step function

## 32.7 References and Further Reading

The following paper describes the use of Chebyshev series,

- R. Broucke, “Ten Subroutines for the Manipulation of Chebyshev Series [C1] (Algorithm 446)”. *Communications of the ACM* 16(4), 254–256 (1973)



## SERIES ACCELERATION

The functions described in this chapter accelerate the convergence of a series using the Levin  $u$ -transform. This method takes a small number of terms from the start of a series and uses a systematic approximation to compute an extrapolated value and an estimate of its error. The  $u$ -transform works for both convergent and divergent series, including asymptotic series.

These functions are declared in the header file `gsl_sum.h`.

### 33.1 Acceleration functions

The following functions compute the full Levin  $u$ -transform of a series with its error estimate. The error estimate is computed by propagating rounding errors from each term through to the final extrapolation.

These functions are intended for summing analytic series where each term is known to high accuracy, and the rounding errors are assumed to originate from finite precision. They are taken to be relative errors of order `GSL_DBL_EPSILON` for each term.

The calculation of the error in the extrapolated value is an  $O(N^2)$  process, which is expensive in time and memory. A faster but less reliable method which estimates the error from the convergence of the extrapolated value is described in the next section. For the method described here a full table of intermediate values and derivatives through to  $O(N)$  must be computed and stored, but this does give a reliable error estimate.

type **gsl\_sum\_levin\_u\_workspace**

Workspace for a Leven  $u$ -transform.

*gsl\_sum\_levin\_u\_workspace* \***gsl\_sum\_levin\_u\_alloc**(size\_t n)

This function allocates a workspace for a Levin  $u$ -transform of  $n$  terms. The size of the workspace is  $O(2n^2+3n)$ .

void **gsl\_sum\_levin\_u\_free**(*gsl\_sum\_levin\_u\_workspace* \*w)

This function frees the memory associated with the workspace  $w$ .

int **gsl\_sum\_levin\_u\_accel**(const double \*array, size\_t array\_size, *gsl\_sum\_levin\_u\_workspace* \*w, double \*sum\_accel, double \*abserr)

This function takes the terms of a series in *array* of size *array\_size* and computes the extrapolated limit of the series using a Levin  $u$ -transform. Additional working space must be provided in  $w$ . The extrapolated sum is stored in *sum\_accel*, with an estimate of the absolute error stored in *abserr*. The actual term-by-term sum is returned in `w->sum_plain`. The algorithm calculates the truncation error (the difference between two successive extrapolations) and round-off error (propagated from the individual terms) to choose an optimal number of terms for the extrapolation. All the terms of the series passed in through *array* should be non-zero.

## 33.2 Acceleration functions without error estimation

The functions described in this section compute the Levin  $u$ -transform of series and attempt to estimate the error from the “truncation error” in the extrapolation, the difference between the final two approximations. Using this method avoids the need to compute an intermediate table of derivatives because the error is estimated from the behavior of the extrapolated value itself. Consequently this algorithm is an  $O(N)$  process and only requires  $O(N)$  terms of storage. If the series converges sufficiently fast then this procedure can be acceptable. It is appropriate to use this method when there is a need to compute many extrapolations of series with similar convergence properties at high-speed. For example, when numerically integrating a function defined by a parameterized series where the parameter varies only slightly. A reliable error estimate should be computed first using the full algorithm described above in order to verify the consistency of the results.

type `gsl_sum_levin_utrunc_workspace`

Workspace for a Levin  $u$ -transform without error estimation

`gsl_sum_levin_utrunc_workspace *gsl_sum_levin_utrunc_alloc(size_t n)`

This function allocates a workspace for a Levin  $u$ -transform of  $n$  terms, without error estimation. The size of the workspace is  $O(3n)$ .

void `gsl_sum_levin_utrunc_free(gsl_sum_levin_utrunc_workspace *w)`

This function frees the memory associated with the workspace  $w$ .

int `gsl_sum_levin_utrunc_accel(const double *array, size_t array_size, gsl_sum_levin_utrunc_workspace *w, double *sum_accel, double *abserr_trunc)`

This function takes the terms of a series in `array` of size `array_size` and computes the extrapolated limit of the series using a Levin  $u$ -transform. Additional working space must be provided in `w`. The extrapolated sum is stored in `sum_accel`. The actual term-by-term sum is returned in `w->sum_plain`. The algorithm terminates when the difference between two successive extrapolations reaches a minimum or is sufficiently small. The difference between these two values is used as estimate of the error and is stored in `abserr_trunc`. To improve the reliability of the algorithm the extrapolated values are replaced by moving averages when calculating the truncation error, smoothing out any fluctuations.

## 33.3 Examples

The following code calculates an estimate of  $\zeta(2) = \pi^2/6$  using the series,

$$\zeta(2) = 1 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$$

After  $N$  terms the error in the sum is  $O(1/N)$ , making direct summation of the series converge slowly.

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_sum.h>

#define N 20

int
main (void)
{
    double t[N];
    double sum_accel, err;
    double sum = 0;
    int n;
```

(continues on next page)

(continued from previous page)

```

gsl_sum_levin_u_workspace * w
= gsl_sum_levin_u_alloc (N);

const double zeta_2 = M_PI * M_PI / 6.0;

/* terms for zeta(2) = \sum_{n=1}^{\infty} 1/n^2 */
for (n = 0; n < N; n++)
{
    double np1 = n + 1.0;
    t[n] = 1.0 / (np1 * np1);
    sum += t[n];
}

gsl_sum_levin_u_accel (t, N, w, &sum_accel, &err);

printf ("term-by-term sum = %.16f using %d terms\n",
        sum, N);

printf ("term-by-term sum = %.16f using %zu terms\n",
        w->sum_plain, w->terms_used);

printf ("exact value      = %.16f\n", zeta_2);
printf ("accelerated sum    = %.16f using %zu terms\n",
        sum_accel, w->terms_used);

printf ("estimated error    = %.16f\n", err);
printf ("actual error      = %.16f\n",
        sum_accel - zeta_2);

gsl_sum_levin_u_free (w);
return 0;
}

```

The output below shows that the Levin  $u$ -transform is able to obtain an estimate of the sum to 1 part in  $10^{10}$  using the first eleven terms of the series. The error estimate returned by the function is also accurate, giving the correct number of significant digits.

```

term-by-term sum = 1.5961632439130233 using 20 terms
term-by-term sum = 1.5759958390005426 using 13 terms
exact value      = 1.6449340668482264
accelerated sum  = 1.6449340669228176 using 13 terms
estimated error  = 0.00000000000888360
actual error     = 0.0000000000745912

```

Note that a direct summation of this series would require  $10^{10}$  terms to achieve the same precision as the accelerated sum does in 13 terms.

## 33.4 References and Further Reading

The algorithms used by these functions are described in the following papers,

- T. Fessler, W.F. Ford, D.A. Smith, HURRY: An acceleration algorithm for scalar sequences and series *ACM Transactions on Mathematical Software*, 9(3):346–354, 1983. and Algorithm 602 9(3):355–357, 1983.

The theory of the  $u$ -transform was presented by Levin,

- D. Levin, Development of Non-Linear Transformations for Improving Convergence of Sequences, *Intern.: J.: Computer Math.* B3:371–388, 1973.

A review paper on the Levin Transform is available online,

- Herbert H. H. Homeier, Scalar Levin-Type Sequence Transformations, <http://arxiv.org/abs/math/0005209>

## WAVELET TRANSFORMS

This chapter describes functions for performing Discrete Wavelet Transforms (DWTs). The library includes wavelets for real data in both one and two dimensions. The wavelet functions are declared in the header files `gsl_wavelet.h` and `gsl_wavelet2d.h`.

### 34.1 Definitions

The continuous wavelet transform and its inverse are defined by the relations,

$$w(s, \tau) = \int_{-\infty}^{\infty} f(t) * \psi_{s,\tau}^*(t) dt$$

and,

$$f(t) = \int_0^{\infty} ds \int_{-\infty}^{\infty} w(s, \tau) * \psi_{s,\tau}(t) d\tau$$

where the basis functions  $\psi_{s,\tau}$  are obtained by scaling and translation from a single function, referred to as the *mother wavelet*.

The discrete version of the wavelet transform acts on equally-spaced samples, with fixed scaling and translation steps ( $s, \tau$ ). The frequency and time axes are sampled *dyadically* on scales of  $2^j$  through a level parameter  $j$ .

The resulting family of functions  $\{\psi_{j,n}\}$  constitutes an orthonormal basis for square-integrable signals. The discrete wavelet transform is an  $O(N)$  algorithm, and is also referred to as the *fast wavelet transform*.

### 34.2 Initialization

type `gsl_wavelet`

This structure contains the filter coefficients defining the wavelet and any associated offset parameters.

`gsl_wavelet *gsl_wavelet_alloc(const gsl_wavelet_type *T, size_t k)`

This function allocates and initializes a wavelet object of type  $T$ . The parameter  $k$  selects the specific member of the wavelet family. A null pointer is returned if insufficient memory is available or if a unsupported member is selected.

The following wavelet types are implemented:

type `gsl_wavelet_type`

`gsl_wavelet_type *gsl_wavelet_daubechies`

*gsl\_wavelet\_type* \***gsl\_wavelet\_daubechies\_centered**

This is the Daubechies wavelet family of maximum phase with  $k/2$  vanishing moments. The implemented wavelets are  $k = 4, 6, \dots, 20$ , with  $k$  even.

*gsl\_wavelet\_type* \***gsl\_wavelet\_haar**

*gsl\_wavelet\_type* \***gsl\_wavelet\_haar\_centered**

This is the Haar wavelet. The only valid choice of  $k$  for the Haar wavelet is  $k = 2$ .

*gsl\_wavelet\_type* \***gsl\_wavelet\_bspline**

*gsl\_wavelet\_type* \***gsl\_wavelet\_bspline\_centered**

This is the biorthogonal B-spline wavelet family of order  $(i, j)$ . The implemented values of  $k = 100 * i + j$  are 103, 105, 202, 204, 206, 208, 301, 303, 305 307, 309.

The centered forms of the wavelets align the coefficients of the various sub-bands on edges. Thus the resulting visualization of the coefficients of the wavelet transform in the phase plane is easier to understand.

const char \***gsl\_wavelet\_name**(const *gsl\_wavelet* \*w)

This function returns a pointer to the name of the wavelet family for  $w$ .

void **gsl\_wavelet\_free**(*gsl\_wavelet* \*w)

This function frees the wavelet object  $w$ .

type **gsl\_wavelet\_workspace**

This structure contains scratch space of the same size as the input data and is used to hold intermediate results during the transform.

*gsl\_wavelet\_workspace* \***gsl\_wavelet\_workspace\_alloc**(size\_t n)

This function allocates a workspace for the discrete wavelet transform. To perform a one-dimensional transform on  $n$  elements, a workspace of size  $n$  must be provided. For two-dimensional transforms of  $n$ -by- $n$  matrices it is sufficient to allocate a workspace of size  $n$ , since the transform operates on individual rows and columns. A null pointer is returned if insufficient memory is available.

void **gsl\_wavelet\_workspace\_free**(*gsl\_wavelet\_workspace* \*work)

This function frees the allocated workspace  $work$ .

## 34.3 Transform Functions

This sections describes the actual functions performing the discrete wavelet transform. Note that the transforms use periodic boundary conditions. If the signal is not periodic in the sample length then spurious coefficients will appear at the beginning and end of each level of the transform.

### 34.3.1 Wavelet transforms in one dimension

int **gsl\_wavelet\_transform**(const *gsl\_wavelet* \*w, double \*data, size\_t stride, size\_t n, *gsl\_wavelet\_direction* dir, *gsl\_wavelet\_workspace* \*work)

int **gsl\_wavelet\_transform\_forward**(const *gsl\_wavelet* \*w, double \*data, size\_t stride, size\_t n, *gsl\_wavelet\_workspace* \*work)

int **gsl\_wavelet\_transform\_inverse**(const *gsl\_wavelet* \*w, double \*data, size\_t stride, size\_t n, *gsl\_wavelet\_workspace* \*work)

These functions compute in-place forward and inverse discrete wavelet transforms of length  $n$  with stride  $stride$  on the array  $data$ . The length of the transform  $n$  is restricted to powers of two. For the `transform` version of the function the argument `dir` can be either `forward` (+1) or `backward` (-1). A workspace  $work$  of length  $n$  must be provided.

For the forward transform, the elements of the original array are replaced by the discrete wavelet transform  $f_i \rightarrow w_{j,k}$  in a packed triangular storage layout, where  $j$  is the index of the level  $j = 0 \dots J - 1$  and  $k$  is the index of the coefficient within each level,  $k = 0 \dots 2^j - 1$ . The total number of levels is  $J = \log_2(n)$ . The output data has the following form,

$$(s_{-1,0}, d_{0,0}, d_{1,0}, d_{1,1}, d_{2,0}, \dots, d_{j,k}, \dots, d_{J-1,2^{J-1}-1})$$

where the first element is the smoothing coefficient  $s_{-1,0}$ , followed by the detail coefficients  $d_{j,k}$  for each level  $j$ . The backward transform inverts these coefficients to obtain the original data.

These functions return a status of `GSL_SUCCESS` upon successful completion. `GSL_EINVAL` is returned if  $n$  is not an integer power of 2 or if insufficient workspace is provided.

### 34.3.2 Wavelet transforms in two dimension

The library provides functions to perform two-dimensional discrete wavelet transforms on square matrices. The matrix dimensions must be an integer power of two. There are two possible orderings of the rows and columns in the two-dimensional wavelet transform, referred to as the “standard” and “non-standard” forms.

The “standard” transform performs a complete discrete wavelet transform on the rows of the matrix, followed by a separate complete discrete wavelet transform on the columns of the resulting row-transformed matrix. This procedure uses the same ordering as a two-dimensional Fourier transform.

The “non-standard” transform is performed in interleaved passes on the rows and columns of the matrix for each level of the transform. The first level of the transform is applied to the matrix rows, and then to the matrix columns. This procedure is then repeated across the rows and columns of the data for the subsequent levels of the transform, until the full discrete wavelet transform is complete. The non-standard form of the discrete wavelet transform is typically used in image analysis.

The functions described in this section are declared in the header file `gsl_wavelet2d.h`.

```
int gsl_wavelet2d_transform(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2,
                           gsl_wavelet_direction dir, gsl_wavelet_workspace *work)
```

```
int gsl_wavelet2d_transform_forward(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2,
                                     gsl_wavelet_workspace *work)
```

```
int gsl_wavelet2d_transform_inverse(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2,
                                     gsl_wavelet_workspace *work)
```

These functions compute two-dimensional in-place forward and inverse discrete wavelet transforms in standard form on the array `data` stored in row-major form with dimensions `size1` and `size2` and physical row length `tda`. The dimensions must be equal (square matrix) and are restricted to powers of two. For the `transform` version of the function the argument `dir` can be either `forward` (+1) or `backward` (-1). A workspace `work` of the appropriate size must be provided. On exit, the appropriate elements of the array `data` are replaced by their two-dimensional wavelet transform.

The functions return a status of `GSL_SUCCESS` upon successful completion. `GSL_EINVAL` is returned if `size1` and `size2` are not equal and integer powers of 2, or if insufficient workspace is provided.

```
int gsl_wavelet2d_transform_matrix(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_direction dir,
                                   gsl_wavelet_workspace *work)
```

```
int gsl_wavelet2d_transform_matrix_forward(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_workspace
                                           *work)
```

```
int gsl_wavelet2d_transform_matrix_inverse(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_workspace
                                           *work)
```

These functions compute the two-dimensional in-place wavelet transform on a matrix *m*.

```
int gsl_wavelet2d_nstransform(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t size2,
                               gsl_wavelet_direction dir, gsl_wavelet_workspace *work)
```

```
int gsl_wavelet2d_nstransform_forward(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t
                                       size2, gsl_wavelet_workspace *work)
```

```
int gsl_wavelet2d_nstransform_inverse(const gsl_wavelet *w, double *data, size_t tda, size_t size1, size_t
                                       size2, gsl_wavelet_workspace *work)
```

These functions compute the two-dimensional wavelet transform in non-standard form.

```
int gsl_wavelet2d_nstransform_matrix(const gsl_wavelet *w, gsl_matrix *m, gsl_wavelet_direction dir,
                                       gsl_wavelet_workspace *work)
```

```
int gsl_wavelet2d_nstransform_matrix_forward(const gsl_wavelet *w, gsl_matrix *m,
                                              gsl_wavelet_workspace *work)
```

```
int gsl_wavelet2d_nstransform_matrix_inverse(const gsl_wavelet *w, gsl_matrix *m,
                                              gsl_wavelet_workspace *work)
```

These functions compute the non-standard form of the two-dimensional in-place wavelet transform on a matrix *m*.

## 34.4 Examples

The following program demonstrates the use of the one-dimensional wavelet transform functions. It computes an approximation to an input signal (of length 256) using the 20 largest components of the wavelet transform, while setting the others to zero.

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_wavelet.h>

int
main (int argc, char **argv)
{
    (void)(argc); /* avoid unused parameter warning */
    int i, n = 256, nc = 20;
    double *orig_data = malloc (n * sizeof (double));
    double *data = malloc (n * sizeof (double));
    double *abscoeff = malloc (n * sizeof (double));
    size_t *p = malloc (n * sizeof (size_t));

    FILE * f;
    gsl_wavelet *w;
    gsl_wavelet_workspace *work;

    w = gsl_wavelet_alloc (gsl_wavelet_daubechies, 4);
    work = gsl_wavelet_workspace_alloc (n);
```

(continues on next page)



(continued from previous page)

```

f = fopen (argv[1], "r");
for (i = 0; i < n; i++)
  {
    fscanf (f, "%lg", &orig_data[i]);
    data[i] = orig_data[i];
  }
fclose (f);

gsl_wavelet_transform_forward (w, data, 1, n, work);

for (i = 0; i < n; i++)
  {
    abscoeff[i] = fabs (data[i]);
  }

gsl_sort_index (p, abscoeff, 1, n);

for (i = 0; (i + nc) < n; i++)
  data[p[i]] = 0;

gsl_wavelet_transform_inverse (w, data, 1, n, work);

for (i = 0; i < n; i++)
  {
    printf ("%g %g\n", orig_data[i], data[i]);
  }

gsl_wavelet_free (w);
gsl_wavelet_workspace_free (work);

free (data);
free (orig_data);
free (abscoeff);
free (p);
return 0;
}

```

The output can be used with the GNU plotutils graph program:

```

$ ./a.out ecg.dat > dwt.txt
$ graph -T ps -x 0 256 32 -h 0.3 -a dwt.txt > dwt.ps

```

Fig. 34.1 shows an original and compressed version of a sample ECG recording from the MIT-BIH Arrhythmia Database, part of the PhysioNet archive of public-domain of medical datasets.

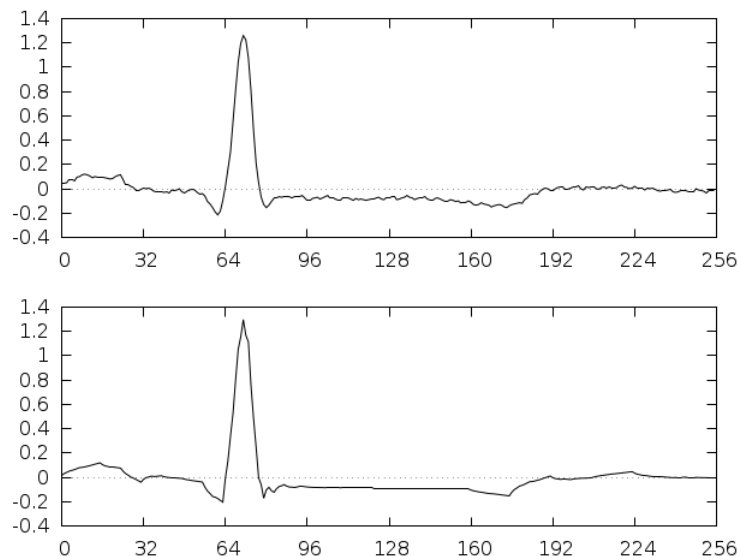


Fig. 34.1: Original (upper) and wavelet-compressed (lower) ECG signals, using the 20 largest components of the Daubechies(4) discrete wavelet transform.

## 34.5 References and Further Reading

The mathematical background to wavelet transforms is covered in the original lectures by Daubechies,

- Ingrid Daubechies. *Ten Lectures on Wavelets*. *CBMS-NSF Regional Conference Series in Applied Mathematics* (1992), SIAM, ISBN 0898712742.

An easy to read introduction to the subject with an emphasis on the application of the wavelet transform in various branches of science is,

- Paul S. Addison. *The Illustrated Wavelet Transform Handbook*. Institute of Physics Publishing (2002), ISBN 0750306920.

For extensive coverage of signal analysis by wavelets, wavelet packets and local cosine bases see,

- S. G. Mallat. *A wavelet tour of signal processing* (Second edition). Academic Press (1999), ISBN 012466606X.

The concept of multiresolution analysis underlying the wavelet transform is described in,

- S. G. Mallat. Multiresolution Approximations and Wavelet Orthonormal Bases of  $L^2(\mathbb{R})$ . *Transactions of the American Mathematical Society*, 315(1), 1989, 69–87.
- S. G. Mallat. A Theory for Multiresolution Signal Decomposition—The Wavelet Representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11, 1989, 674–693.

The coefficients for the individual wavelet families implemented by the library can be found in the following papers,

- I. Daubechies. Orthonormal Bases of Compactly Supported Wavelets. *Communications on Pure and Applied Mathematics*, 41 (1988) 909–996.
- A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal Bases of Compactly Supported Wavelets. *Communications on Pure and Applied Mathematics*, 45 (1992) 485–560.

The PhysioNet archive of physiological datasets can be found online at <http://www.physionet.org/> and is described in the following paper,

- Goldberger et al. PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals. *Circulation* 101(23):e215-e220 2000.



## DISCRETE HANKEL TRANSFORMS

This chapter describes functions for performing Discrete Hankel Transforms (DHTs). The functions are declared in the header file `gsl_dht.h`.

### 35.1 Definitions

The discrete Hankel transform acts on a vector of sampled data, where the samples are assumed to have been taken at points related to the zeros of a Bessel function of fixed order; compare this to the case of the discrete Fourier transform, where samples are taken at points related to the zeroes of the sine or cosine function.

Starting with its definition, the Hankel transform (or Bessel transform) of order  $\nu$  of a function  $f$  with  $\nu > -1/2$  is defined as (see Johnson, 1987 and Lemoine, 1994)

$$F_\nu(u) = \int_0^\infty f(t)J_\nu(ut)tdt$$

If the integral exists,  $F_\nu$  is called the Hankel transformation of  $f$ . The reverse transform is given by

$$f(t) = \int_0^\infty F_\nu(u)J_\nu(ut)udu$$

where  $\int_0^\infty f(t)t^{1/2}dt$  must exist and be absolutely convergent, and where  $f(t)$  satisfies Dirichlet's conditions (of limited total fluctuations) in the interval  $[0, \infty]$ .

Now the discrete Hankel transform works on a discrete function  $f$ , which is sampled on points  $n = 1 \dots M$  located at positions  $t_n = (j_{\nu,n}/j_{\nu,M})X$  in real space and at  $u_n = j_{\nu,n}/X$  in reciprocal space. Here,  $j_{\nu,m}$  are the  $m$ -th zeros of the Bessel function  $J_\nu(x)$  arranged in ascending order. Moreover, the discrete functions are assumed to be band limited, so  $f(t_n) = 0$  and  $F(u_n) = 0$  for  $n > M$ . Accordingly, the function  $f$  is defined on the interval  $[0, X]$ .

Following the work of Johnson, 1987 and Lemoine, 1994, the discrete Hankel transform is given by

$$F_\nu(u_m) = \frac{2X^2}{j_{\nu,M}^2} \sum_{k=1}^{M-1} f\left(\frac{j_{\nu,k}X}{j_{\nu,M}}\right) \frac{J_\nu(j_{\nu,m}j_{\nu,k}/j_{\nu,M})}{J_{\nu+1}(j_{\nu,k})^2}.$$

It is this discrete expression which defines the discrete Hankel transform calculated by GSL. In GSL, forward and backward transforms are defined equally and calculate  $F_\nu(u_m)$ . Following Johnson, the backward transform reads

$$f(t_k) = \frac{2}{X^2} \sum_{m=1}^{M-1} F\left(\frac{j_{\nu,m}}{X}\right) \frac{J_\nu(j_{\nu,m}j_{\nu,k}/j_{\nu,M})}{J_{\nu+1}(j_{\nu,m})^2}.$$

Obviously, using the forward transform instead of the backward transform gives an additional factor  $X^4/j_{\nu,M}^2 = t_m^2/u_m^2$ .

The kernel in the summation above defines the matrix of the  $\nu$ -Hankel transform of size  $M - 1$ . The coefficients of this matrix, being dependent on  $\nu$  and  $M$ , must be precomputed and stored; the `gsl_dht` object encapsulates this data. The allocation function `gsl_dht_alloc()` returns a `gsl_dht` object which must be properly initialized with `gsl_dht_init()` before it can be used to perform transforms on data sample vectors, for fixed  $\nu$  and  $M$ , using the `gsl_dht_apply()` function. The implementation allows to define the length  $X$  of the fundamental interval, for convenience, while discrete Hankel transforms are often defined on the unit interval instead of  $[0, X]$ .

Notice that by assumption  $f(t)$  vanishes at the endpoints of the interval, consistent with the inversion formula and the sampling formula given above. Therefore, this transform corresponds to an orthogonal expansion in eigenfunctions of the Dirichlet problem for the Bessel differential equation.

## 35.2 Functions

type `gsl_dht`

Workspace for computing discrete Hankel transforms

`gsl_dht *gsl_dht_alloc(size_t size)`

This function allocates a Discrete Hankel transform object of size `size`.

int `gsl_dht_init(gsl_dht *t, double nu, double xmax)`

This function initializes the transform `t` for the given values of `nu` and `xmax`.

`gsl_dht *gsl_dht_new(size_t size, double nu, double xmax)`

This function allocates a Discrete Hankel transform object of size `size` and initializes it for the given values of `nu` and `xmax`.

void `gsl_dht_free(gsl_dht *t)`

This function frees the transform `t`.

int `gsl_dht_apply(const gsl_dht *t, double *f_in, double *f_out)`

This function applies the transform `t` to the array `f_in` whose size is equal to the size of the transform. The result is stored in the array `f_out` which must be of the same length.

Applying this function to its output gives the original data multiplied by  $(X^2/j_{\nu,M})^2$ , up to numerical errors.

double `gsl_dht_x_sample(const gsl_dht *t, int n)`

This function returns the value of the  $n$ -th sample point in the unit interval,  $(j_{\nu,n+1}/j_{\nu,M})X$ . These are the points where the function  $f(t)$  is assumed to be sampled.

double `gsl_dht_k_sample(const gsl_dht *t, int n)`

This function returns the value of the  $n$ -th sample point in “k-space”,  $j_{\nu,n+1}/X$ .

## 35.3 References and Further Reading

The algorithms used by these functions are described in the following papers,

- H. Fisk Johnson, Comp.: Phys.: Comm.: 43, 181 (1987).
- D. Lemoine, J. Chem.: Phys.: 101, 3936 (1994).

## ONE DIMENSIONAL ROOT-FINDING

This chapter describes routines for finding roots of arbitrary one-dimensional functions. The library provides low level components for a variety of iterative solvers and convergence tests. These can be combined by the user to achieve the desired solution, with full access to the intermediate steps of the iteration. Each class of methods uses the same framework, so that you can switch between solvers at runtime without needing to recompile your program. Each instance of a solver keeps track of its own state, allowing the solvers to be used in multi-threaded programs.

The header file `gsl_roots.h` contains prototypes for the root finding functions and related declarations.

### 36.1 Overview

One-dimensional root finding algorithms can be divided into two classes, *root bracketing* and *root polishing*. Algorithms which proceed by bracketing a root are guaranteed to converge. Bracketing algorithms begin with a bounded region known to contain a root. The size of this bounded region is reduced, iteratively, until it encloses the root to a desired tolerance. This provides a rigorous error estimate for the location of the root.

The technique of *root polishing* attempts to improve an initial guess to the root. These algorithms converge only if started “close enough” to a root, and sacrifice a rigorous error bound for speed. By approximating the behavior of a function in the vicinity of a root they attempt to find a higher order improvement of an initial guess. When the behavior of the function is compatible with the algorithm and a good initial guess is available a polishing algorithm can provide rapid convergence.

In GSL both types of algorithm are available in similar frameworks. The user provides a high-level driver for the algorithms, and the library provides the individual functions necessary for each of the steps. There are three main phases of the iteration. The steps are,

- initialize solver state, `s`, for algorithm `T`
- update `s` using the iteration `T`
- test `s` for convergence, and repeat iteration if necessary

The state for bracketing solvers is held in a `gsl_root_fsolver` struct. The updating procedure uses only function evaluations (not derivatives). The state for root polishing solvers is held in a `gsl_root_fdfsolver` struct. The updates require both the function and its derivative (hence the name `fdf`) to be supplied by the user.

## 36.2 Caveats

Note that root finding functions can only search for one root at a time. When there are several roots in the search area, the first root to be found will be returned; however it is difficult to predict which of the roots this will be. *In most cases, no error will be reported if you try to find a root in an area where there is more than one.*

Care must be taken when a function may have a multiple root (such as  $f(x) = (x - x_0)^2$  or  $f(x) = (x - x_0)^3$ ). It is not possible to use root-bracketing algorithms on even-multiplicity roots. For these algorithms the initial interval must contain a zero-crossing, where the function is negative at one end of the interval and positive at the other end. Roots with even-multiplicity do not cross zero, but only touch it instantaneously. Algorithms based on root bracketing will still work for odd-multiplicity roots (e.g. cubic, quintic, ...). Root polishing algorithms generally work with higher multiplicity roots, but at a reduced rate of convergence. In these cases the *Steffenson algorithm* can be used to accelerate the convergence of multiple roots.

While it is not absolutely required that  $f$  have a root within the search region, numerical root finding functions should not be used haphazardly to check for the *existence* of roots. There are better ways to do this. Because it is easy to create situations where numerical root finders can fail, it is a bad idea to throw a root finder at a function you do not know much about. In general it is best to examine the function visually by plotting before searching for a root.

## 36.3 Initializing the Solver

type `gsl_root_fsolver`

This is a workspace for finding roots using methods which do not require derivatives.

type `gsl_root_fdfsolver`

This is a workspace for finding roots using methods which require derivatives.

`gsl_root_fsolver *gsl_root_fsolver_alloc(const gsl_root_fsolver_type *T)`

This function returns a pointer to a newly allocated instance of a solver of type  $T$ . For example, the following code creates an instance of a bisection solver:

```
const gsl_root_fsolver_type * T = gsl_root_fsolver_bisection;
gsl_root_fsolver * s = gsl_root_fsolver_alloc (T);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

`gsl_root_fdfsolver *gsl_root_fdfsolver_alloc(const gsl_root_fdfsolver_type *T)`

This function returns a pointer to a newly allocated instance of a derivative-based solver of type  $T$ . For example, the following code creates an instance of a Newton-Raphson solver:

```
const gsl_root_fdfsolver_type * T = gsl_root_fdfsolver_newton;
gsl_root_fdfsolver * s = gsl_root_fdfsolver_alloc (T);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

int `gsl_root_fsolver_set(gsl_root_fsolver *s, gsl_function *f, double x_lower, double x_upper)`

This function initializes, or reinitializes, an existing solver  $s$  to use the function  $f$  and the initial search interval  $[x\_lower, x\_upper]$ .

int `gsl_root_fdfsolver_set(gsl_root_fdfsolver *s, gsl_function_fdf *fdf, double root)`

This function initializes, or reinitializes, an existing solver  $s$  to use the function and derivative  $fdf$  and the initial guess  $root$ .



```
void gsl_root_fsolver_free(gsl_root_fsolver *s)
```

```
void gsl_root_fdfsolver_free(gsl_root_fdfsolver *s)
```

These functions free all the memory associated with the solver *s*.

```
const char *gsl_root_fsolver_name(const gsl_root_fsolver *s)
```

```
const char *gsl_root_fdfsolver_name(const gsl_root_fdfsolver *s)
```

These functions return a pointer to the name of the solver. For example:

```
printf ("s is a '%s' solver\n", gsl_root_fsolver_name (s));
```

would print something like *s* is a 'bisection' solver.

## 36.4 Providing the function to solve

You must provide a continuous function of one variable for the root finders to operate on, and, sometimes, its first derivative. In order to allow for general parameters the functions are defined by the following data types:

type **gsl\_function**

This data type defines a general function with parameters.

```
double (* function) (double x, void * params)
```

this function should return the value  $f(x, params)$  for argument *x* and parameters *params*

```
void * params
```

a pointer to the parameters of the function

Here is an example for the general quadratic function,

$$f(x) = ax^2 + bx + c$$

with  $a = 3$ ,  $b = 2$ ,  $c = 1$ . The following code defines a *gsl\_function* *F* which you could pass to a root finder as a function pointer:

```
struct my_f_params { double a; double b; double c; };

double
my_f (double x, void * p)
{
    struct my_f_params * params = (struct my_f_params *)p;
    double a = (params->a);
    double b = (params->b);
    double c = (params->c);

    return (a * x + b) * x + c;
}

gsl_function F;
struct my_f_params params = { 3.0, 2.0, 1.0 };

F.function = &my_f;
F.params = &params;
```

The function  $f(x)$  can be evaluated using the macro `GSL_FN_EVAL(&F, x)` defined in `gsl_math.h`.

#### type `gsl_function_fdf`

This data type defines a general function with parameters and its first derivative.

```
double (* f) (double x, void * params)
```

this function should return the value of  $f(x, params)$  for argument `x` and parameters `params`

```
double (* df) (double x, void * params)
```

this function should return the value of the derivative of `f` with respect to `x`,  $f'(x, params)$ , for argument `x` and parameters `params`

```
void (* fdf) (double x, void * params, double * f, double * df)
```

this function should set the values of the function `f` to  $f(x, params)$  and its derivative `df` to  $f'(x, params)$  for argument `x` and parameters `params`. This function provides an optimization of the separate functions for  $f(x)$  and  $f'(x)$ —it is always faster to compute the function and its derivative at the same time.

```
void * params
```

a pointer to the parameters of the function

Here is an example where  $f(x) = \exp(2x)$ :

```
double
my_f (double x, void * params)
{
    return exp (2 * x);
}

double
my_df (double x, void * params)
{
    return 2 * exp (2 * x);
}

void
my_fdf (double x, void * params,
        double * f, double * df)
{
    double t = exp (2 * x);

    *f = t;
    *df = 2 * t; /* uses existing value */
}

gsl_function_fdf FDF;

FDF.f = &my_f;
FDF.df = &my_df;
FDF.fdf = &my_fdf;
FDF.params = 0;
```

The function  $f(x)$  can be evaluated using the macro `GSL_FN_FDF_EVAL_F(&FDF, x)` and the derivative  $f'(x)$  can be evaluated using the macro `GSL_FN_FDF_EVAL_DF(&FDF, x)`. Both the function  $y = f(x)$  and its derivative  $dy = f'(x)$

can be evaluated at the same time using the macro `GSL_FN_FDF_EVAL_F_DF(&FDF, x, y, dy)`. The macro stores  $f(x)$  in its `y` argument and  $f'(x)$  in its `dy` argument—both of these should be pointers to double.

## 36.5 Search Bounds and Guesses

You provide either search bounds or an initial guess; this section explains how search bounds and guesses work and how function arguments control them.

A guess is simply an  $x$  value which is iterated until it is within the desired precision of a root. It takes the form of a double.

Search bounds are the endpoints of an interval which is iterated until the length of the interval is smaller than the requested precision. The interval is defined by two values, the lower limit and the upper limit. Whether the endpoints are intended to be included in the interval or not depends on the context in which the interval is used.

## 36.6 Iteration

The following functions drive the iteration of each algorithm. Each function performs one iteration to update the state of any solver of the corresponding type. The same functions work for all solvers so that different methods can be substituted at runtime without modifications to the code.

```
int gsl_root_fsolver_iterate(gsl_root_fsolver *s)
```

```
int gsl_root_fdfsolver_iterate(gsl_root_fdfsolver *s)
```

These functions perform a single iteration of the solver  $s$ . If the iteration encounters an unexpected problem then an error code will be returned,

`GSL_EBADFUNC`

the iteration encountered a singular point where the function or its derivative evaluated to `Inf` or `NaN`.

`GSL_EZERODIV`

the derivative of the function vanished at the iteration point, preventing the algorithm from continuing without a division by zero.

The solver maintains a current best estimate of the root at all times. The bracketing solvers also keep track of the current best interval bounding the root. This information can be accessed with the following auxiliary functions,

```
double gsl_root_fsolver_root(const gsl_root_fsolver *s)
```

```
double gsl_root_fdfsolver_root(const gsl_root_fdfsolver *s)
```

These functions return the current estimate of the root for the solver  $s$ .

```
double gsl_root_fsolver_x_lower(const gsl_root_fsolver *s)
```

```
double gsl_root_fsolver_x_upper(const gsl_root_fsolver *s)
```

These functions return the current bracketing interval for the solver  $s$ .

## 36.7 Search Stopping Parameters

A root finding procedure should stop when one of the following conditions is true:

- A root has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the precision of the current result in several standard ways.

int **gsl\_root\_test\_interval**(double x\_lower, double x\_upper, double epsabs, double epsrel)

This function tests for the convergence of the interval  $[x\_lower, x\_upper]$  with absolute error *epsabs* and relative error *epsrel*. The test returns GSL\_SUCCESS if the following condition is achieved,

$$|a - b| < epsabs + epsrel \min(|a|, |b|)$$

when the interval  $x = [a, b]$  does not include the origin. If the interval includes the origin then  $\min(|a|, |b|)$  is replaced by zero (which is the minimum value of  $|x|$  over the interval). This ensures that the relative error is accurately estimated for roots close to the origin.

This condition on the interval also implies that any estimate of the root  $r$  in the interval satisfies the same condition with respect to the true root  $r^*$ ,

$$|r - r^*| < epsabs + epsrel r^*$$

assuming that the true root  $r^*$  is contained within the interval.

int **gsl\_root\_test\_delta**(double x1, double x0, double epsabs, double epsrel)

This function tests for the convergence of the sequence  $x0, x1$  with absolute error *epsabs* and relative error *epsrel*. The test returns GSL\_SUCCESS if the following condition is achieved,

$$|x_1 - x_0| < epsabs + epsrel |x_1|$$

and returns GSL\_CONTINUE otherwise.

int **gsl\_root\_test\_residual**(double f, double epsabs)

This function tests the residual value  $f$  against the absolute error bound *epsabs*. The test returns GSL\_SUCCESS if the following condition is achieved,

$$|f| < epsabs$$

and returns GSL\_CONTINUE otherwise. This criterion is suitable for situations where the precise location of the root,  $x$ , is unimportant provided a value can be found where the residual,  $|f(x)|$ , is small enough.

## 36.8 Root Bracketing Algorithms

The root bracketing algorithms described in this section require an initial interval which is guaranteed to contain a root—if  $a$  and  $b$  are the endpoints of the interval then  $f(a)$  must differ in sign from  $f(b)$ . This ensures that the function crosses zero at least once in the interval. If a valid initial interval is used then these algorithm cannot fail, provided the function is well-behaved.

Note that a bracketing algorithm cannot find roots of even degree, since these do not cross the  $x$ -axis.

type **gsl\_root\_fsolver\_type**

**`gsl_root_fsolver_type *gsl_root_fsolver_bisection`**

The *bisection algorithm* is the simplest method of bracketing the roots of a function. It is the slowest algorithm provided by the library, with linear convergence.

On each iteration, the interval is bisected and the value of the function at the midpoint is calculated. The sign of this value is used to determine which half of the interval does not contain a root. That half is discarded to give a new, smaller interval containing the root. This procedure can be continued indefinitely until the interval is sufficiently small.

At any time the current estimate of the root is taken as the midpoint of the interval.

**`gsl_root_fsolver_type *gsl_root_fsolver_falsepos`**

The *false position algorithm* is a method of finding roots based on linear interpolation. Its convergence is linear, but it is usually faster than bisection.

On each iteration a line is drawn between the endpoints  $(a, f(a))$  and  $(b, f(b))$  and the point where this line crosses the  $x$ -axis taken as a “midpoint”. The value of the function at this point is calculated and its sign is used to determine which side of the interval does not contain a root. That side is discarded to give a new, smaller interval containing the root. This procedure can be continued indefinitely until the interval is sufficiently small.

The best estimate of the root is taken from the linear interpolation of the interval on the current iteration.

**`gsl_root_fsolver_type *gsl_root_fsolver_brent`**

The *Brent-Dekker method* (referred to here as *Brent’s method*) combines an interpolation strategy with the bisection algorithm. This produces a fast algorithm which is still robust.

On each iteration Brent’s method approximates the function using an interpolating curve. On the first iteration this is a linear interpolation of the two endpoints. For subsequent iterations the algorithm uses an inverse quadratic fit to the last three points, for higher accuracy. The intercept of the interpolating curve with the  $x$ -axis is taken as a guess for the root. If it lies within the bounds of the current interval then the interpolating point is accepted, and used to generate a smaller interval. If the interpolating point is not accepted then the algorithm falls back to an ordinary bisection step.

The best estimate of the root is taken from the most recent interpolation or bisection.

## 36.9 Root Finding Algorithms using Derivatives

The root polishing algorithms described in this section require an initial guess for the location of the root. There is no absolute guarantee of convergence—the function must be suitable for this technique and the initial guess must be sufficiently close to the root for it to work. When these conditions are satisfied then convergence is quadratic.

These algorithms make use of both the function and its derivative.

**type `gsl_root_fdfsolver_type`****`gsl_root_fdfsolver_type *gsl_root_fdfsolver_newton`**

Newton’s Method is the standard root-polishing algorithm. The algorithm begins with an initial guess for the location of the root. On each iteration, a line tangent to the function  $f$  is drawn at that position. The point where this line crosses the  $x$ -axis becomes the new guess. The iteration is defined by the following sequence,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Newton’s method converges quadratically for single roots, and linearly for multiple roots.

*gsl\_root\_fdfsolver\_type* \***gsl\_root\_fdfsolver\_secant**

The *secant method* is a simplified version of Newton's method which does not require the computation of the derivative on every step.

On its first iteration the algorithm begins with Newton's method, using the derivative to compute a first step,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Subsequent iterations avoid the evaluation of the derivative by replacing it with a numerical estimate, the slope of the line through the previous two points,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'_{est}} \text{ where } f'_{est} = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

When the derivative does not change significantly in the vicinity of the root the secant method gives a useful saving. Asymptotically the secant method is faster than Newton's method whenever the cost of evaluating the derivative is more than 0.44 times the cost of evaluating the function itself. As with all methods of computing a numerical derivative the estimate can suffer from cancellation errors if the separation of the points becomes too small.

On single roots, the method has a convergence of order  $(1 + \sqrt{5})/2$  (approximately 1.62). It converges linearly for multiple roots.

*gsl\_root\_fdfsolver\_type* \***gsl\_root\_fdfsolver\_steffenson**

The *Steffenson Method*<sup>1</sup> provides the fastest convergence of all the routines. It combines the basic Newton algorithm with an Aitken "delta-squared" acceleration. If the Newton iterates are  $x_i$  then the acceleration procedure generates a new sequence  $R_i$ ,

$$R_i = x_i - \frac{(x_{i+1} - x_i)^2}{(x_{i+2} - 2x_{i+1} + x_i)}$$

which converges faster than the original sequence under reasonable conditions. The new sequence requires three terms before it can produce its first value so the method returns accelerated values on the second and subsequent iterations. On the first iteration it returns the ordinary Newton estimate. The Newton iterate is also returned if the denominator of the acceleration term ever becomes zero.

As with all acceleration procedures this method can become unstable if the function is not well-behaved.

## 36.10 Examples

For any root finding algorithm we need to prepare the function to be solved. For this example we will use the general quadratic equation described earlier. We first need a header file (`demo_fn.h`) to define the function parameters,

```
struct quadratic_params
{
    double a, b, c;
};

double quadratic (double x, void *params);
double quadratic_deriv (double x, void *params);
void quadratic_fdf (double x, void *params,
                   double *y, double *dy);
```

<sup>1</sup> J.F. Steffensen (1873–1961). The spelling used in the name of the function is slightly incorrect, but has been preserved to avoid incompatibility.

We place the function definitions in a separate file (`demo_fn.c`),

```
double
quadratic (double x, void *params)
{
    struct quadratic_params *p
        = (struct quadratic_params *) params;

    double a = p->a;
    double b = p->b;
    double c = p->c;

    return (a * x + b) * x + c;
}

double
quadratic_deriv (double x, void *params)
{
    struct quadratic_params *p
        = (struct quadratic_params *) params;

    double a = p->a;
    double b = p->b;

    return 2.0 * a * x + b;
}

void
quadratic_fdf (double x, void *params,
               double *y, double *dy)
{
    struct quadratic_params *p
        = (struct quadratic_params *) params;

    double a = p->a;
    double b = p->b;
    double c = p->c;

    *y = (a * x + b) * x + c;
    *dy = 2.0 * a * x + b;
}
```

The first program uses the function solver `gsl_root_fsolver_brent` for Brent's method and the general quadratic defined above to solve the following equation,

$$x^2 - 5 = 0$$

with solution  $x = \sqrt{5} = 2.236068\dots$

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_roots.h>
```

(continues on next page)

(continued from previous page)

```
#include "demo_fn.h"
#include "demo_fn.c"

int
main (void)
{
    int status;
    int iter = 0, max_iter = 100;
    const gsl_root_fsolver_type *T;
    gsl_root_fsolver *s;
    double r = 0, r_expected = sqrt (5.0);
    double x_lo = 0.0, x_hi = 5.0;
    gsl_function F;
    struct quadratic_params params = {1.0, 0.0, -5.0};

    F.function = &quadratic;
    F.params = &params;

    T = gsl_root_fsolver_brent;
    s = gsl_root_fsolver_alloc (T);
    gsl_root_fsolver_set (s, &F, x_lo, x_hi);

    printf ("using %s method\n",
           gsl_root_fsolver_name (s));

    printf ("%5s [%9s, %9s] %9s %10s %9s\n",
           "iter", "lower", "upper", "root",
           "err", "err(est)");

    do
    {
        iter++;
        status = gsl_root_fsolver_iterate (s);
        r = gsl_root_fsolver_root (s);
        x_lo = gsl_root_fsolver_x_lower (s);
        x_hi = gsl_root_fsolver_x_upper (s);
        status = gsl_root_test_interval (x_lo, x_hi,
                                       0, 0.001);

        if (status == GSL_SUCCESS)
            printf ("Converged:\n");

        printf ("%5d [%.7f, %.7f] %.7f %+.7f %.7f\n",
               iter, x_lo, x_hi,
               r, r - r_expected,
               x_hi - x_lo);
    }
    while (status == GSL_CONTINUE && iter < max_iter);

    gsl_root_fsolver_free (s);

    return status;
}
```

(continues on next page)



(continued from previous page)

}

Here are the results of the iterations:

```
$ ./a.out
using brent method
iter [ lower, upper] root err err(est)
  1 [1.0000000, 5.0000000] 1.0000000 -1.2360680 4.0000000
  2 [1.0000000, 3.0000000] 3.0000000 +0.7639320 2.0000000
  3 [2.0000000, 3.0000000] 2.0000000 -0.2360680 1.0000000
  4 [2.2000000, 3.0000000] 2.2000000 -0.0360680 0.8000000
  5 [2.2000000, 2.2366300] 2.2366300 +0.0005621 0.0366300
Converged:
  6 [2.2360634, 2.2366300] 2.2360634 -0.0000046 0.0005666
```

If the program is modified to use the bisection solver instead of Brent's method, by changing `gsl_root_fsolver_brent` to `gsl_root_fsolver_bisection` the slower convergence of the Bisection method can be observed:

```
$ ./a.out
using bisection method
iter [ lower, upper] root err err(est)
  1 [0.0000000, 2.5000000] 1.2500000 -0.9860680 2.5000000
  2 [1.2500000, 2.5000000] 1.8750000 -0.3610680 1.2500000
  3 [1.8750000, 2.5000000] 2.1875000 -0.0485680 0.6250000
  4 [2.1875000, 2.5000000] 2.3437500 +0.1076820 0.3125000
  5 [2.1875000, 2.3437500] 2.2656250 +0.0295570 0.1562500
  6 [2.1875000, 2.2656250] 2.2265625 -0.0095055 0.0781250
  7 [2.2265625, 2.2656250] 2.2460938 +0.0100258 0.0390625
  8 [2.2265625, 2.2460938] 2.2363281 +0.0002601 0.0195312
  9 [2.2265625, 2.2363281] 2.2314453 -0.0046227 0.0097656
 10 [2.2314453, 2.2363281] 2.2338867 -0.0021813 0.0048828
 11 [2.2338867, 2.2363281] 2.2351074 -0.0009606 0.0024414
Converged:
 12 [2.2351074, 2.2363281] 2.2357178 -0.0003502 0.0012207
```

The next program solves the same function using a derivative solver instead.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_roots.h>

#include "demo_fn.h"
#include "demo_fn.c"

int
main (void)
{
  int status;
  int iter = 0, max_iter = 100;
  const gsl_root_fdfsolver_type *T;
  gsl_root_fdfsolver *s;
```

(continues on next page)

(continued from previous page)

```

double x0, x = 5.0, r_expected = sqrt (5.0);
gsl_function_fdf FDF;
struct quadratic_params params = {1.0, 0.0, -5.0};

FDF.f = &quadratic;
FDF.df = &quadratic_deriv;
FDF.fdf = &quadratic_fdf;
FDF.params = &params;

T = gsl_root_fdfsolver_newton;
s = gsl_root_fdfsolver_alloc (T);
gsl_root_fdfsolver_set (s, &FDF, x);

printf ("using %s method\n",
        gsl_root_fdfsolver_name (s));

printf ("%5s %10s %10s %10s\n",
        "iter", "root", "err", "err(est)");
do
  {
    iter++;
    status = gsl_root_fdfsolver_iterate (s);
    x0 = x;
    x = gsl_root_fdfsolver_root (s);
    status = gsl_root_test_delta (x, x0, 0, 1e-3);

    if (status == GSL_SUCCESS)
      printf ("Converged:\n");

    printf ("%5d %10.7f %+10.7f %10.7f\n",
            iter, x, x - r_expected, x - x0);
  }
while (status == GSL_CONTINUE && iter < max_iter);

gsl_root_fdfsolver_free (s);
return status;
}

```

Here are the results for Newton's method:

```

$ ./a.out
using newton method
iter      root      err  err(est)
  1  3.00000000 +0.7639320 -2.00000000
  2  2.33333333 +0.0972654 -0.66666667
  3  2.2380952 +0.0020273 -0.0952381
Converged:
  4  2.2360689 +0.0000009 -0.0020263

```

Note that the error can be estimated more accurately by taking the difference between the current iterate and next iterate rather than the previous iterate. The other derivative solvers can be investigated by changing `gsl_root_fdfsolver_newton` to `gsl_root_fdfsolver_secant` or `gsl_root_fdfsolver_steffenson`.

## 36.11 References and Further Reading

For information on the Brent-Dekker algorithm see the following two papers,

- R. P. Brent, “An algorithm with guaranteed convergence for finding a zero of a function”, *Computer Journal*, 14 (1971) 422–425
- J. C. P. Bus and T. J. Dekker, “Two Efficient Algorithms with Guaranteed Convergence for Finding a Zero of a Function”, *ACM Transactions of Mathematical Software*, Vol.: 1 No.: 4 (1975) 330–345



## ONE DIMENSIONAL MINIMIZATION

This chapter describes routines for finding minima of arbitrary one-dimensional functions. The library provides low level components for a variety of iterative minimizers and convergence tests. These can be combined by the user to achieve the desired solution, with full access to the intermediate steps of the algorithms. Each class of methods uses the same framework, so that you can switch between minimizers at runtime without needing to recompile your program. Each instance of a minimizer keeps track of its own state, allowing the minimizers to be used in multi-threaded programs.

The header file `gsl_min.h` contains prototypes for the minimization functions and related declarations. To use the minimization algorithms to find the maximum of a function simply invert its sign.

### 37.1 Overview

The minimization algorithms begin with a bounded region known to contain a minimum. The region is described by a lower bound  $a$  and an upper bound  $b$ , with an estimate of the location of the minimum  $x$ , as shown in Fig. 37.1.

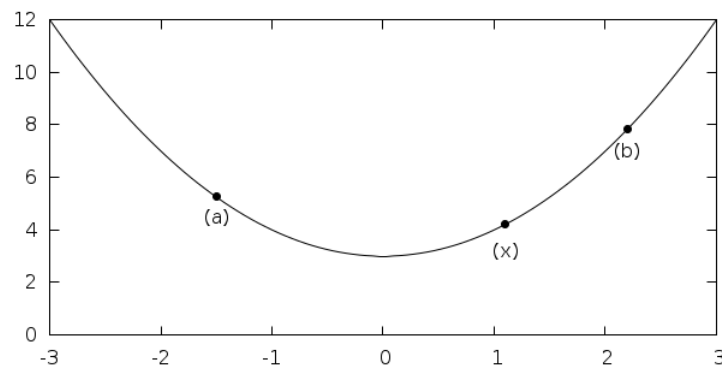


Fig. 37.1: Function with lower and upper bounds with an estimate of the minimum.

The value of the function at  $x$  must be less than the value of the function at the ends of the interval,

$$f(a) > f(x) < f(b)$$

This condition guarantees that a minimum is contained somewhere within the interval. On each iteration a new point  $x'$  is selected using one of the available algorithms. If the new point is a better estimate of the minimum, i.e.: where  $f(x') < f(x)$ , then the current estimate of the minimum  $x$  is updated. The new point also allows the size of the bounded interval to be reduced, by choosing the most compact set of points which satisfies the constraint  $f(a) > f(x) < f(b)$ . The interval is reduced until it encloses the true minimum to a desired tolerance. This provides a best estimate of the location of the minimum and a rigorous error estimate.

Several bracketing algorithms are available within a single framework. The user provides a high-level driver for the algorithm, and the library provides the individual functions necessary for each of the steps. There are three main phases of the iteration. The steps are,

- initialize minimizer state,  $s$ , for algorithm  $T$
- update  $s$  using the iteration  $T$
- test  $s$  for convergence, and repeat iteration if necessary

The state for the minimizers is held in a `gsl_min_fminimizer` struct. The updating procedure uses only function evaluations (not derivatives).

## 37.2 Caveats

Note that minimization functions can only search for one minimum at a time. When there are several minima in the search area, the first minimum to be found will be returned; however it is difficult to predict which of the minima this will be. *In most cases, no error will be reported if you try to find a minimum in an area where there is more than one.*

With all minimization algorithms it can be difficult to determine the location of the minimum to full numerical precision. The behavior of the function in the region of the minimum  $x^*$  can be approximated by a Taylor expansion,

$$y = f(x^*) + \frac{1}{2}f''(x^*)(x - x^*)^2$$

and the second term of this expansion can be lost when added to the first term at finite precision. This magnifies the error in locating  $x^*$ , making it proportional to  $\sqrt{\epsilon}$  (where  $\epsilon$  is the relative accuracy of the floating point numbers). For functions with higher order minima, such as  $x^4$ , the magnification of the error is correspondingly worse. The best that can be achieved is to converge to the limit of numerical accuracy in the function values, rather than the location of the minimum itself.

## 37.3 Initializing the Minimizer

type `gsl_min_fminimizer`

This is a workspace for minimizing functions.

`gsl_min_fminimizer *gsl_min_fminimizer_alloc(const gsl_min_fminimizer_type *T)`

This function returns a pointer to a newly allocated instance of a minimizer of type  $T$ . For example, the following code creates an instance of a golden section minimizer:

```
const gsl_min_fminimizer_type * T = gsl_min_fminimizer_goldensection;
gsl_min_fminimizer * s = gsl_min_fminimizer_alloc (T);
```

If there is insufficient memory to create the minimizer then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

int `gsl_min_fminimizer_set(gsl_min_fminimizer *s, gsl_function *f, double x_minimum, double x_lower, double x_upper)`

This function sets, or resets, an existing minimizer  $s$  to use the function  $f$  and the initial search interval [ $x\_lower$ ,  $x\_upper$ ], with a guess for the location of the minimum  $x\_minimum$ .

If the interval given does not contain a minimum, then the function returns an error code of `GSL_EINVAL`.

```
int gsl_min_fminimizer_set_with_values(gsl_min_fminimizer *s, gsl_function *f, double x_minimum, double
                                     f_minimum, double x_lower, double f_lower, double x_upper,
                                     double f_upper)
```

This function is equivalent to `gsl_min_fminimizer_set()` but uses the values `f_minimum`, `f_lower` and `f_upper` instead of computing `f(x_minimum)`, `f(x_lower)` and `f(x_upper)`.

```
void gsl_min_fminimizer_free(gsl_min_fminimizer *s)
```

This function frees all the memory associated with the minimizer `s`.

```
const char *gsl_min_fminimizer_name(const gsl_min_fminimizer *s)
```

This function returns a pointer to the name of the minimizer. For example:

```
printf ("s is a '%s' minimizer\n", gsl_min_fminimizer_name (s));
```

would print something like `s is a 'brent' minimizer`.

## 37.4 Providing the function to minimize

You must provide a continuous function of one variable for the minimizers to operate on. In order to allow for general parameters the functions are defined by a `gsl_function` data type (*Providing the function to solve*).

## 37.5 Iteration

The following functions drive the iteration of each algorithm. Each function performs one iteration to update the state of any minimizer of the corresponding type. The same functions work for all minimizers so that different methods can be substituted at runtime without modifications to the code.

```
int gsl_min_fminimizer_iterate(gsl_min_fminimizer *s)
```

This function performs a single iteration of the minimizer `s`. If the iteration encounters an unexpected problem then an error code will be returned,

```
GSL_EBADFUNC
```

the iteration encountered a singular point where the function evaluated to `Inf` or `NaN`.

```
GSL_FAILURE
```

the algorithm could not improve the current best approximation or bounding interval.

The minimizer maintains a current best estimate of the position of the minimum at all times, and the current interval bounding the minimum. This information can be accessed with the following auxiliary functions,

```
double gsl_min_fminimizer_x_minimum(const gsl_min_fminimizer *s)
```

This function returns the current estimate of the position of the minimum for the minimizer `s`.

```
double gsl_min_fminimizer_x_upper(const gsl_min_fminimizer *s)
```

```
double gsl_min_fminimizer_x_lower(const gsl_min_fminimizer *s)
```

These functions return the current upper and lower bound of the interval for the minimizer `s`.

```
double gsl_min_fminimizer_f_minimum(const gsl_min_fminimizer *s)
```

```
double gsl_min_fminimizer_f_upper(const gsl_min_fminimizer *s)
```

double **gsl\_min\_fminimizer\_f\_lower**(const *gsl\_min\_fminimizer* \*s)

These functions return the value of the function at the current estimate of the minimum and at the upper and lower bounds of the interval for the minimizer *s*.

## 37.6 Stopping Parameters

A minimization procedure should stop when one of the following conditions is true:

- A minimum has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The function below allows the user to test the precision of the current result.

int **gsl\_min\_test\_interval**(double *x\_lower*, double *x\_upper*, double *epsabs*, double *epsrel*)

This function tests for the convergence of the interval [*x\_lower*, *x\_upper*] with absolute error *epsabs* and relative error *epsrel*. The test returns GSL\_SUCCESS if the following condition is achieved,

$$|a - b| < \textit{epsabs} + \textit{epsrel} \min(|a|, |b|)$$

when the interval  $x = [a, b]$  does not include the origin. If the interval includes the origin then  $\min(|a|, |b|)$  is replaced by zero (which is the minimum value of  $|x|$  over the interval). This ensures that the relative error is accurately estimated for minima close to the origin.

This condition on the interval also implies that any estimate of the minimum  $x_m$  in the interval satisfies the same condition with respect to the true minimum  $x_m^*$ ,

$$|x_m - x_m^*| < \textit{epsabs} + \textit{epsrel} x_m^*$$

assuming that the true minimum  $x_m^*$  is contained within the interval.

## 37.7 Minimization Algorithms

The minimization algorithms described in this section require an initial interval which is guaranteed to contain a minimum—if *a* and *b* are the endpoints of the interval and *x* is an estimate of the minimum then  $f(a) > f(x) < f(b)$ . This ensures that the function has at least one minimum somewhere in the interval. If a valid initial interval is used then these algorithm cannot fail, provided the function is well-behaved.

type **gsl\_min\_fminimizer\_type**

*gsl\_min\_fminimizer\_type* \***gsl\_min\_fminimizer\_goldensection**

The *golden section algorithm* is the simplest method of bracketing the minimum of a function. It is the slowest algorithm provided by the library, with linear convergence.

On each iteration, the algorithm first compares the subintervals from the endpoints to the current minimum. The larger subinterval is divided in a golden section (using the famous ratio  $(3 - \sqrt{5})/2 \approx 0.3819660$  and the value of the function at this new point is calculated. The new value is used with the constraint  $f(a') > f(x') < f(b')$  to select new interval containing the minimum, by discarding the least useful point. This procedure can be continued indefinitely until the interval is sufficiently small. Choosing the golden section as the bisection ratio can be shown to provide the fastest convergence for this type of algorithm.



`gsl_min_fminimizer_type *gsl_min_fminimizer_brent`

The *Brent minimization algorithm* combines a parabolic interpolation with the golden section algorithm. This produces a fast algorithm which is still robust.

The outline of the algorithm can be summarized as follows: on each iteration Brent's method approximates the function using an interpolating parabola through three existing points. The minimum of the parabola is taken as a guess for the minimum. If it lies within the bounds of the current interval then the interpolating point is accepted, and used to generate a smaller interval. If the interpolating point is not accepted then the algorithm falls back to an ordinary golden section step. The full details of Brent's method include some additional checks to improve convergence.

`gsl_min_fminimizer_type *gsl_min_fminimizer_quad_golden`

This is a variant of Brent's algorithm which uses the safeguarded step-length algorithm of Gill and Murray.

## 37.8 Examples

The following program uses the Brent algorithm to find the minimum of the function  $f(x) = \cos(x) + 1$ , which occurs at  $x = \pi$ . The starting interval is  $(0, 6)$ , with an initial guess for the minimum of 2.

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_min.h>

double fn1 (double x, void * params)
{
  (void)(params); /* avoid unused parameter warning */
  return cos(x) + 1.0;
}

int
main (void)
{
  int status;
  int iter = 0, max_iter = 100;
  const gsl_min_fminimizer_type *T;
  gsl_min_fminimizer *s;
  double m = 2.0, m_expected = M_PI;
  double a = 0.0, b = 6.0;
  gsl_function F;

  F.function = &fn1;
  F.params = 0;

  T = gsl_min_fminimizer_brent;
  s = gsl_min_fminimizer_alloc (T);
  gsl_min_fminimizer_set (s, &F, m, a, b);

  printf ("using %s method\n",
         gsl_min_fminimizer_name (s));

  printf ("%5s [%9s, %9s] %9s %10s %9s\n",
         "iter", "lower", "upper", "min",
```

(continues on next page)

```

        "err", "err(est)");

printf ("%5d [%.7f, %.7f] %.7f %+.7f %.7f\n",
        iter, a, b,
        m, m - m_expected, b - a);

do
{
    iter++;
    status = gsl_min_fminimizer_iterate (s);

    m = gsl_min_fminimizer_x_minimum (s);
    a = gsl_min_fminimizer_x_lower (s);
    b = gsl_min_fminimizer_x_upper (s);

    status
        = gsl_min_test_interval (a, b, 0.001, 0.0);

    if (status == GSL_SUCCESS)
        printf ("Converged:\n");

    printf ("%5d [%.7f, %.7f] "
            "%.7f %+.7f %.7f\n",
            iter, a, b,
            m, m - m_expected, b - a);
}
while (status == GSL_CONTINUE && iter < max_iter);

gsl_min_fminimizer_free (s);

return status;
}

```

Here are the results of the minimization procedure.

```

using brent method
iter [ lower, upper] min err err(est)
  0 [0.0000000, 6.0000000] 2.0000000 -1.1415927 6.0000000
  1 [2.0000000, 6.0000000] 3.5278640 +0.3862713 4.0000000
  2 [2.0000000, 3.5278640] 3.1748217 +0.0332290 1.5278640
  3 [2.0000000, 3.1748217] 3.1264576 -0.0151351 1.1748217
  4 [3.1264576, 3.1748217] 3.1414743 -0.0001183 0.0483641
  5 [3.1414743, 3.1748217] 3.1415930 +0.0000004 0.0333474
Converged:
  6 [3.1414743, 3.1415930] 3.1415927 +0.0000000 0.0001187

```

## 37.9 References and Further Reading

Further information on Brent's algorithm is available in the following book,

- Richard Brent, *Algorithms for minimization without derivatives*, Prentice-Hall (1973), republished by Dover in paperback (2002), ISBN 0-486-41998-3.



## MULTIDIMENSIONAL ROOT-FINDING

This chapter describes functions for multidimensional root-finding (solving nonlinear systems with  $n$  equations in  $n$  unknowns). The library provides low level components for a variety of iterative solvers and convergence tests. These can be combined by the user to achieve the desired solution, with full access to the intermediate steps of the iteration. Each class of methods uses the same framework, so that you can switch between solvers at runtime without needing to recompile your program. Each instance of a solver keeps track of its own state, allowing the solvers to be used in multi-threaded programs. The solvers are based on the original Fortran library MINPACK.

The header file `gsl_multiroots.h` contains prototypes for the multidimensional root finding functions and related declarations.

### 38.1 Overview

The problem of multidimensional root finding requires the simultaneous solution of  $n$  equations,  $f_i$ , in  $n$  variables,  $x_i$ ,

$$f_i(x_1, \dots, x_n) = 0 \quad \text{for } i = 1 \dots n.$$

In general there are no bracketing methods available for  $n$  dimensional systems, and no way of knowing whether any solutions exist. All algorithms proceed from an initial guess using a variant of the Newton iteration,

$$x \rightarrow x' = x - J^{-1}f(x)$$

where  $x$ ,  $f$  are vector quantities and  $J$  is the Jacobian matrix  $J_{ij} = \partial f_i / \partial x_j$ . Additional strategies can be used to enlarge the region of convergence. These include requiring a decrease in the norm  $|f|$  on each step proposed by Newton's method, or taking steepest-descent steps in the direction of the negative gradient of  $|f|$ .

Several root-finding algorithms are available within a single framework. The user provides a high-level driver for the algorithms, and the library provides the individual functions necessary for each of the steps. There are three main phases of the iteration. The steps are,

- initialize solver state,  $s$ , for algorithm T
- update  $s$  using the iteration T
- test  $s$  for convergence, and repeat iteration if necessary

The evaluation of the Jacobian matrix can be problematic, either because programming the derivatives is intractable or because computation of the  $n^2$  terms of the matrix becomes too expensive. For these reasons the algorithms provided by the library are divided into two classes according to whether the derivatives are available or not.

The state for solvers with an analytic Jacobian matrix is held in a `gsl_multiroot_fdfsolver` struct. The updating procedure requires both the function and its derivatives to be supplied by the user.

The state for solvers which do not use an analytic Jacobian matrix is held in a `gsl_multiroot_fsolver` struct. The updating procedure uses only function evaluations (not derivatives). The algorithms estimate the matrix  $J$  or  $J^{-1}$  by approximate methods.

## 38.2 Initializing the Solver

The following functions initialize a multidimensional solver, either with or without derivatives. The solver itself depends only on the dimension of the problem and the algorithm and can be reused for different problems.

type **gsl\_multiroot\_fsolver**

This is a workspace for multidimensional root-finding without derivatives.

type **gsl\_multiroot\_fdfsolver**

This is a workspace for multidimensional root-finding with derivatives.

*gsl\_multiroot\_fsolver* \***gsl\_multiroot\_fsolver\_alloc**(const *gsl\_multiroot\_fsolver\_type* \*T, size\_t n)

This function returns a pointer to a newly allocated instance of a solver of type *T* for a system of *n* dimensions. For example, the following code creates an instance of a hybrid solver, to solve a 3-dimensional system of equations:

```
const gsl_multiroot_fsolver_type * T = gsl_multiroot_fsolver_hybrid;
gsl_multiroot_fsolver * s = gsl_multiroot_fsolver_alloc (T, 3);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of *GSL\_ENOMEM*.

*gsl\_multiroot\_fdfsolver* \***gsl\_multiroot\_fdfsolver\_alloc**(const *gsl\_multiroot\_fdfsolver\_type* \*T, size\_t n)

This function returns a pointer to a newly allocated instance of a derivative solver of type *T* for a system of *n* dimensions. For example, the following code creates an instance of a Newton-Raphson solver, for a 2-dimensional system of equations:

```
const gsl_multiroot_fdfsolver_type * T = gsl_multiroot_fdfsolver_newton;
gsl_multiroot_fdfsolver * s = gsl_multiroot_fdfsolver_alloc (T, 2);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of *GSL\_ENOMEM*.

int **gsl\_multiroot\_fsolver\_set**(*gsl\_multiroot\_fsolver* \*s, *gsl\_multiroot\_function* \*f, const *gsl\_vector* \*x)

int **gsl\_multiroot\_fdfsolver\_set**(*gsl\_multiroot\_fdfsolver* \*s, *gsl\_multiroot\_function\_df* \*fdf, const *gsl\_vector* \*x)

These functions set, or reset, an existing solver *s* to use the function *f* or function and derivative *fdf*, and the initial guess *x*. Note that the initial position is copied from *x*, this argument is not modified by subsequent iterations.

void **gsl\_multiroot\_fsolver\_free**(*gsl\_multiroot\_fsolver* \*s)

void **gsl\_multiroot\_fdfsolver\_free**(*gsl\_multiroot\_fdfsolver* \*s)

These functions free all the memory associated with the solver *s*.

const char \***gsl\_multiroot\_fsolver\_name**(const *gsl\_multiroot\_fsolver* \*s)

const char \***gsl\_multiroot\_fdfsolver\_name**(const *gsl\_multiroot\_fdfsolver* \*s)

These functions return a pointer to the name of the solver. For example:

```
printf ("s is a '%s' solver\n", gsl_multiroot_fdfsolver_name (s));
```

would print something like *s is a 'newton' solver*.

## 38.3 Providing the function to solve

You must provide  $n$  functions of  $n$  variables for the root finders to operate on. In order to allow for general parameters the functions are defined by the following data types:

### type `gsl_multiroot_function`

This data type defines a general system of functions with parameters.

```
int (* f) (const gsl_vector * x, void * params, gsl_vector * f)
```

this function should store the vector result  $f(x, params)$  in `f` for argument `x` and parameters `params`, returning an appropriate error code if the function cannot be computed.

```
size_t n
```

the dimension of the system, i.e. the number of components of the vectors `x` and `f`.

```
void * params
```

a pointer to the parameters of the function.

Here is an example using Powell's test function,

$$f_1(x) = Ax_0x_1 - 1$$

$$f_2(x) = \exp(-x_0) + \exp(-x_1) - (1 + 1/A)$$

with  $A = 10^4$ . The following code defines a `gsl_multiroot_function` system `F` which you could pass to a solver:

```
struct powell_params { double A; };

int
powell (gsl_vector * x, void * p, gsl_vector * f) {
    struct powell_params * params
        = (struct powell_params *)p;
    const double A = (params->A);
    const double x0 = gsl_vector_get(x,0);
    const double x1 = gsl_vector_get(x,1);

    gsl_vector_set (f, 0, A * x0 * x1 - 1);
    gsl_vector_set (f, 1, (exp(-x0) + exp(-x1)
        - (1.0 + 1.0/A)));
    return GSL_SUCCESS
}

gsl_multiroot_function F;
struct powell_params params = { 10000.0 };

F.f = &powell;
F.n = 2;
F.params = &params;
```

### type `gsl_multiroot_function_fdf`

This data type defines a general system of functions with parameters and the corresponding Jacobian matrix of derivatives,

```
int (* f) (const gsl_vector * x, void * params, gsl_vector * f)
```

this function should store the vector result  $f(x, params)$  in `f` for argument `x` and parameters `params`, returning an appropriate error code if the function cannot be computed.

```
int (* df) (const gsl_vector * x, void * params, gsl_matrix * J)
```

this function should store the n-by-n matrix result

$$J_{ij} = \partial f_i(x, params) / \partial x_j$$

in J for argument x and parameters params, returning an appropriate error code if the function cannot be computed.

```
int (* fdf) (const gsl_vector * x, void * params, gsl_vector * f, gsl_matrix * J)
```

This function should set the values of the f and J as above, for arguments x and parameters params. This function provides an optimization of the separate functions for  $f(x)$  and  $J(x)$ —it is always faster to compute the function and its derivative at the same time.

```
size_t n
```

the dimension of the system, i.e. the number of components of the vectors x and f.

```
void * params
```

a pointer to the parameters of the function.

The example of Powell's test function defined above can be extended to include analytic derivatives using the following code:

```
int
powell_df (gsl_vector * x, void * p, gsl_matrix * J)
{
    struct powell_params * params
        = (struct powell_params *)p;
    const double A = (params->A);
    const double x0 = gsl_vector_get(x,0);
    const double x1 = gsl_vector_get(x,1);
    gsl_matrix_set (J, 0, 0, A * x1);
    gsl_matrix_set (J, 0, 1, A * x0);
    gsl_matrix_set (J, 1, 0, -exp(-x0));
    gsl_matrix_set (J, 1, 1, -exp(-x1));
    return GSL_SUCCESS
}

int
powell_fdf (gsl_vector * x, void * p,
            gsl_matrix * f, gsl_matrix * J) {
    struct powell_params * params
        = (struct powell_params *)p;
    const double A = (params->A);
    const double x0 = gsl_vector_get(x,0);
    const double x1 = gsl_vector_get(x,1);

    const double u0 = exp(-x0);
    const double u1 = exp(-x1);

    gsl_vector_set (f, 0, A * x0 * x1 - 1);
    gsl_vector_set (f, 1, u0 + u1 - (1 + 1/A));

    gsl_matrix_set (J, 0, 0, A * x1);
    gsl_matrix_set (J, 0, 1, A * x0);
```

(continues on next page)



(continued from previous page)

```

    gsl_matrix_set (J, 1, 0, -u0);
    gsl_matrix_set (J, 1, 1, -u1);
    return GSL_SUCCESS
}

gsl_multiroot_function_fdf FDF;

FDF.f = &powell_f;
FDF.df = &powell_df;
FDF.fdf = &powell_fdf;
FDF.n = 2;
FDF.params = 0;

```

Note that the function `powell_fdf` is able to reuse existing terms from the function when calculating the Jacobian, thus saving time.

## 38.4 Iteration

The following functions drive the iteration of each algorithm. Each function performs one iteration to update the state of any solver of the corresponding type. The same functions work for all solvers so that different methods can be substituted at runtime without modifications to the code.

```
int gsl_multiroot_fsolver_iterate(gsl_multiroot_fsolver *s)
```

```
int gsl_multiroot_fdfsolver_iterate(gsl_multiroot_fdfsolver *s)
```

These functions perform a single iteration of the solver *s*. If the iteration encounters an unexpected problem then an error code will be returned,

`GSL_EBADFUNC`

the iteration encountered a singular point where the function or its derivative evaluated to Inf or NaN.

`GSL_ENOPROG`

the iteration is not making any progress, preventing the algorithm from continuing.

The solver maintains a current best estimate of the root  $s \rightarrow x$  and its function value  $s \rightarrow f$  at all times. This information can be accessed with the following auxiliary functions,

```
gsl_vector *gsl_multiroot_fsolver_root(const gsl_multiroot_fsolver *s)
```

```
gsl_vector *gsl_multiroot_fdfsolver_root(const gsl_multiroot_fdfsolver *s)
```

These functions return the current estimate of the root for the solver *s*, given by  $s \rightarrow x$ .

```
gsl_vector *gsl_multiroot_fsolver_f(const gsl_multiroot_fsolver *s)
```

```
gsl_vector *gsl_multiroot_fdfsolver_f(const gsl_multiroot_fdfsolver *s)
```

These functions return the function value  $f(x)$  at the current estimate of the root for the solver *s*, given by  $s \rightarrow f$ .

```
gsl_vector *gsl_multiroot_fsolver_dx(const gsl_multiroot_fsolver *s)
```

```
gsl_vector *gsl_multiroot_fdfsolver_dx(const gsl_multiroot_fdfsolver *s)
```

These functions return the last step  $dx$  taken by the solver *s*, given by  $s \rightarrow dx$ .

## 38.5 Search Stopping Parameters

A root finding procedure should stop when one of the following conditions is true:

- A multidimensional root has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the precision of the current result in several standard ways.

int **gsl\_multiroot\_test\_delta**(const *gsl\_vector* \*dx, const *gsl\_vector* \*x, double epsabs, double epsrel)

This function tests for the convergence of the sequence by comparing the last step  $dx$  with the absolute error  $epsabs$  and relative error  $epsrel$  to the current position  $x$ . The test returns `GSL_SUCCESS` if the following condition is achieved,

$$|dx_i| < epsabs + epsrel |x_i|$$

for each component of  $x$  and returns `GSL_CONTINUE` otherwise.

int **gsl\_multiroot\_test\_residual**(const *gsl\_vector* \*f, double epsabs)

This function tests the residual value  $f$  against the absolute error bound  $epsabs$ . The test returns `GSL_SUCCESS` if the following condition is achieved,

$$\sum_i |f_i| < epsabs$$

and returns `GSL_CONTINUE` otherwise. This criterion is suitable for situations where the precise location of the root,  $x$ , is unimportant provided a value can be found where the residual is small enough.

## 38.6 Algorithms using Derivatives

The root finding algorithms described in this section make use of both the function and its derivative. They require an initial guess for the location of the root, but there is no absolute guarantee of convergence—the function must be suitable for this technique and the initial guess must be sufficiently close to the root for it to work. When the conditions are satisfied then convergence is quadratic.

type **gsl\_multiroot\_fdfsolver\_type**

The following are available algorithms for minimizing functions using derivatives.

*gsl\_multiroot\_fdfsolver\_type* \***gsl\_multiroot\_fdfsolver\_hybridsj**

This is a modified version of Powell's Hybrid method as implemented in the HYBRJ algorithm in MINPACK. Minpack was written by Jorge J. Moré, Burton S. Garbow and Kenneth E. Hillstom. The Hybrid algorithm retains the fast convergence of Newton's method but will also reduce the residual when Newton's method is unreliable.

The algorithm uses a generalized trust region to keep each step under control. In order to be accepted a proposed new position  $x'$  must satisfy the condition  $|D(x' - x)| < \delta$ , where  $D$  is a diagonal scaling matrix and  $\delta$  is the size of the trust region. The components of  $D$  are computed internally, using the column norms of the Jacobian to estimate the sensitivity of the residual to each component of  $x$ . This improves the behavior of the algorithm for badly scaled functions.

On each iteration the algorithm first determines the standard Newton step by solving the system  $Jdx = -f$ . If this step falls inside the trust region it is used as a trial step in the next stage. If not, the algorithm uses

the linear combination of the Newton and gradient directions which is predicted to minimize the norm of the function while staying inside the trust region,

$$dx = -\alpha J^{-1} f(x) - \beta \nabla |f(x)|^2$$

This combination of Newton and gradient directions is referred to as a *dogleg step*.

The proposed step is now tested by evaluating the function at the resulting point,  $x'$ . If the step reduces the norm of the function sufficiently then it is accepted and size of the trust region is increased. If the proposed step fails to improve the solution then the size of the trust region is decreased and another trial step is computed.

The speed of the algorithm is increased by computing the changes to the Jacobian approximately, using a rank-1 update. If two successive attempts fail to reduce the residual then the full Jacobian is recomputed. The algorithm also monitors the progress of the solution and returns an error if several steps fail to make any improvement,

`GSL_ENOPROG`

the iteration is not making any progress, preventing the algorithm from continuing.

`GSL_ENOPROGJ`

re-evaluations of the Jacobian indicate that the iteration is not making any progress, preventing the algorithm from continuing.

`gsl_multiroot_fdfsolver_type *gsl_multiroot_fdfsolver_hybridj`

This algorithm is an unscaled version of HYBRIDSJ. The steps are controlled by a spherical trust region  $|x' - x| < \delta$ , instead of a generalized region. This can be useful if the generalized region estimated by HYBRIDSJ is inappropriate.

`gsl_multiroot_fdfsolver_type *gsl_multiroot_fdfsolver_newton`

Newton's Method is the standard root-polishing algorithm. The algorithm begins with an initial guess for the location of the solution. On each iteration a linear approximation to the function  $F$  is used to estimate the step which will zero all the components of the residual. The iteration is defined by the following sequence,

$$x \rightarrow x' = x - J^{-1} f(x)$$

where the Jacobian matrix  $J$  is computed from the derivative functions provided by  $f$ . The step  $dx$  is obtained by solving the linear system,

$$Jdx = -f(x)$$

using LU decomposition. If the Jacobian matrix is singular, an error code of `GSL_EDOM` is returned.

`gsl_multiroot_fdfsolver_type *gsl_multiroot_fdfsolver_gnewton`

This is a modified version of Newton's method which attempts to improve global convergence by requiring every step to reduce the Euclidean norm of the residual,  $|f(x)|$ . If the Newton step leads to an increase in the norm then a reduced step of relative size,

$$t = (\sqrt{1 + 6r} - 1)/(3r)$$

is proposed, with  $r$  being the ratio of norms  $|f(x')|^2/|f(x)|^2$ . This procedure is repeated until a suitable step size is found.

## 38.7 Algorithms without Derivatives

The algorithms described in this section do not require any derivative information to be supplied by the user. Any derivatives needed are approximated by finite differences. Note that if the finite-differencing step size chosen by these routines is inappropriate, an explicit user-supplied numerical derivative can always be used with the algorithms described in the previous section.

type `gsl_multiroot_fsolver_type`

The following are available algorithms for minimizing functions without derivatives.

`gsl_multiroot_fsolver_type` \*`gsl_multiroot_fsolver_hybrids`

This is a version of the Hybrid algorithm which replaces calls to the Jacobian function by its finite difference approximation. The finite difference approximation is computed using `gsl_multiroots_fdjac()` with a relative step size of `GSL_SQRT_DBL_EPSILON`. Note that this step size will not be suitable for all problems.

`gsl_multiroot_fsolver_type` \*`gsl_multiroot_fsolver_hybrid`

This is a finite difference version of the Hybrid algorithm without internal scaling.

`gsl_multiroot_fsolver_type` \*`gsl_multiroot_fsolver_dnewton`

The *discrete Newton algorithm* is the simplest method of solving a multidimensional system. It uses the Newton iteration

$$x \rightarrow x - J^{-1}f(x)$$

where the Jacobian matrix  $J$  is approximated by taking finite differences of the function  $f$ . The approximation scheme used by this implementation is,

$$J_{ij} = (f_i(x + \delta_j) - f_i(x))/\delta_j$$

where  $\delta_j$  is a step of size  $\sqrt{\epsilon}|x_j|$  with  $\epsilon$  being the machine precision ( $\epsilon \approx 2.22 \times 10^{-16}$ ). The order of convergence of Newton's algorithm is quadratic, but the finite differences require  $n^2$  function evaluations on each iteration. The algorithm may become unstable if the finite differences are not a good approximation to the true derivatives.

`gsl_multiroot_fsolver_type` \*`gsl_multiroot_fsolver_broyden`

The *Broyden algorithm* is a version of the discrete Newton algorithm which attempts to avoid the expensive update of the Jacobian matrix on each iteration. The changes to the Jacobian are also approximated, using a rank-1 update,

$$J^{-1} \rightarrow J^{-1} - (J^{-1}df - dx)dx^T J^{-1} / dx^T J^{-1}df$$

where the vectors  $dx$  and  $df$  are the changes in  $x$  and  $f$ . On the first iteration the inverse Jacobian is estimated using finite differences, as in the discrete Newton algorithm.

This approximation gives a fast update but is unreliable if the changes are not small, and the estimate of the inverse Jacobian becomes worse as time passes. The algorithm has a tendency to become unstable unless it starts close to the root. The Jacobian is refreshed if this instability is detected (consult the source for details).

This algorithm is included only for demonstration purposes, and is not recommended for serious use.

## 38.8 Examples

The multidimensional solvers are used in a similar way to the one-dimensional root finding algorithms. This first example demonstrates the HYBRIDS scaled-hybrid algorithm, which does not require derivatives. The program solves the Rosenbrock system of equations,

$$f_1(x, y) = a(1 - x)$$

$$f_2(x, y) = b(y - x^2)$$

with  $a = 1, b = 10$ . The solution of this system lies at  $(x, y) = (1, 1)$  in a narrow valley.

The first stage of the program is to define the system of equations:

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_multiroots.h>

struct rparams
{
    double a;
    double b;
};

int
rosenbrock_f (const gsl_vector * x, void *params,
              gsl_vector * f)
{
    double a = ((struct rparams *) params)->a;
    double b = ((struct rparams *) params)->b;

    const double x0 = gsl_vector_get (x, 0);
    const double x1 = gsl_vector_get (x, 1);

    const double y0 = a * (1 - x0);
    const double y1 = b * (x1 - x0 * x0);

    gsl_vector_set (f, 0, y0);
    gsl_vector_set (f, 1, y1);

    return GSL_SUCCESS;
}
```

The main program begins by creating the function object `f`, with the arguments  $(x, y)$  and parameters  $(a, b)$ . The solver `s` is initialized to use this function, with the `gsl_multiroot_fsolver_hybrids` method:

```
int
main (void)
{
    const gsl_multiroot_fsolver_type *T;
    gsl_multiroot_fsolver *s;

    int status;
    size_t i, iter = 0;
```

(continues on next page)

(continued from previous page)

```

const size_t n = 2;
struct rparams p = {1.0, 10.0};
gsl_multiroot_function f = {&rosenbrock_f, n, &p};

double x_init[2] = {-10.0, -5.0};
gsl_vector *x = gsl_vector_alloc (n);

gsl_vector_set (x, 0, x_init[0]);
gsl_vector_set (x, 1, x_init[1]);

T = gsl_multiroot_fsolver_hybrids;
s = gsl_multiroot_fsolver_alloc (T, 2);
gsl_multiroot_fsolver_set (s, &f, x);

print_state (iter, s);

do
{
    iter++;
    status = gsl_multiroot_fsolver_iterate (s);

    print_state (iter, s);

    if (status) /* check if solver is stuck */
        break;

    status =
        gsl_multiroot_test_residual (s->f, 1e-7);
}
while (status == GSL_CONTINUE && iter < 1000);

printf ("status = %s\n", gsl_strerror (status));

gsl_multiroot_fsolver_free (s);
gsl_vector_free (x);
return 0;
}

```

Note that it is important to check the return status of each solver step, in case the algorithm becomes stuck. If an error condition is detected, indicating that the algorithm cannot proceed, then the error can be reported to the user, a new starting point chosen or a different algorithm used.

The intermediate state of the solution is displayed by the following function. The solver state contains the vector `s->x` which is the current position, and the vector `s->f` with corresponding function values:

```

int
print_state (size_t iter, gsl_multiroot_fsolver * s)
{
    printf ("iter = %3u x = % .3f % .3f "
           "f(x) = % .3e % .3e\n",
           iter,
           gsl_vector_get (s->x, 0),

```

(continues on next page)

(continued from previous page)

```

    gsl_vector_get (s->x, 1),
    gsl_vector_get (s->f, 0),
    gsl_vector_get (s->f, 1));
}

```

Here are the results of running the program. The algorithm is started at  $(-10, -5)$  far from the solution. Since the solution is hidden in a narrow valley the earliest steps follow the gradient of the function downhill, in an attempt to reduce the large value of the residual. Once the root has been approximately located, on iteration 8, the Newton behavior takes over and convergence is very rapid:

```

iter = 0 x = -10.000 -5.000 f(x) = 1.100e+01 -1.050e+03
iter = 1 x = -10.000 -5.000 f(x) = 1.100e+01 -1.050e+03
iter = 2 x = -3.976 24.827 f(x) = 4.976e+00 9.020e+01
iter = 3 x = -3.976 24.827 f(x) = 4.976e+00 9.020e+01
iter = 4 x = -3.976 24.827 f(x) = 4.976e+00 9.020e+01
iter = 5 x = -1.274 -5.680 f(x) = 2.274e+00 -7.302e+01
iter = 6 x = -1.274 -5.680 f(x) = 2.274e+00 -7.302e+01
iter = 7 x = 0.249 0.298 f(x) = 7.511e-01 2.359e+00
iter = 8 x = 0.249 0.298 f(x) = 7.511e-01 2.359e+00
iter = 9 x = 1.000 0.878 f(x) = 1.268e-10 -1.218e+00
iter = 10 x = 1.000 0.989 f(x) = 1.124e-11 -1.080e-01
iter = 11 x = 1.000 1.000 f(x) = 0.000e+00 0.000e+00
status = success

```

Note that the algorithm does not update the location on every iteration. Some iterations are used to adjust the trust-region parameter, after trying a step which was found to be divergent, or to recompute the Jacobian, when poor convergence behavior is detected.

The next example program adds derivative information, in order to accelerate the solution. There are two derivative functions `rosenbrock_df` and `rosenbrock_fdf`. The latter computes both the function and its derivative simultaneously. This allows the optimization of any common terms. For simplicity we substitute calls to the separate `f` and `df` functions at this point in the code below:

```

int
rosenbrock_df (const gsl_vector * x, void *params,
              gsl_matrix * J)
{
    const double a = ((struct rparams *) params)->a;
    const double b = ((struct rparams *) params)->b;

    const double x0 = gsl_vector_get (x, 0);

    const double df00 = -a;
    const double df01 = 0;
    const double df10 = -2 * b * x0;
    const double df11 = b;

    gsl_matrix_set (J, 0, 0, df00);
    gsl_matrix_set (J, 0, 1, df01);
    gsl_matrix_set (J, 1, 0, df10);
    gsl_matrix_set (J, 1, 1, df11);

    return GSL_SUCCESS;
}

```

(continues on next page)

(continued from previous page)

```
}  
  
int  
rosenbrock_fdf (const gsl_vector * x, void *params,  
                gsl_vector * f, gsl_matrix * J)  
{  
    rosenbrock_f (x, params, f);  
    rosenbrock_df (x, params, J);  
  
    return GSL_SUCCESS;  
}
```

The main program now makes calls to the corresponding `fdfsolver` versions of the functions:

```
int  
main (void)  
{  
    const gsl_multiroot_fdfsolver_type *T;  
    gsl_multiroot_fdfsolver *s;  
  
    int status;  
    size_t i, iter = 0;  
  
    const size_t n = 2;  
    struct rparams p = {1.0, 10.0};  
    gsl_multiroot_function_fdf f = {&rosenbrock_f,  
                                    &rosenbrock_df,  
                                    &rosenbrock_fdf,  
                                    n, &p};  
  
    double x_init[2] = {-10.0, -5.0};  
    gsl_vector *x = gsl_vector_alloc (n);  
  
    gsl_vector_set (x, 0, x_init[0]);  
    gsl_vector_set (x, 1, x_init[1]);  
  
    T = gsl_multiroot_fdfsolver_gnewton;  
    s = gsl_multiroot_fdfsolver_alloc (T, n);  
    gsl_multiroot_fdfsolver_set (s, &f, x);  
  
    print_state (iter, s);  
  
    do  
    {  
        iter++;  
  
        status = gsl_multiroot_fdfsolver_iterate (s);  
  
        print_state (iter, s);  
  
        if (status)  
            break;  
    }  
}
```

(continues on next page)



(continued from previous page)

```

    status = gsl_multiroot_test_residual (s->f, 1e-7);
}
while (status == GSL_CONTINUE && iter < 1000);

printf ("status = %s\n", gsl_strerror (status));

gsl_multiroot_fdfsolver_free (s);
gsl_vector_free (x);
return 0;
}

```

The addition of derivative information to the `gsl_multiroot_fsolver_hybrids` solver does not make any significant difference to its behavior, since it is able to approximate the Jacobian numerically with sufficient accuracy. To illustrate the behavior of a different derivative solver we switch to `gsl_multiroot_fdfsolver_gnewton`. This is a traditional Newton solver with the constraint that it scales back its step if the full step would lead “uphill”. Here is the output for the `gsl_multiroot_fdfsolver_gnewton` algorithm:

```

iter = 0 x = -10.000 -5.000 f(x) = 1.100e+01 -1.050e+03
iter = 1 x = -4.231 -65.317 f(x) = 5.231e+00 -8.321e+02
iter = 2 x = 1.000 -26.358 f(x) = -8.882e-16 -2.736e+02
iter = 3 x = 1.000 1.000 f(x) = -2.220e-16 -4.441e-15
status = success

```

The convergence is much more rapid, but takes a wide excursion out to the point  $(-4.23, -65.3)$ . This could cause the algorithm to go astray in a realistic application. The hybrid algorithm follows the downhill path to the solution more reliably.

## 38.9 References and Further Reading

The original version of the Hybrid method is described in the following articles by Powell,

- M.J.D. Powell, “A Hybrid Method for Nonlinear Equations” (Chap 6, p 87–114) and “A Fortran Subroutine for Solving systems of Nonlinear Algebraic Equations” (Chap 7, p 115–161), in *Numerical Methods for Nonlinear Algebraic Equations*, P. Rabinowitz, editor. Gordon and Breach, 1970.

The following papers are also relevant to the algorithms described in this section,

- J.J. Moré, M.Y. Cosnard, “Numerical Solution of Nonlinear Equations”, *ACM Transactions on Mathematical Software*, Vol 5, No 1, (1979), p 64–85
- C.G. Broyden, “A Class of Methods for Solving Nonlinear Simultaneous Equations”, *Mathematics of Computation*, Vol 19 (1965), p 577–593
- J.J. Moré, B.S. Garbow, K.E. Hillstom, “Testing Unconstrained Optimization Software”, *ACM Transactions on Mathematical Software*, Vol 7, No 1 (1981), p 17–41



## MULTIDIMENSIONAL MINIMIZATION

This chapter describes routines for finding minima of arbitrary multidimensional functions. The library provides low level components for a variety of iterative minimizers and convergence tests. These can be combined by the user to achieve the desired solution, while providing full access to the intermediate steps of the algorithms. Each class of methods uses the same framework, so that you can switch between minimizers at runtime without needing to recompile your program. Each instance of a minimizer keeps track of its own state, allowing the minimizers to be used in multi-threaded programs. The minimization algorithms can be used to maximize a function by inverting its sign.

The header file `gsl_multimin.h` contains prototypes for the minimization functions and related declarations.

### 39.1 Overview

The problem of multidimensional minimization requires finding a point  $x$  such that the scalar function,

$$f(x_1, \dots, x_n)$$

takes a value which is lower than at any neighboring point. For smooth functions the gradient  $g = \nabla f$  vanishes at the minimum. In general there are no bracketing methods available for the minimization of  $n$ -dimensional functions. The algorithms proceed from an initial guess using a search algorithm which attempts to move in a downhill direction.

Algorithms making use of the gradient of the function perform a one-dimensional line minimisation along this direction until the lowest point is found to a suitable tolerance. The search direction is then updated with local information from the function and its derivatives, and the whole process repeated until the true  $n$ -dimensional minimum is found.

Algorithms which do not require the gradient of the function use different strategies. For example, the Nelder-Mead Simplex algorithm maintains  $n + 1$  trial parameter vectors as the vertices of a  $n$ -dimensional simplex. On each iteration it tries to improve the worst vertex of the simplex by geometrical transformations. The iterations are continued until the overall size of the simplex has decreased sufficiently.

Both types of algorithms use a standard framework. The user provides a high-level driver for the algorithms, and the library provides the individual functions necessary for each of the steps. There are three main phases of the iteration. The steps are,

- initialize minimizer state,  $s$ , for algorithm T
- update  $s$  using the iteration T
- test  $s$  for convergence, and repeat iteration if necessary

Each iteration step consists either of an improvement to the line-minimisation in the current direction or an update to the search direction itself. The state for the minimizers is held in a `gsl_multimin_dfminimizer` struct or a `gsl_multimin_fminimizer` struct.

## 39.2 Caveats

Note that the minimization algorithms can only search for one local minimum at a time. When there are several local minima in the search area, the first minimum to be found will be returned; however it is difficult to predict which of the minima this will be. In most cases, no error will be reported if you try to find a local minimum in an area where there is more than one.

It is also important to note that the minimization algorithms find local minima; there is no way to determine whether a minimum is a global minimum of the function in question.

## 39.3 Initializing the Multidimensional Minimizer

The following function initializes a multidimensional minimizer. The minimizer itself depends only on the dimension of the problem and the algorithm and can be reused for different problems.

type **`gsl_multimin_fdfminimizer`**

This is a workspace for minimizing functions using derivatives.

type **`gsl_multimin_fminimizer`**

This is a workspace for minimizing functions without derivatives.

*`gsl_multimin_fdfminimizer`* \***`gsl_multimin_fdfminimizer_alloc`**(const *`gsl_multimin_fdfminimizer_type`* \*T, size\_t n)

*`gsl_multimin_fminimizer`* \***`gsl_multimin_fminimizer_alloc`**(const *`gsl_multimin_fminimizer_type`* \*T, size\_t n)

This function returns a pointer to a newly allocated instance of a minimizer of type *T* for an *n*-dimension function. If there is insufficient memory to create the minimizer then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

int **`gsl_multimin_fdfminimizer_set`**(*`gsl_multimin_fdfminimizer`* \*s, *`gsl_multimin_function_fdf`* \*fdf, const *`gsl_vector`* \*x, double step\_size, double tol)

int **`gsl_multimin_fminimizer_set`**(*`gsl_multimin_fminimizer`* \*s, *`gsl_multimin_function`* \*f, const *`gsl_vector`* \*x, const *`gsl_vector`* \*step\_size)

The function `gsl_multimin_fdfminimizer_set()` initializes the minimizer *s* to minimize the function *fdf* starting from the initial point *x*. The size of the first trial step is given by *step\_size*. The accuracy of the line minimization is specified by *tol*. The precise meaning of this parameter depends on the method used. Typically the line minimization is considered successful if the gradient of the function *g* is orthogonal to the current search direction *p* to a relative accuracy of *tol*, where  $p \cdot g < tol |p| |g|$ . A *tol* value of 0.1 is suitable for most purposes, since line minimization only needs to be carried out approximately. Note that setting *tol* to zero will force the use of “exact” line-searches, which are extremely expensive.

The function `gsl_multimin_fminimizer_set()` initializes the minimizer *s* to minimize the function *f*, starting from the initial point *x*. The size of the initial trial steps is given in vector *step\_size*. The precise meaning of this parameter depends on the method used.

void **`gsl_multimin_fdfminimizer_free`**(*`gsl_multimin_fdfminimizer`* \*s)

void **`gsl_multimin_fminimizer_free`**(*`gsl_multimin_fminimizer`* \*s)

This function frees all the memory associated with the minimizer *s*.

const char \***`gsl_multimin_fdfminimizer_name`**(const *`gsl_multimin_fdfminimizer`* \*s)

const char \***`gsl_multimin_fminimizer_name`**(const *`gsl_multimin_fminimizer`* \*s)

This function returns a pointer to the name of the minimizer. For example:

```
printf ("s is a '%s' minimizer\n", gsl_multimin_fdfminimizer_name (s));
```

would print something like `s is a 'conjugate_pr' minimizer`.

## 39.4 Providing a function to minimize

You must provide a parametric function of  $n$  variables for the minimizers to operate on. You may also need to provide a routine which calculates the gradient of the function and a third routine which calculates both the function value and the gradient together. In order to allow for general parameters the functions are defined by the following data types:

type **`gsl_multimin_function_fdf`**

This data type defines a general function of  $n$  variables with parameters and the corresponding gradient vector of derivatives,

```
double (* f) (const gsl_vector * x, void * params)
```

this function should return the result  $f(x, params)$  for argument `x` and parameters `params`. If the function cannot be computed, an error value of `GSL_NAN` should be returned.

```
void (* df) (const gsl_vector * x, void * params, gsl_vector * g)
```

this function should store the  $n$ -dimensional gradient

$$g_i = \partial f(x, params) / \partial x_i$$

in the vector `g` for argument `x` and parameters `params`, returning an appropriate error code if the function cannot be computed.

```
void (* fdf) (const gsl_vector * x, void * params, double * f, gsl_vector * g)
```

This function should set the values of the `f` and `g` as above, for arguments `x` and parameters `params`. This function provides an optimization of the separate functions for  $f(x)$  and  $g(x)$ —it is always faster to compute the function and its derivative at the same time.

```
size_t n
```

the dimension of the system, i.e. the number of components of the vectors `x`.

```
void * params
```

a pointer to the parameters of the function.

type **`gsl_multimin_function`**

This data type defines a general function of  $n$  variables with parameters,

```
double (* f) (const gsl_vector * x, void * params)
```

this function should return the result  $f(x, params)$  for argument `x` and parameters `params`. If the function cannot be computed, an error value of `GSL_NAN` should be returned.

```
size_t n
```

the dimension of the system, i.e. the number of components of the vectors `x`.

```
void * params
```

a pointer to the parameters of the function.

The following example function defines a simple two-dimensional paraboloid with five parameters,

```

/* Paraboloid centered on (p[0],p[1]), with
   scale factors (p[2],p[3]) and minimum p[4] */

double
my_f (const gsl_vector *v, void *params)
{
    double x, y;
    double *p = (double *)params;

    x = gsl_vector_get(v, 0);
    y = gsl_vector_get(v, 1);

    return p[2] * (x - p[0]) * (x - p[0]) +
           p[3] * (y - p[1]) * (y - p[1]) + p[4];
}

/* The gradient of f, df = (df/dx, df/dy). */
void
my_df (const gsl_vector *v, void *params,
       gsl_vector *df)
{
    double x, y;
    double *p = (double *)params;

    x = gsl_vector_get(v, 0);
    y = gsl_vector_get(v, 1);

    gsl_vector_set(df, 0, 2.0 * p[2] * (x - p[0]));
    gsl_vector_set(df, 1, 2.0 * p[3] * (y - p[1]));
}

/* Compute both f and df together. */
void
my_fdf (const gsl_vector *x, void *params,
        double *f, gsl_vector *df)
{
    *f = my_f(x, params);
    my_df(x, params, df);
}

```

The function can be initialized using the following code:

```

gsl_multimin_function_fdf my_func;

/* Paraboloid center at (1,2), scale factors (10, 20),
   minimum value 30 */
double p[5] = { 1.0, 2.0, 10.0, 20.0, 30.0 };

my_func.n = 2; /* number of function components */
my_func.f = &my_f;
my_func.df = &my_df;
my_func.fdf = &my_fdf;
my_func.params = (void *)p;

```

## 39.5 Iteration

The following function drives the iteration of each algorithm. The function performs one iteration to update the state of the minimizer. The same function works for all minimizers so that different methods can be substituted at runtime without modifications to the code.

```
int gsl_multimin_fdfminimizer_iterate(gsl_multimin_fdfminimizer *s)
```

```
int gsl_multimin_fminimizer_iterate(gsl_multimin_fminimizer *s)
```

These functions perform a single iteration of the minimizer *s*. If the iteration encounters an unexpected problem then an error code will be returned. The error code `GSL_ENOPROG` signifies that the minimizer is unable to improve on its current estimate, either due to numerical difficulty or because a genuine local minimum has been reached.

The minimizer maintains a current best estimate of the minimum at all times. This information can be accessed with the following auxiliary functions,

```
gsl_vector *gsl_multimin_fdfminimizer_x(const gsl_multimin_fdfminimizer *s)
```

```
gsl_vector *gsl_multimin_fminimizer_x(const gsl_multimin_fminimizer *s)
```

```
double gsl_multimin_fdfminimizer_minimum(const gsl_multimin_fdfminimizer *s)
```

```
double gsl_multimin_fminimizer_minimum(const gsl_multimin_fminimizer *s)
```

```
gsl_vector *gsl_multimin_fdfminimizer_gradient(const gsl_multimin_fdfminimizer *s)
```

```
gsl_vector *gsl_multimin_fdfminimizer_dx(const gsl_multimin_fdfminimizer *s)
```

```
double gsl_multimin_fminimizer_size(const gsl_multimin_fminimizer *s)
```

These functions return the current best estimate of the location of the minimum, the value of the function at that point, its gradient, the last step increment of the estimate, and minimizer specific characteristic size for the minimizer *s*.

```
int gsl_multimin_fdfminimizer_restart(gsl_multimin_fdfminimizer *s)
```

This function resets the minimizer *s* to use the current point as a new starting point.

## 39.6 Stopping Criteria

A minimization procedure should stop when one of the following conditions is true:

- A minimum has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the precision of the current result.

```
int gsl_multimin_test_gradient(const gsl_vector *g, double epsabs)
```

This function tests the norm of the gradient *g* against the absolute tolerance *epsabs*. The gradient of a multi-dimensional function goes to zero at a minimum. The test returns `GSL_SUCCESS` if the following condition is

achieved,

$$|g| < \text{epsabs}$$

and returns `GSL_CONTINUE` otherwise. A suitable choice of `epsabs` can be made from the desired accuracy in the function for small variations in  $x$ . The relationship between these quantities is given by  $\delta f = g \delta x$ .

int **gsl\_multimin\_test\_size**(const double size, double epsabs)

This function tests the minimizer specific characteristic size (if applicable to the used minimizer) against absolute tolerance `epsabs`. The test returns `GSL_SUCCESS` if the size is smaller than tolerance, otherwise `GSL_CONTINUE` is returned.

## 39.7 Algorithms with Derivatives

There are several minimization methods available. The best choice of algorithm depends on the problem. The algorithms described in this section use the value of the function and its gradient at each evaluation point.

type **gsl\_multimin\_fdfminimizer\_type**

This type specifies a minimization algorithm using gradients.

*gsl\_multimin\_fdfminimizer\_type* \***gsl\_multimin\_fdfminimizer\_conjugate\_fr**

This is the Fletcher-Reeves conjugate gradient algorithm. The conjugate gradient algorithm proceeds as a succession of line minimizations. The sequence of search directions is used to build up an approximation to the curvature of the function in the neighborhood of the minimum.

An initial search direction  $p$  is chosen using the gradient, and line minimization is carried out in that direction. The accuracy of the line minimization is specified by the parameter `tol`. The minimum along this line occurs when the function gradient  $g$  and the search direction  $p$  are orthogonal. The line minimization terminates when  $p \cdot g < \text{tol}|p||g|$ . The search direction is updated using the Fletcher-Reeves formula  $p' = g' - \beta p$  where  $\beta = -|g'|^2/|g|^2$ , and the line minimization is then repeated for the new search direction.

*gsl\_multimin\_fdfminimizer\_type* \***gsl\_multimin\_fdfminimizer\_conjugate\_pr**

This is the Polak-Ribiere conjugate gradient algorithm. It is similar to the Fletcher-Reeves method, differing only in the choice of the coefficient  $\beta$ . Both methods work well when the evaluation point is close enough to the minimum of the objective function that it is well approximated by a quadratic hypersurface.

*gsl\_multimin\_fdfminimizer\_type* \***gsl\_multimin\_fdfminimizer\_vector\_bfgs2**

*gsl\_multimin\_fdfminimizer\_type* \***gsl\_multimin\_fdfminimizer\_vector\_bfgs**

These methods use the vector Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. This is a quasi-Newton method which builds up an approximation to the second derivatives of the function  $f$  using the difference between successive gradient vectors. By combining the first and second derivatives the algorithm is able to take Newton-type steps towards the function minimum, assuming quadratic behavior in that region.

The `bfgs2` version of this minimizer is the most efficient version available, and is a faithful implementation of the line minimization scheme described in Fletcher's *Practical Methods of Optimization*, Algorithms 2.6.2 and 2.6.4. It supersedes the original `bfgs` routine and requires substantially fewer function and gradient evaluations. The user-supplied tolerance `tol` corresponds to the parameter  $\sigma$  used by Fletcher. A value of 0.1 is recommended for typical use (larger values correspond to less accurate line searches).

*gsl\_multimin\_fdfminimizer\_type* \***gsl\_multimin\_fdfminimizer\_steepest\_descent**

The steepest descent algorithm follows the downhill gradient of the function at each step. When a downhill step is successful the step-size is increased by a factor of two. If the downhill step leads to a higher function value then the algorithm backtracks and the step size is decreased using the parameter `tol`. A suitable value of `tol` for most applications is 0.1. The steepest descent method is inefficient and is included only for demonstration purposes.



## 39.8 Algorithms without Derivatives

The algorithms described in this section use only the value of the function at each evaluation point.

type **`gsl_multimin_fminimizer_type`**

This type specifies minimization algorithms which do not use gradients.

*`gsl_multimin_fminimizer_type`* \***`gsl_multimin_fminimizer_nmsimplex2`**

*`gsl_multimin_fminimizer_type`* \***`gsl_multimin_fminimizer_nmsimplex`**

These methods use the Simplex algorithm of Nelder and Mead. Starting from the initial vector  $x = p_0$ , the algorithm constructs an additional  $n$  vectors  $p_i$  using the step size vector  $s = \text{step\_size}$  as follows:

$$\begin{aligned} p_0 &= (x_0, x_1, \dots, x_n) \\ p_1 &= (x_0 + s_0, x_1, \dots, x_n) \\ p_2 &= (x_0, x_1 + s_1, \dots, x_n) \\ &\dots = \dots \\ p_n &= (x_0, x_1, \dots, x_n + s_n) \end{aligned}$$

These vectors form the  $n + 1$  vertices of a simplex in  $n$  dimensions. On each iteration the algorithm uses simple geometrical transformations to update the vector corresponding to the highest function value. The geometric transformations are reflection, reflection followed by expansion, contraction and multiple contraction. Using these transformations the simplex moves through the space towards the minimum, where it contracts itself.

After each iteration, the best vertex is returned. Note, that due to the nature of the algorithm not every step improves the current best parameter vector. Usually several iterations are required.

The minimizer-specific characteristic size is calculated as the average distance from the geometrical center of the simplex to all its vertices. This size can be used as a stopping criteria, as the simplex contracts itself near the minimum. The size is returned by the function `gsl_multimin_fminimizer_size()`.

The `gsl_multimin_fminimizer_nmsimplex2` version of this minimiser is a new  $O(N)$  operations implementation of the earlier  $O(N^2)$  operations `gsl_multimin_fminimizer_nmsimplex` minimiser. It uses the same underlying algorithm, but the simplex updates are computed more efficiently for high-dimensional problems. In addition, the size of simplex is calculated as the RMS distance of each vertex from the center rather than the mean distance, allowing a linear update of this quantity on each step. The memory usage is  $O(N^2)$  for both algorithms.

*`gsl_multimin_fminimizer_type`* \***`gsl_multimin_fminimizer_nmsimplex2rand`**

This method is a variant of `gsl_multimin_fminimizer_nmsimplex2` which initialises the simplex around the starting point  $x$  using a randomly-oriented set of basis vectors instead of the fixed coordinate axes. The final dimensions of the simplex are scaled along the coordinate axes by the vector `step_size`. The randomisation uses a simple deterministic generator so that repeated calls to `gsl_multimin_fminimizer_set()` for a given solver object will vary the orientation in a well-defined way.

## 39.9 Examples

This example program finds the minimum of the *paraboloid function* defined earlier. The location of the minimum is offset from the origin in  $x$  and  $y$ , and the function value at the minimum is non-zero. The main program is given below, it requires the example function given earlier in this chapter.

```
int
main (void)
{
    size_t iter = 0;
    int status;

    const gsl_multimin_fdfminimizer_type *T;
    gsl_multimin_fdfminimizer *s;

    /* Position of the minimum (1,2), scale factors
       10,20, height 30. */
    double par[5] = { 1.0, 2.0, 10.0, 20.0, 30.0 };

    gsl_vector *x;
    gsl_multimin_function_fdf my_func;

    my_func.n = 2;
    my_func.f = my_f;
    my_func.df = my_df;
    my_func.fdf = my_fdf;
    my_func.params = par;

    /* Starting point, x = (5,7) */
    x = gsl_vector_alloc (2);
    gsl_vector_set (x, 0, 5.0);
    gsl_vector_set (x, 1, 7.0);

    T = gsl_multimin_fdfminimizer_conjugate_fr;
    s = gsl_multimin_fdfminimizer_alloc (T, 2);

    gsl_multimin_fdfminimizer_set (s, &my_func, x, 0.01, 1e-4);

    do
    {
        iter++;
        status = gsl_multimin_fdfminimizer_iterate (s);

        if (status)
            break;

        status = gsl_multimin_test_gradient (s->gradient, 1e-3);

        if (status == GSL_SUCCESS)
            printf ("Minimum found at:\n");

        printf ("%5d %.5f %.5f %10.5f\n", iter,
                gsl_vector_get (s->x, 0),
```

(continues on next page)

(continued from previous page)

```

        gsl_vector_get (s->x, 1),
        s->f);

    }
    while (status == GSL_CONTINUE && iter < 100);

    gsl_multimin_fdfminimizer_free (s);
    gsl_vector_free (x);

    return 0;
}

```

The initial step-size is chosen as 0.01, a conservative estimate in this case, and the line minimization parameter is set at 0.0001. The program terminates when the norm of the gradient has been reduced below 0.001. The output of the program is shown below,

	x	y	f
1	4.99629	6.99072	687.84780
2	4.98886	6.97215	683.55456
3	4.97400	6.93501	675.01278
4	4.94429	6.86073	658.10798
5	4.88487	6.71217	625.01340
6	4.76602	6.41506	561.68440
7	4.52833	5.82083	446.46694
8	4.05295	4.63238	261.79422
9	3.10219	2.25548	75.49762
10	2.85185	1.62963	67.03704
11	2.19088	1.76182	45.31640
12	0.86892	2.02622	30.18555
Minimum found at:			
13	1.00000	2.00000	30.00000

Note that the algorithm gradually increases the step size as it successfully moves downhill, as can be seen by plotting the successive points in [Fig. 39.1](#).

The conjugate gradient algorithm finds the minimum on its second direction because the function is purely quadratic. Additional iterations would be needed for a more complicated function.

Here is another example using the Nelder-Mead Simplex algorithm to minimize the same example object function, as above.

```

int
main(void)
{
    double par[5] = {1.0, 2.0, 10.0, 20.0, 30.0};

    const gsl_multimin_fminimizer_type *T =
        gsl_multimin_fminimizer_rmsimplex2;
    gsl_multimin_fminimizer *s = NULL;
    gsl_vector *ss, *x;
    gsl_multimin_function minex_func;

    size_t iter = 0;

```

(continues on next page)

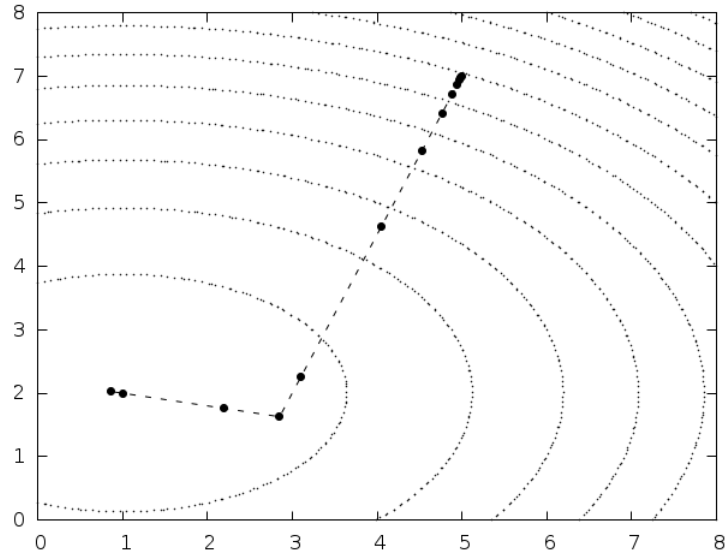


Fig. 39.1: Function contours with path taken by minimization algorithm

(continued from previous page)

```

int status;
double size;

/* Starting point */
x = gsl_vector_alloc (2);
gsl_vector_set (x, 0, 5.0);
gsl_vector_set (x, 1, 7.0);

/* Set initial step sizes to 1 */
ss = gsl_vector_alloc (2);
gsl_vector_set_all (ss, 1.0);

/* Initialize method and iterate */
minex_func.n = 2;
minex_func.f = my_f;
minex_func.params = par;

s = gsl_multimin_fminimizer_alloc (T, 2);
gsl_multimin_fminimizer_set (s, &minex_func, x, ss);

do
{
    iter++;
    status = gsl_multimin_fminimizer_iterate(s);

    if (status)
        break;

    size = gsl_multimin_fminimizer_size (s);
    status = gsl_multimin_test_size (size, 1e-2);
}

```

(continues on next page)

(continued from previous page)

```

    if (status == GSL_SUCCESS)
    {
        printf ("converged to minimum at\n");
    }

    printf ("%5d %10.3e %10.3e f() = %7.3f size = %.3f\n",
            iter,
            gsl_vector_get (s->x, 0),
            gsl_vector_get (s->x, 1),
            s->fval, size);
}
while (status == GSL_CONTINUE && iter < 100);

gsl_vector_free(x);
gsl_vector_free(ss);
gsl_multimin_fminimizer_free (s);

return status;
}

```

The minimum search stops when the Simplex size drops to 0.01. The output is shown below.

```

 1  6.500e+00  5.000e+00 f() = 512.500 size = 1.130
 2  5.250e+00  4.000e+00 f() = 290.625 size = 1.409
 3  5.250e+00  4.000e+00 f() = 290.625 size = 1.409
 4  5.500e+00  1.000e+00 f() = 252.500 size = 1.409
 5  2.625e+00  3.500e+00 f() = 101.406 size = 1.847
 6  2.625e+00  3.500e+00 f() = 101.406 size = 1.847
 7  0.000e+00  3.000e+00 f() = 60.000 size = 1.847
 8  2.094e+00  1.875e+00 f() = 42.275 size = 1.321
 9  2.578e-01  1.906e+00 f() = 35.684 size = 1.069
10  5.879e-01  2.445e+00 f() = 35.664 size = 0.841
11  1.258e+00  2.025e+00 f() = 30.680 size = 0.476
12  1.258e+00  2.025e+00 f() = 30.680 size = 0.367
13  1.093e+00  1.849e+00 f() = 30.539 size = 0.300
14  8.830e-01  2.004e+00 f() = 30.137 size = 0.172
15  8.830e-01  2.004e+00 f() = 30.137 size = 0.126
16  9.582e-01  2.060e+00 f() = 30.090 size = 0.106
17  1.022e+00  2.004e+00 f() = 30.005 size = 0.063
18  1.022e+00  2.004e+00 f() = 30.005 size = 0.043
19  1.022e+00  2.004e+00 f() = 30.005 size = 0.043
20  1.022e+00  2.004e+00 f() = 30.005 size = 0.027
21  1.022e+00  2.004e+00 f() = 30.005 size = 0.022
22  9.920e-01  1.997e+00 f() = 30.001 size = 0.016
23  9.920e-01  1.997e+00 f() = 30.001 size = 0.013
converged to minimum at
24  9.920e-01  1.997e+00 f() = 30.001 size = 0.008

```

The simplex size first increases, while the simplex moves towards the minimum. After a while the size begins to decrease as the simplex contracts around the minimum.

## 39.10 References and Further Reading

The conjugate gradient and BFGS methods are described in detail in the following book,

- R. Fletcher, *Practical Methods of Optimization (Second Edition)* Wiley (1987), ISBN 0471915475.

A brief description of multidimensional minimization algorithms and more recent references can be found in,

- C.W. Ueberhuber, *Numerical Computation (Volume 2)*, Chapter 14, Section 4.4 “Minimization Methods”, p.: 325–335, Springer (1997), ISBN 3-540-62057-5.

The simplex algorithm is described in the following paper,

- J.A. Nelder and R. Mead, *A simplex method for function minimization*, Computer Journal vol.: 7 (1965), 308–313.

## LINEAR LEAST-SQUARES FITTING

This chapter describes routines for performing least squares fits to experimental data using linear combinations of functions. The data may be weighted or unweighted, i.e. with known or unknown errors. For weighted data the functions compute the best fit parameters and their associated covariance matrix. For unweighted data the covariance matrix is estimated from the scatter of the points, giving a variance-covariance matrix.

The functions are divided into separate versions for simple one- or two-parameter regression and multiple-parameter fits.

### 40.1 Overview

Least-squares fits are found by minimizing  $\chi^2$  (chi-squared), the weighted sum of squared residuals over  $n$  experimental datapoints  $(x_i, y_i)$  for the model  $Y(c, x)$ ,

$$\chi^2 = \sum_i w_i (y_i - Y(c, x_i))^2$$

The  $p$  parameters of the model are  $c = \{c_0, c_1, \dots\}$ . The weight factors  $w_i$  are given by  $w_i = 1/\sigma_i^2$  where  $\sigma_i$  is the experimental error on the data-point  $y_i$ . The errors are assumed to be Gaussian and uncorrelated. For unweighted data the chi-squared sum is computed without any weight factors.

The fitting routines return the best-fit parameters  $c$  and their  $p \times p$  covariance matrix. The covariance matrix measures the statistical errors on the best-fit parameters resulting from the errors on the data,  $\sigma_i$ , and is defined as

$$C_{ab} = \langle \delta c_a \delta c_b \rangle$$

where  $\langle \rangle$  denotes an average over the Gaussian error distributions of the underlying datapoints.

The covariance matrix is calculated by error propagation from the data errors  $\sigma_i$ . The change in a fitted parameter  $\delta c_a$  caused by a small change in the data  $\delta y_i$  is given by

$$\delta c_a = \sum_i \frac{\partial c_a}{\partial y_i} \delta y_i$$

allowing the covariance matrix to be written in terms of the errors on the data,

$$C_{ab} = \sum_{i,j} \frac{\partial c_a}{\partial y_i} \frac{\partial c_b}{\partial y_j} \langle \delta y_i \delta y_j \rangle$$

For uncorrelated data the fluctuations of the underlying datapoints satisfy

$$\langle \delta y_i \delta y_j \rangle = \sigma_i^2 \delta_{ij}$$

giving a corresponding parameter covariance matrix of

$$C_{ab} = \sum_i \frac{1}{w_i} \frac{\partial c_a}{\partial y_i} \frac{\partial c_b}{\partial y_i}$$

When computing the covariance matrix for unweighted data, i.e. data with unknown errors, the weight factors  $w_i$  in this sum are replaced by the single estimate  $w = 1/\sigma^2$ , where  $\sigma^2$  is the computed variance of the residuals about the best-fit model,  $\sigma^2 = \sum (y_i - Y(c, x_i))^2 / (n - p)$ . This is referred to as the *variance-covariance matrix*.

The standard deviations of the best-fit parameters are given by the square root of the corresponding diagonal elements of the covariance matrix,  $\sigma_{c_a} = \sqrt{C_{aa}}$ . The correlation coefficient of the fit parameters  $c_a$  and  $c_b$  is given by  $\rho_{ab} = C_{ab} / \sqrt{C_{aa}C_{bb}}$ .

## 40.2 Linear regression

The functions in this section are used to fit simple one or two parameter linear regression models. The functions are declared in the header file `gsl_fit.h`.

### 40.2.1 Linear regression with a constant term

The functions described in this section can be used to perform least-squares fits to a straight line model,  $Y(c, x) = c_0 + c_1x$ .

int **gsl\_fit\_linear**(const double \*x, const size\_t xstride, const double \*y, const size\_t ystride, size\_t n, double \*c0, double \*c1, double \*cov00, double \*cov01, double \*cov11, double \*sumsq)

This function computes the best-fit linear regression coefficients (`c0`, `c1`) of the model  $Y = c_0 + c_1X$  for the dataset  $(x, y)$ , two vectors of length  $n$  with strides `xstride` and `ystride`. The errors on  $y$  are assumed unknown so the variance-covariance matrix for the parameters (`c0`, `c1`) is estimated from the scatter of the points around the best-fit line and returned via the parameters (`cov00`, `cov01`, `cov11`). The sum of squares of the residuals from the best-fit line is returned in `sumsq`. Note: the correlation coefficient of the data can be computed using `gsl_stats_correlation()`, it does not depend on the fit.

int **gsl\_fit\_wlinear**(const double \*x, const size\_t xstride, const double \*w, const size\_t wstride, const double \*y, const size\_t ystride, size\_t n, double \*c0, double \*c1, double \*cov00, double \*cov01, double \*cov11, double \*chisq)

This function computes the best-fit linear regression coefficients (`c0`, `c1`) of the model  $Y = c_0 + c_1X$  for the weighted dataset  $(x, y)$ , two vectors of length  $n$  with strides `xstride` and `ystride`. The vector  $w$ , of length  $n$  and stride `wstride`, specifies the weight of each datapoint. The weight is the reciprocal of the variance for each datapoint in  $y$ .

The covariance matrix for the parameters (`c0`, `c1`) is computed using the weights and returned via the parameters (`cov00`, `cov01`, `cov11`). The weighted sum of squares of the residuals from the best-fit line,  $\chi^2$ , is returned in `chisq`.

int **gsl\_fit\_linear\_est**(double x, double c0, double c1, double cov00, double cov01, double cov11, double \*y, double \*y\_err)

This function uses the best-fit linear regression coefficients `c0`, `c1` and their covariance `cov00`, `cov01`, `cov11` to compute the fitted function  $y$  and its standard deviation `y_err` for the model  $Y = c_0 + c_1X$  at the point  $x$ .



## 40.2.2 Linear regression without a constant term

The functions described in this section can be used to perform least-squares fits to a straight line model without a constant term,  $Y = c_1 X$ .

```
int gsl_fit_mul(const double *x, const size_t xstride, const double *y, const size_t ystride, size_t n, double *c1,
               double *cov11, double *sumsq)
```

This function computes the best-fit linear regression coefficient `c1` of the model  $Y = c_1 X$  for the datasets (`x`, `y`), two vectors of length `n` with strides `xstride` and `ystride`. The errors on `y` are assumed unknown so the variance of the parameter `c1` is estimated from the scatter of the points around the best-fit line and returned via the parameter `cov11`. The sum of squares of the residuals from the best-fit line is returned in `sumsq`.

```
int gsl_fit_wmul(const double *x, const size_t xstride, const double *w, const size_t wstride, const double *y,
                const size_t ystride, size_t n, double *c1, double *cov11, double *sumsq)
```

This function computes the best-fit linear regression coefficient `c1` of the model  $Y = c_1 X$  for the weighted datasets (`x`, `y`), two vectors of length `n` with strides `xstride` and `ystride`. The vector `w`, of length `n` and stride `wstride`, specifies the weight of each datapoint. The weight is the reciprocal of the variance for each datapoint in `y`.

The variance of the parameter `c1` is computed using the weights and returned via the parameter `cov11`. The weighted sum of squares of the residuals from the best-fit line,  $\chi^2$ , is returned in `chisq`.

```
int gsl_fit_mul_est(double x, double c1, double cov11, double *y, double *y_err)
```

This function uses the best-fit linear regression coefficient `c1` and its covariance `cov11` to compute the fitted function `y` and its standard deviation `y_err` for the model  $Y = c_1 X$  at the point `x`.

## 40.3 Multi-parameter regression

This section describes routines which perform least squares fits to a linear model by minimizing the cost function

$$\chi^2 = \sum_i w_i (y_i - \sum_j X_{ij} c_j)^2 = \|y - Xc\|_W^2$$

where  $y$  is a vector of  $n$  observations,  $X$  is an  $n$ -by- $p$  matrix of predictor variables,  $c$  is a vector of the  $p$  unknown best-fit parameters to be estimated, and  $\|r\|_W^2 = r^T W r$ . The matrix  $W = \text{diag}(w_1, w_2, \dots, w_n)$  defines the weights or uncertainties of the observation vector.

This formulation can be used for fits to any number of functions and/or variables by preparing the  $n$ -by- $p$  matrix  $X$  appropriately. For example, to fit to a  $p$ -th order polynomial in  $x$ , use the following matrix,

$$X_{ij} = x_i^j$$

where the index  $i$  runs over the observations and the index  $j$  runs from 0 to  $p - 1$ .

To fit to a set of  $p$  sinusoidal functions with fixed frequencies  $\omega_1, \omega_2, \dots, \omega_p$ , use,

$$X_{ij} = \sin(\omega_j x_i)$$

To fit to  $p$  independent variables  $x_1, x_2, \dots, x_p$ , use,

$$X_{ij} = x_j(i)$$

where  $x_j(i)$  is the  $i$ -th value of the predictor variable  $x_j$ .

The solution of the general linear least-squares system requires an additional working space for intermediate results, such as the singular value decomposition of the matrix  $X$ .

These functions are declared in the header file `gsl_multifit.h`.

type **gsl\_multifit\_linear\_workspace**

This workspace contains internal variables for fitting multi-parameter models.

*gsl\_multifit\_linear\_workspace* \***gsl\_multifit\_linear\_alloc**(const size\_t n, const size\_t p)

This function allocates a workspace for fitting a model to a maximum of  $n$  observations using a maximum of  $p$  parameters. The user may later supply a smaller least squares system if desired. The size of the workspace is  $O(np + p^2)$ .

void **gsl\_multifit\_linear\_free**(*gsl\_multifit\_linear\_workspace* \*work)

This function frees the memory associated with the workspace  $w$ .

int **gsl\_multifit\_linear\_svd**(const *gsl\_matrix* \*X, *gsl\_multifit\_linear\_workspace* \*work)

This function performs a singular value decomposition of the matrix  $X$  and stores the SVD factors internally in *work*.

int **gsl\_multifit\_linear\_bsvd**(const *gsl\_matrix* \*X, *gsl\_multifit\_linear\_workspace* \*work)

This function performs a singular value decomposition of the matrix  $X$  and stores the SVD factors internally in *work*. The matrix  $X$  is first balanced by applying column scaling factors to improve the accuracy of the singular values.

int **gsl\_multifit\_linear**(const *gsl\_matrix* \*X, const *gsl\_vector* \*y, *gsl\_vector* \*c, *gsl\_matrix* \*cov, double \*chisq, *gsl\_multifit\_linear\_workspace* \*work)

This function computes the best-fit parameters  $c$  of the model  $y = Xc$  for the observations  $y$  and the matrix of predictor variables  $X$ , using the preallocated workspace provided in *work*. The  $p$ -by- $p$  variance-covariance matrix of the model parameters *cov* is set to  $\sigma^2(X^T X)^{-1}$ , where  $\sigma$  is the standard deviation of the fit residuals. The sum of squares of the residuals from the best-fit,  $\chi^2$ , is returned in *chisq*. If the coefficient of determination is desired, it can be computed from the expression  $R^2 = 1 - \chi^2/TSS$ , where the total sum of squares (TSS) of the observations  $y$  may be computed from *gsl\_stats\_tss*() .

The best-fit is found by singular value decomposition of the matrix  $X$  using the modified Golub-Reinsch SVD algorithm, with column scaling to improve the accuracy of the singular values. Any components which have zero singular value (to machine precision) are discarded from the fit.

int **gsl\_multifit\_linear\_tsvd**(const *gsl\_matrix* \*X, const *gsl\_vector* \*y, const double tol, *gsl\_vector* \*c, *gsl\_matrix* \*cov, double \*chisq, size\_t \*rank, *gsl\_multifit\_linear\_workspace* \*work)

This function computes the best-fit parameters  $c$  of the model  $y = Xc$  for the observations  $y$  and the matrix of predictor variables  $X$ , using a truncated SVD expansion. Singular values which satisfy  $s_i \leq tol \times s_0$  are discarded from the fit, where  $s_0$  is the largest singular value. The  $p$ -by- $p$  variance-covariance matrix of the model parameters *cov* is set to  $\sigma^2(X^T X)^{-1}$ , where  $\sigma$  is the standard deviation of the fit residuals. The sum of squares of the residuals from the best-fit,  $\chi^2$ , is returned in *chisq*. The effective rank (number of singular values used in solution) is returned in *rank*. If the coefficient of determination is desired, it can be computed from the expression  $R^2 = 1 - \chi^2/TSS$ , where the total sum of squares (TSS) of the observations  $y$  may be computed from *gsl\_stats\_tss*() .

int **gsl\_multifit\_wlinear**(const *gsl\_matrix* \*X, const *gsl\_vector* \*w, const *gsl\_vector* \*y, *gsl\_vector* \*c, *gsl\_matrix* \*cov, double \*chisq, *gsl\_multifit\_linear\_workspace* \*work)

This function computes the best-fit parameters  $c$  of the weighted model  $y = Xc$  for the observations  $y$  with weights  $w$  and the matrix of predictor variables  $X$ , using the preallocated workspace provided in *work*. The  $p$ -by- $p$  covariance matrix of the model parameters *cov* is computed as  $(X^T W X)^{-1}$ . The weighted sum of squares of the residuals from the best-fit,  $\chi^2$ , is returned in *chisq*. If the coefficient of determination is desired, it can be computed from the expression  $R^2 = 1 - \chi^2/WTSS$ , where the weighted total sum of squares (WTSS) of the observations  $y$  may be computed from *gsl\_stats\_wtss*() .

int **gsl\_multifit\_wlinear\_tsvd**(const *gsl\_matrix* \*X, const *gsl\_vector* \*w, const *gsl\_vector* \*y, const double tol, *gsl\_vector* \*c, *gsl\_matrix* \*cov, double \*chisq, size\_t \*rank, *gsl\_multifit\_linear\_workspace* \*work)

This function computes the best-fit parameters  $c$  of the weighted model  $y = Xc$  for the observations  $y$  with

weights  $w$  and the matrix of predictor variables  $X$ , using a truncated SVD expansion. Singular values which satisfy  $s_i \leq tol \times s_0$  are discarded from the fit, where  $s_0$  is the largest singular value. The  $p$ -by- $p$  covariance matrix of the model parameters  $cov$  is computed as  $(X^T W X)^{-1}$ . The weighted sum of squares of the residuals from the best-fit,  $\chi^2$ , is returned in *chisq*. The effective rank of the system (number of singular values used in the solution) is returned in *rank*. If the coefficient of determination is desired, it can be computed from the expression  $R^2 = 1 - \chi^2 / WTSS$ , where the weighted total sum of squares (WTSS) of the observations  $y$  may be computed from *gsl\_stats\_wtss()*.

```
int gsl_multifit_linear_est(const gsl_vector *x, const gsl_vector *c, const gsl_matrix *cov, double *y, double *y_err)
```

This function uses the best-fit multilinear regression coefficients  $c$  and their covariance matrix  $cov$  to compute the fitted function value  $y$  and its standard deviation  $y\_err$  for the model  $y = x.c$  at the point  $x$ .

```
int gsl_multifit_linear_residuals(const gsl_matrix *X, const gsl_vector *y, const gsl_vector *c, gsl_vector *r)
```

This function computes the vector of residuals  $r = y - Xc$  for the observations  $y$ , coefficients  $c$  and matrix of predictor variables  $X$ .

```
size_t gsl_multifit_linear_rank(const double tol, const gsl_multifit_linear_workspace *work)
```

This function returns the rank of the matrix  $X$  which must first have its singular value decomposition computed. The rank is computed by counting the number of singular values  $\sigma_j$  which satisfy  $\sigma_j > tol \times \sigma_0$ , where  $\sigma_0$  is the largest singular value.

## 40.4 Regularized regression

Ordinary weighted least squares models seek a solution vector  $c$  which minimizes the residual

$$\chi^2 = \|y - Xc\|_W^2$$

where  $y$  is the  $n$ -by-1 observation vector,  $X$  is the  $n$ -by- $p$  design matrix,  $c$  is the  $p$ -by-1 solution vector,  $W = \text{diag}(w_1, \dots, w_n)$  is the data weighting matrix, and  $\|r\|_W^2 = r^T W r$ . In cases where the least squares matrix  $X$  is ill-conditioned, small perturbations (ie: noise) in the observation vector could lead to widely different solution vectors  $c$ . One way of dealing with ill-conditioned matrices is to use a “truncated SVD” in which small singular values, below some given tolerance, are discarded from the solution. The truncated SVD method is available using the functions *gsl\_multifit\_linear\_tsvd()* and *gsl\_multifit\_wlinear\_tsvd()*. Another way to help solve ill-posed problems is to include a regularization term in the least squares minimization

$$\chi^2 = \|y - Xc\|_W^2 + \lambda^2 \|Lc\|^2$$

for a suitably chosen regularization parameter  $\lambda$  and matrix  $L$ . This type of regularization is known as Tikhonov, or ridge, regression. In some applications,  $L$  is chosen as the identity matrix, giving preference to solution vectors  $c$  with smaller norms. Including this regularization term leads to the explicit “normal equations” solution

$$c = (X^T W X + \lambda^2 L^T L)^{-1} X^T W y$$

which reduces to the ordinary least squares solution when  $L = 0$ . In practice, it is often advantageous to transform a regularized least squares system into the form

$$\chi^2 = \|\tilde{y} - \tilde{X}\tilde{c}\|^2 + \lambda^2 \|\tilde{c}\|^2$$

This is known as the Tikhonov “standard form” and has the normal equations solution

$$\tilde{c} = (\tilde{X}^T \tilde{X} + \lambda^2 I)^{-1} \tilde{X}^T \tilde{y}$$

For an  $m$ -by- $p$  matrix  $L$  which is full rank and has  $m \geq p$  (ie:  $L$  is square or has more rows than columns), we can calculate the “thin” QR decomposition of  $L$ , and note that  $\|Lc\| = \|Rc\|$  since the  $Q$  factor will not change the norm. Since  $R$  is  $p$ -by- $p$ , we can then use the transformation

$$\begin{aligned}\tilde{X} &= W^{\frac{1}{2}} X R^{-1} \\ \tilde{y} &= W^{\frac{1}{2}} y \\ \tilde{c} &= Rc\end{aligned}$$

to achieve the standard form. For a rectangular matrix  $L$  with  $m < p$ , a more sophisticated approach is needed (see Hansen 1998, chapter 2.3). In practice, the normal equations solution above is not desirable due to numerical instabilities, and so the system is solved using the singular value decomposition of the matrix  $\tilde{X}$ . The matrix  $L$  is often chosen as the identity matrix, or as a first or second finite difference operator, to ensure a smoothly varying coefficient vector  $c$ , or as a diagonal matrix to selectively damp each model parameter differently. If  $L \neq I$ , the user must first convert the least squares problem to standard form using `gsl_multifit_linear_stdform1()` or `gsl_multifit_linear_stdform2()`, solve the system, and then backtransform the solution vector to recover the solution of the original problem (see `gsl_multifit_linear_genform1()` and `gsl_multifit_linear_genform2()`).

In many regularization problems, care must be taken when choosing the regularization parameter  $\lambda$ . Since both the residual norm  $\|y - Xc\|$  and solution norm  $\|Lc\|$  are being minimized, the parameter  $\lambda$  represents a tradeoff between minimizing either the residuals or the solution vector. A common tool for visualizing the compromise between the minimization of these two quantities is known as the L-curve. The L-curve is a log-log plot of the residual norm  $\|y - Xc\|$  on the horizontal axis and the solution norm  $\|Lc\|$  on the vertical axis. This curve nearly always has an L shaped appearance, with a distinct corner separating the horizontal and vertical sections of the curve. The regularization parameter corresponding to this corner is often chosen as the optimal value. GSL provides routines to calculate the L-curve for all relevant regularization parameters as well as locating the corner.

Another method of choosing the regularization parameter is known as Generalized Cross Validation (GCV). This method is based on the idea that if an arbitrary element  $y_i$  is left out of the right hand side, the resulting regularized solution should predict this element accurately. This leads to choosing the parameter  $\lambda$  which minimizes the GCV function

$$G(\lambda) = \frac{\|y - Xc_\lambda\|^2}{\text{Tr}(I_n - XX_\lambda^T)^2}$$

where  $X_\lambda^T$  is the matrix which relates the solution  $c_\lambda$  to the right hand side  $y$ , ie:  $c_\lambda = X_\lambda^T y$ . GSL provides routines to compute the GCV curve and its minimum.

For most applications, the steps required to solve a regularized least squares problem are as follows:

1. Construct the least squares system  $(X, y, W, L)$
2. Transform the system to standard form  $(\tilde{X}, \tilde{y})$ . This step can be skipped if  $L = I_p$  and  $W = I_n$ .
3. Calculate the SVD of  $\tilde{X}$ .
4. Determine an appropriate regularization parameter  $\lambda$  (using for example L-curve or GCV analysis).
5. Solve the standard form system using the chosen  $\lambda$  and the SVD of  $\tilde{X}$ .
6. Backtransform the standard form solution  $\tilde{c}$  to recover the original solution vector  $c$ .

```
int gsl_multifit_linear_stdform1(const gsl_vector *L, const gsl_matrix *X, const gsl_vector *y, gsl_matrix
    *Xs, gsl_vector *ys, gsl_multifit_linear_workspace *work)
```

```
int gsl_multifit_linear_wstdform1(const gsl_vector *L, const gsl_matrix *X, const gsl_vector *w, const
    gsl_vector *y, gsl_matrix *Xs, gsl_vector *ys,
    gsl_multifit_linear_workspace *work)
```

These functions define a regularization matrix  $L = \text{diag}(l_0, l_1, \dots, l_{p-1})$ . The diagonal matrix element  $l_i$  is provided by the  $i$ -th element of the input vector  $L$ . The  $n$ -by- $p$  least squares matrix  $X$  and vector  $y$  of length  $n$  are

then converted to standard form as described above and the parameters  $(\tilde{X}, \tilde{y})$  are stored in *Xs* and *ys* on output. *Xs* and *ys* have the same dimensions as *X* and *y*. Optional data weights may be supplied in the vector *w* of length *n*. In order to apply this transformation,  $L^{-1}$  must exist and so none of the  $l_i$  may be zero. After the standard form system has been solved, use `gsl_multifit_linear_genform1()` to recover the original solution vector. It is allowed to have  $X = Xs$  and  $y = ys$  for an in-place transform. In order to perform a weighted regularized fit with  $L = I$ , the user may call `gsl_multifit_linear_applyW()` to convert to standard form.

```
int gsl_multifit_linear_L_decomp(gsl_matrix *L, gsl_vector *tau)
```

This function factors the  $m$ -by- $p$  regularization matrix  $L$  into a form needed for the later transformation to standard form.  $L$  may have any number of rows  $m$ . If  $m \geq p$  the QR decomposition of  $L$  is computed and stored in  $L$  on output. If  $m < p$ , the QR decomposition of  $L^T$  is computed and stored in  $L$  on output. On output, the Householder scalars are stored in the vector *tau* of size  $\text{MIN}(m, p)$ . These outputs will be used by `gsl_multifit_linear_wstdform2()` to complete the transformation to standard form.

```
int gsl_multifit_linear_stdform2(const gsl_matrix *LQR, const gsl_vector *Ltau, const gsl_matrix *X, const
    gsl_vector *y, gsl_matrix *Xs, gsl_vector *ys, gsl_matrix *M,
    gsl_multifit_linear_workspace *work)
```

```
int gsl_multifit_linear_wstdform2(const gsl_matrix *LQR, const gsl_vector *Ltau, const gsl_matrix *X,
    const gsl_vector *w, const gsl_vector *y, gsl_matrix *Xs, gsl_vector *ys,
    gsl_matrix *M, gsl_multifit_linear_workspace *work)
```

These functions convert the least squares system  $(X, y, W, L)$  to standard form  $(\tilde{X}, \tilde{y})$  which are stored in *Xs* and *ys* respectively. The  $m$ -by- $p$  regularization matrix  $L$  is specified by the inputs *LQR* and *Ltau*, which are outputs from `gsl_multifit_linear_L_decomp()`. The dimensions of the standard form parameters  $(\tilde{X}, \tilde{y})$  depend on whether  $m$  is larger or less than  $p$ . For  $m \geq p$ , *Xs* is  $n$ -by- $p$ , *ys* is  $n$ -by-1, and *M* is not used. For  $m < p$ , *Xs* is  $(n-p+m)$ -by- $m$ , *ys* is  $(n-p+m)$ -by-1, and *M* is additional  $n$ -by- $p$  workspace, which is required to recover the original solution vector after the system has been solved (see `gsl_multifit_linear_genform2()`). Optional data weights may be supplied in the vector *w* of length  $n$ , where  $W = \text{diag}(w)$ .

```
int gsl_multifit_linear_solve(const double lambda, const gsl_matrix *Xs, const gsl_vector *ys, gsl_vector
    *cs, double *rnorm, double *snorm, gsl_multifit_linear_workspace *work)
```

This function computes the regularized best-fit parameters  $\tilde{c}$  which minimize the cost function  $\chi^2 = \|\tilde{y} - \tilde{X}\tilde{c}\|^2 + \lambda^2 \|\tilde{c}\|^2$  which is in standard form. The least squares system must therefore be converted to standard form prior to calling this function. The observation vector  $\tilde{y}$  is provided in *ys* and the matrix of predictor variables  $\tilde{X}$  in *Xs*. The solution vector  $\tilde{c}$  is returned in *cs*, which has length  $\min(m, p)$ . The SVD of *Xs* must be computed prior to calling this function, using `gsl_multifit_linear_svd()`. The regularization parameter  $\lambda$  is provided in *lambda*. The residual norm  $\|\tilde{y} - \tilde{X}\tilde{c}\| = \|y - Xc\|_W$  is returned in *rnorm*. The solution norm  $\|\tilde{c}\| = \|Lc\|$  is returned in *snorm*.

```
int gsl_multifit_linear_genform1(const gsl_vector *L, const gsl_vector *cs, gsl_vector *c,
    gsl_multifit_linear_workspace *work)
```

After a regularized system has been solved with  $L = \text{diag}(l_0, l_1, \dots, l_{p-1})$ , this function backtransforms the standard form solution vector *cs* to recover the solution vector of the original problem *c*. The diagonal matrix elements  $l_i$  are provided in the vector *L*. It is allowed to have  $c = cs$  for an in-place transform.

```
int gsl_multifit_linear_genform2(const gsl_matrix *LQR, const gsl_vector *Ltau, const gsl_matrix *X, const
    gsl_vector *y, const gsl_vector *cs, const gsl_matrix *M, gsl_vector *c,
    gsl_multifit_linear_workspace *work)
```

```
int gsl_multifit_linear_wgenform2(const gsl_matrix *LQR, const gsl_vector *Ltau, const gsl_matrix *X,
    const gsl_vector *w, const gsl_vector *y, const gsl_vector *cs, const
    gsl_matrix *M, gsl_vector *c, gsl_multifit_linear_workspace *work)
```

After a regularized system has been solved with a general rectangular matrix  $L$ , specified by  $(LQR, Ltau)$ , this function backtransforms the standard form solution *cs* to recover the solution vector of the original problem, which is stored in *c*, of length  $p$ . The original least squares matrix and observation vector are provided in *X*



and  $y$  respectively.  $M$  is the matrix computed by `gsl_multifit_linear_stdform2()`. For weighted fits, the weight vector  $w$  must also be supplied.

```
int gsl_multifit_linear_applyW(const gsl_matrix *X, const gsl_vector *w, const gsl_vector *y, gsl_matrix
                               *WX, gsl_vector *Wy)
```

For weighted least squares systems with  $L = I$ , this function may be used to convert the system to standard form by applying the weight matrix  $W = \text{diag}(w)$  to the least squares matrix  $X$  and observation vector  $y$ . On output,  $WX$  is equal to  $W^{1/2}X$  and  $Wy$  is equal to  $W^{1/2}y$ . It is allowed for  $WX = X$  and  $Wy = y$  for an in-place transform.

```
int gsl_multifit_linear_lcurve(const gsl_vector *y, gsl_vector *reg_param, gsl_vector *rho, gsl_vector *eta,
                               gsl_multifit_linear_workspace *work)
```

This function computes the L-curve for a least squares system using the right hand side vector  $y$  and the SVD decomposition of the least squares matrix  $X$ , which must be provided to `gsl_multifit_linear_svd()` prior to calling this function. The output vectors `reg_param`, `rho`, and `eta` must all be the same size, and will contain the regularization parameters  $\lambda_i$ , residual norms  $\|y - Xc_i\|$ , and solution norms  $\|Lc_i\|$  which compose the L-curve, where  $c_i$  is the regularized solution vector corresponding to  $\lambda_i$ . The user may determine the number of points on the L-curve by adjusting the size of these input arrays. The regularization parameters  $\lambda_i$  are estimated from the singular values of  $X$ , and chosen to represent the most relevant portion of the L-curve.

```
int gsl_multifit_linear_lcurvature(const gsl_vector *y, const gsl_vector *reg_param, const gsl_vector *rho,
                                   const gsl_vector *eta, gsl_vector *kappa, gsl_multifit_linear_workspace
                                   *work)
```

This function computes the curvature of the L-curve as a function of the regularization parameter  $\lambda$ , using the right hand side vector  $y$ , the vector of regularization parameters, `reg_param`, vector of residual norms, `rho`, and vector of solution norms, `eta`. The arrays `reg_param`, `rho`, and `eta` can be computed by `gsl_multifit_linear_lcurve()`. The curvature is defined as

$$\kappa(\lambda) = \frac{\hat{\rho}'\hat{\eta}'' - \hat{\rho}''\hat{\eta}'}{((\hat{\rho}')^2 + (\hat{\eta}')^2)^{\frac{3}{2}}}$$

where  $\hat{\rho}(\lambda) = \log \|y - Xc_\lambda\|$  and  $\hat{\eta}(\lambda) = \log \|Lc_\lambda\|$ . The curvature values are stored in `kappa` on output.

```
int gsl_multifit_linear_lcorner(const gsl_vector *rho, const gsl_vector *eta, size_t *idx)
```

This function attempts to locate the corner of the L-curve ( $\|y - Xc\|, \|Lc\|$ ) defined by the `rho` and `eta` input arrays respectively. The corner is defined as the point of maximum curvature of the L-curve in log-log scale. The `rho` and `eta` arrays can be outputs of `gsl_multifit_linear_lcurve()`. The algorithm used simply fits a circle to 3 consecutive points on the L-curve and uses the circle's radius to determine the curvature at the middle point. Therefore, the input array sizes must be  $\geq 3$ . With more points provided for the L-curve, a better estimate of the curvature can be obtained. The array index corresponding to maximum curvature (ie: the corner) is returned in `idx`. If the input arrays contain colinear points, this function could fail and return `GSL_EINVAL`.

```
int gsl_multifit_linear_lcorner2(const gsl_vector *reg_param, const gsl_vector *eta, size_t *idx)
```

This function attempts to locate the corner of an alternate L-curve ( $\lambda^2, \|Lc\|^2$ ) studied by Rezaghi and Hosseini, 2009. This alternate L-curve can provide better estimates of the regularization parameter for smooth solution vectors. The regularization parameters  $\lambda$  and solution norms  $\|Lc\|$  are provided in the `reg_param` and `eta` input arrays respectively. The corner is defined as the point of maximum curvature of this alternate L-curve in linear scale. The `reg_param` and `eta` arrays can be outputs of `gsl_multifit_linear_lcurve()`. The algorithm used simply fits a circle to 3 consecutive points on the L-curve and uses the circle's radius to determine the curvature at the middle point. Therefore, the input array sizes must be  $\geq 3$ . With more points provided for the L-curve, a better estimate of the curvature can be obtained. The array index corresponding to maximum curvature (ie: the corner) is returned in `idx`. If the input arrays contain colinear points, this function could fail and return `GSL_EINVAL`.

```
int gsl_multifit_linear_gcv_init(const gsl_vector *y, gsl_vector *reg_param, gsl_vector *UTy, double
                                 *delta0, gsl_multifit_linear_workspace *work)
```

This function performs some initialization in preparation for computing the GCV curve and its minimum. The right hand side vector is provided in  $y$ . On output, `reg_param` is set to a vector of regularization parameters

in decreasing order and may be of any size. The vector  $UTy$  of size  $p$  is set to  $U^T y$ . The parameter  $\delta$  is needed for subsequent steps of the GCV calculation.

```
int gsl_multifit_linear_gcv_curve(const gsl_vector *reg_param, const gsl_vector *UTy, const double delta0,
                                gsl_vector *G, gsl_multifit_linear_workspace *work)
```

This function calculates the GCV curve  $G(\lambda)$  and stores it in  $G$  on output, which must be the same size as  $reg\_param$ . The inputs  $reg\_param$ ,  $UTy$  and  $\delta$  are computed in `gsl_multifit_linear_gcv_init()`.

```
int gsl_multifit_linear_gcv_min(const gsl_vector *reg_param, const gsl_vector *UTy, const gsl_vector *G,
                                const double delta0, double *lambda, gsl_multifit_linear_workspace *work)
```

This function computes the value of the regularization parameter which minimizes the GCV curve  $G(\lambda)$  and stores it in  $lambda$ . The input  $G$  is calculated by `gsl_multifit_linear_gcv_curve()` and the inputs  $reg\_param$ ,  $UTy$  and  $\delta$  are computed by `gsl_multifit_linear_gcv_init()`.

```
double gsl_multifit_linear_gcv_calc(const double lambda, const gsl_vector *UTy, const double delta0,
                                    gsl_multifit_linear_workspace *work)
```

This function returns the value of the GCV curve  $G(\lambda)$  corresponding to the input  $lambda$ .

```
int gsl_multifit_linear_gcv(const gsl_vector *y, gsl_vector *reg_param, gsl_vector *G, double *lambda,
                             double *G_lambda, gsl_multifit_linear_workspace *work)
```

This function combines the steps `gcv_init`, `gcv_curve`, and `gcv_min` defined above into a single function. The input  $y$  is the right hand side vector. On output,  $reg\_param$  and  $G$ , which must be the same size, are set to vectors of  $\lambda$  and  $G(\lambda)$  values respectively. The output  $lambda$  is set to the optimal value of  $\lambda$  which minimizes the GCV curve. The minimum value of the GCV curve is returned in  $G\_lambda$ .

```
int gsl_multifit_linear_Lk(const size_t p, const size_t k, gsl_matrix *L)
```

This function computes the discrete approximation to the derivative operator  $L_k$  of order  $k$  on a regular grid of  $p$  points and stores it in  $L$ . The dimensions of  $L$  are  $(p - k)$ -by- $p$ .

```
int gsl_multifit_linear_Lsobolev(const size_t p, const size_t kmax, const gsl_vector *alpha, gsl_matrix *L,
                                 gsl_multifit_linear_workspace *work)
```

This function computes the regularization matrix  $L$  corresponding to the weighted Sobolev norm  $\|Lc\|^2 = \sum_k \alpha_k^2 \|L_k c\|^2$  where  $L_k$  approximates the derivative operator of order  $k$ . This regularization norm can be useful in applications where it is necessary to smooth several derivatives of the solution.  $p$  is the number of model parameters,  $kmax$  is the highest derivative to include in the summation above, and  $alpha$  is the vector of weights of size  $kmax + 1$ , where  $alpha[k] = \alpha_k$  is the weight assigned to the derivative of order  $k$ . The output matrix  $L$  is size  $p$ -by- $p$  and upper triangular.

```
double gsl_multifit_linear_rcond(const gsl_multifit_linear_workspace *work)
```

This function returns the reciprocal condition number of the least squares matrix  $X$ , defined as the ratio of the smallest and largest singular values,  $rcond = \sigma_{min}/\sigma_{max}$ . The routine `gsl_multifit_linear_svd()` must first be called to compute the SVD of  $X$ .

## 40.5 Robust linear regression

Ordinary least squares (OLS) models are often heavily influenced by the presence of outliers. Outliers are data points which do not follow the general trend of the other observations, although there is strictly no precise definition of an outlier. Robust linear regression refers to regression algorithms which are robust to outliers. The most common type of robust regression is M-estimation. The general M-estimator minimizes the objective function

$$\sum_i \rho(e_i) = \sum_i \rho(y_i - Y(c, x_i))$$

where  $e_i = y_i - Y(c, x_i)$  is the residual of the  $i$ th data point, and  $\rho(e_i)$  is a function which should have the following properties:

- $\rho(e) \geq 0$

- $\rho(0) = 0$
- $\rho(-e) = \rho(e)$
- $\rho(e_1) > \rho(e_2)$  for  $|e_1| > |e_2|$

The special case of ordinary least squares is given by  $\rho(e_i) = e_i^2$ . Letting  $\psi = \rho'$  be the derivative of  $\rho$ , differentiating the objective function with respect to the coefficients  $c$  and setting the partial derivatives to zero produces the system of equations

$$\sum_i \psi(e_i) X_i = 0$$

where  $X_i$  is a vector containing row  $i$  of the design matrix  $X$ . Next, we define a weight function  $w(e) = \psi(e)/e$ , and let  $w_i = w(e_i)$ :

$$\sum_i w_i e_i X_i = 0$$

This system of equations is equivalent to solving a weighted ordinary least squares problem, minimizing  $\chi^2 = \sum_i w_i e_i^2$ . The weights however, depend on the residuals  $e_i$ , which depend on the coefficients  $c$ , which depend on the weights. Therefore, an iterative solution is used, called Iteratively Reweighted Least Squares (IRLS).

1. Compute initial estimates of the coefficients  $c^{(0)}$  using ordinary least squares
2. For iteration  $k$ , form the residuals  $e_i^{(k)} = (y_i - X_i c^{(k-1)}) / (t \sigma^{(k)} \sqrt{1 - h_i})$ , where  $t$  is a tuning constant depending on the choice of  $\psi$ , and  $h_i$  are the statistical leverages (diagonal elements of the matrix  $X(X^T X)^{-1} X^T$ ). Including  $t$  and  $h_i$  in the residual calculation has been shown to improve the convergence of the method. The residual standard deviation is approximated as  $\sigma^{(k)} = MAD / 0.6745$ , where MAD is the Median-Absolute-Deviation of the  $n - p$  largest residuals from the previous iteration.
3. Compute new weights  $w_i^{(k)} = \psi(e_i^{(k)}) / e_i^{(k)}$ .
4. Compute new coefficients  $c^{(k)}$  by solving the weighted least squares problem with weights  $w_i^{(k)}$ .
5. Steps 2 through 4 are iterated until the coefficients converge or until some maximum iteration limit is reached. Coefficients are tested for convergence using the criteria:

$$|c_i^{(k)} - c_i^{(k-1)}| \leq \epsilon \times \max(|c_i^{(k)}|, |c_i^{(k-1)}|)$$

for all  $0 \leq i < p$  where  $\epsilon$  is a small tolerance factor.

The key to this method lies in selecting the function  $\psi(e_i)$  to assign smaller weights to large residuals, and larger weights to smaller residuals. As the iteration proceeds, outliers are assigned smaller and smaller weights, eventually having very little or no effect on the fitted model.

#### type `gsl_multifit_robust_workspace`

This workspace is used for robust least squares fitting.

`gsl_multifit_robust_workspace *gsl_multifit_robust_alloc`(const `gsl_multifit_robust_type` \*T, const size\_t n, const size\_t p)

This function allocates a workspace for fitting a model to  $n$  observations using  $p$  parameters. The size of the workspace is  $O(np + p^2)$ . The type  $T$  specifies the function  $\psi$  and can be selected from the following choices.

#### type `gsl_multifit_robust_type`

`gsl_multifit_robust_type *gsl_multifit_robust_default`

This specifies the `gsl_multifit_robust_bisquare` type (see below) and is a good general purpose choice for robust regression.



*gsl\_multifit\_robust\_type* \***gsl\_multifit\_robust\_bisquare**

This is Tukey's biweight (bisquare) function and is a good general purpose choice for robust regression. The weight function is given by

$$w(e) = \begin{cases} (1 - e^2)^2, & |e| \leq 1 \\ 0, & |e| > 1 \end{cases}$$

and the default tuning constant is  $t = 4.685$ .

*gsl\_multifit\_robust\_type* \***gsl\_multifit\_robust\_cauchy**

This is Cauchy's function, also known as the Lorentzian function. This function does not guarantee a unique solution, meaning different choices of the coefficient vector  $\mathbf{c}$  could minimize the objective function. Therefore this option should be used with care. The weight function is given by

$$w(e) = \frac{1}{1 + e^2}$$

and the default tuning constant is  $t = 2.385$ .

*gsl\_multifit\_robust\_type* \***gsl\_multifit\_robust\_fair**

This is the fair  $\rho$  function, which guarantees a unique solution and has continuous derivatives to three orders. The weight function is given by

$$w(e) = \frac{1}{1 + |e|}$$

and the default tuning constant is  $t = 1.400$ .

*gsl\_multifit\_robust\_type* \***gsl\_multifit\_robust\_huber**

This specifies Huber's  $\rho$  function, which is a parabola in the vicinity of zero and increases linearly for a given threshold  $|e| > t$ . This function is also considered an excellent general purpose robust estimator, however, occasional difficulties can be encountered due to the discontinuous first derivative of the  $\psi$  function. The weight function is given by

$$w(e) = \begin{cases} 1, & |e| \leq 1 \\ \frac{1}{|e|}, & |e| > 1 \end{cases}$$

and the default tuning constant is  $t = 1.345$ .

*gsl\_multifit\_robust\_type* \***gsl\_multifit\_robust\_ols**

This specifies the ordinary least squares solution, which can be useful for quickly checking the difference between the various robust and OLS solutions. The weight function is given by

$$w(e) = 1$$

and the default tuning constant is  $t = 1$ .

*gsl\_multifit\_robust\_type* \***gsl\_multifit\_robust\_welsch**

This specifies the Welsch function which can perform well in cases where the residuals have an exponential distribution. The weight function is given by

$$w(e) = \exp(-e^2)$$

and the default tuning constant is  $t = 2.985$ .

void **gsl\_multifit\_robust\_free**(*gsl\_multifit\_robust\_workspace* \*w)

This function frees the memory associated with the workspace  $w$ .

const char \***gsl\_multifit\_robust\_name**(const *gsl\_multifit\_robust\_workspace* \*w)

This function returns the name of the robust type  $T$  specified to *gsl\_multifit\_robust\_alloc()*.

int **gsl\_multifit\_robust\_tune**(const double *tune*, *gsl\_multifit\_robust\_workspace* \*w)

This function sets the tuning constant  $t$  used to adjust the residuals at each iteration to *tune*. Decreasing the tuning constant increases the downweight assigned to large residuals, while increasing the tuning constant decreases the downweight assigned to large residuals.

int **gsl\_multifit\_robust\_maxiter**(const size\_t *maxiter*, *gsl\_multifit\_robust\_workspace* \*w)

This function sets the maximum number of iterations in the iteratively reweighted least squares algorithm to *maxiter*. By default, this value is set to 100 by *gsl\_multifit\_robust\_alloc()*.

int **gsl\_multifit\_robust\_weights**(const *gsl\_vector* \*r, *gsl\_vector* \*wts, *gsl\_multifit\_robust\_workspace* \*w)

This function assigns weights to the vector *wts* using the residual vector *r* and previously specified weighting function. The output weights are given by  $wts_i = w(r_i/(t\sigma))$ , where the weighting functions  $w$  are detailed in *gsl\_multifit\_robust\_alloc()*.  $\sigma$  is an estimate of the residual standard deviation based on the Median-Absolute-Deviation and  $t$  is the tuning constant. This function is useful if the user wishes to implement their own robust regression rather than using the supplied *gsl\_multifit\_robust()* routine below.

int **gsl\_multifit\_robust**(const *gsl\_matrix* \*X, const *gsl\_vector* \*y, *gsl\_vector* \*c, *gsl\_matrix* \*cov, *gsl\_multifit\_robust\_workspace* \*w)

This function computes the best-fit parameters  $c$  of the model  $y = Xc$  for the observations  $y$  and the matrix of predictor variables  $X$ , attempting to reduce the influence of outliers using the algorithm outlined above. The  $p$ -by- $p$  variance-covariance matrix of the model parameters *cov* is estimated as  $\sigma^2(X^T X)^{-1}$ , where  $\sigma$  is an approximation of the residual standard deviation using the theory of robust regression. Special care must be taken when estimating  $\sigma$  and other statistics such as  $R^2$ , and so these are computed internally and are available by calling the function *gsl\_multifit\_robust\_statistics()*.

If the coefficients do not converge within the maximum iteration limit, the function returns GSL\_EMAXITER. In this case, the current estimates of the coefficients and covariance matrix are returned in *c* and *cov* and the internal fit statistics are computed with these estimates.

int **gsl\_multifit\_robust\_est**(const *gsl\_vector* \*x, const *gsl\_vector* \*c, const *gsl\_matrix* \*cov, double \*y, double \*y\_err)

This function uses the best-fit robust regression coefficients *c* and their covariance matrix *cov* to compute the fitted function value *y* and its standard deviation *y\_err* for the model  $y = x \cdot c$  at the point  $x$ .

int **gsl\_multifit\_robust\_residuals**(const *gsl\_matrix* \*X, const *gsl\_vector* \*y, const *gsl\_vector* \*c, *gsl\_vector* \*r, *gsl\_multifit\_robust\_workspace* \*w)

This function computes the vector of studentized residuals  $r_i = \frac{y_i - (Xc)_i}{\sigma\sqrt{1-h_i}}$  for the observations  $y$ , coefficients  $c$  and matrix of predictor variables  $X$ . The routine *gsl\_multifit\_robust()* must first be called to compute the statistical leverages  $h_i$  of the matrix  $X$  and residual standard deviation estimate  $\sigma$ .

*gsl\_multifit\_robust\_stats* **gsl\_multifit\_robust\_statistics**(const *gsl\_multifit\_robust\_workspace* \*w)

This function returns a structure containing relevant statistics from a robust regression. The function *gsl\_multifit\_robust()* must be called first to perform the regression and calculate these statistics. The returned *gsl\_multifit\_robust\_stats* structure contains the following fields.

type **gsl\_multifit\_robust\_stats**  
double *sigma\_ols*

This contains the standard deviation of the residuals as computed from ordinary least squares (OLS).

double *sigma\_mad*

This contains an estimate of the standard deviation of the final residuals using the Median-Absolute-Deviation statistic

double *sigma\_rob*

This contains an estimate of the standard deviation of the final residuals from the theory of robust regression (see Street et al, 1988).

double sigma

This contains an estimate of the standard deviation of the final residuals by attempting to reconcile `sigma_rob` and `sigma_ols` in a reasonable way.

double Rsq

This contains the  $R^2$  coefficient of determination statistic using the estimate `sigma`.

double adj\_Rsq

This contains the adjusted  $R^2$  coefficient of determination statistic using the estimate `sigma`.

double rmse

This contains the root mean squared error of the final residuals

double sse

This contains the residual sum of squares taking into account the robust covariance matrix.

size\_t dof

This contains the number of degrees of freedom  $n - p$

size\_t numit

Upon successful convergence, this contains the number of iterations performed

gsl\_vector \* weights

This contains the final weight vector of length `n`

gsl\_vector \* r

This contains the final residual vector of length `n`,  $r = y - Xc$

## 40.6 Large dense linear systems

This module is concerned with solving large dense least squares systems  $Xc = y$  where the  $n$ -by- $p$  matrix  $X$  has  $n \gg p$  (ie: many more rows than columns). This type of matrix is called a “tall skinny” matrix, and for some applications, it may not be possible to fit the entire matrix in memory at once to use the standard SVD approach. Therefore, the algorithms in this module are designed to allow the user to construct smaller blocks of the matrix  $X$  and accumulate those blocks into the larger system one at a time. The algorithms in this module never need to store the entire matrix  $X$  in memory. The large linear least squares routines support data weights and Tikhonov regularization, and are designed to minimize the residual

$$\chi^2 = \|y - Xc\|_W^2 + \lambda^2 \|Lc\|^2$$

where  $y$  is the  $n$ -by-1 observation vector,  $X$  is the  $n$ -by- $p$  design matrix,  $c$  is the  $p$ -by-1 solution vector,  $W = \text{diag}(w_1, \dots, w_n)$  is the data weighting matrix,  $L$  is an  $m$ -by- $p$  regularization matrix,  $\lambda$  is a regularization parameter, and  $\|r\|_W^2 = r^T W r$ . In the discussion which follows, we will assume that the system has been converted into Tikhonov standard form,

$$\chi^2 = \|\tilde{y} - \tilde{X}\tilde{c}\|^2 + \lambda^2 \|\tilde{c}\|^2$$

and we will drop the tilde characters from the various parameters. For a discussion of the transformation to standard form, see [Regularized regression](#).

The basic idea is to partition the matrix  $X$  and observation vector  $y$  as

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_k \end{pmatrix} c = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_k \end{pmatrix}$$

into  $k$  blocks, where each block  $(X_i, y_i)$  may have any number of rows, but each  $X_i$  has  $p$  columns. The sections below describe the methods available for solving this partitioned system. The functions are declared in the header file `gsl_multilarge.h`.

### 40.6.1 Normal Equations Approach

The normal equations approach to the large linear least squares problem described above is popular due to its speed and simplicity. Since the normal equations solution to the problem is given by

$$c = (X^T X + \lambda^2 I)^{-1} X^T y$$

only the  $p$ -by- $p$  matrix  $X^T X$  and  $p$ -by-1 vector  $X^T y$  need to be stored. Using the partition scheme described above, these are given by

$$\begin{aligned} X^T X &= \sum_i X_i^T X_i \\ X^T y &= \sum_i X_i^T y_i \end{aligned}$$

Since the matrix  $X^T X$  is symmetric, only half of it needs to be calculated. Once all of the blocks  $(X_i, y_i)$  have been accumulated into the final  $X^T X$  and  $X^T y$ , the system can be solved with a Cholesky factorization of the  $X^T X$  matrix. The  $X^T X$  matrix is first transformed via a diagonal scaling transformation to attempt to reduce its condition number as much as possible to recover a more accurate solution vector. The normal equations approach is the fastest method for solving the large least squares problem, and is accurate for well-conditioned matrices  $X$ . However, for ill-conditioned matrices, as is often the case for large systems, this method can suffer from numerical instabilities (see Trefethen and Bau, 1997). The number of operations for this method is  $O(np^2 + \frac{1}{3}p^3)$ .

### 40.6.2 Tall Skinny QR (TSQR) Approach

An algorithm which has better numerical stability for ill-conditioned problems is known as the Tall Skinny QR (TSQR) method. This method is based on computing the thin QR decomposition of the least squares matrix  $X = QR$ , where  $Q$  is an  $n$ -by- $p$  matrix with orthogonal columns, and  $R$  is a  $p$ -by- $p$  upper triangular matrix. Once these factors are calculated, the residual becomes

$$\chi^2 = \|Q^T y - Rc\|^2 + \lambda^2 \|c\|^2$$

which can be written as the matrix equation

$$\begin{pmatrix} R \\ \lambda I \end{pmatrix} c = \begin{pmatrix} Q^T y \\ 0 \end{pmatrix}$$

The matrix on the left hand side is now a much smaller  $2p$ -by- $p$  matrix which can be solved with a standard SVD approach. The  $Q$  matrix is just as large as the original matrix  $X$ , however it does not need to be explicitly constructed. The TSQR algorithm computes only the  $p$ -by- $p$  matrix  $R$  and the  $p$ -by-1 vector  $Q^T y$ , and updates these quantities as

new blocks are added to the system. Each time a new block of rows  $(X_i, y_i)$  is added, the algorithm performs a QR decomposition of the matrix

$$\begin{pmatrix} R_{i-1} \\ X_i \end{pmatrix}$$

where  $R_{i-1}$  is the upper triangular  $R$  factor for the matrix

$$\begin{pmatrix} X_1 \\ \vdots \\ X_{i-1} \end{pmatrix}$$

This QR decomposition is done efficiently taking into account the sparse structure of  $R_{i-1}$ . See Demmel et al, 2008 for more details on how this is accomplished. The number of operations for this method is  $O(2np^2 - \frac{2}{3}p^3)$ .

### 40.6.3 Large Dense Linear Systems Solution Steps

The typical steps required to solve large regularized linear least squares problems are as follows:

1. Choose the regularization matrix  $L$ .
2. Construct a block of rows of the least squares matrix, right hand side vector, and weight vector  $(X_i, y_i, w_i)$ .
3. Transform the block to standard form  $(\tilde{X}_i, \tilde{y}_i)$ . This step can be skipped if  $L = I$  and  $W = I$ .
4. Accumulate the standard form block  $(\tilde{X}_i, \tilde{y}_i)$  into the system.
5. Repeat steps 2-4 until the entire matrix and right hand side vector have been accumulated.
6. Determine an appropriate regularization parameter  $\lambda$  (using for example L-curve analysis).
7. Solve the standard form system using the chosen  $\lambda$ .
8. Backtransform the standard form solution  $\tilde{c}$  to recover the original solution vector  $c$ .

### 40.6.4 Large Dense Linear Least Squares Routines

type **gsl\_multilarge\_linear\_workspace**

This workspace contains parameters for solving large linear least squares problems.

*gsl\_multilarge\_linear\_workspace* \***gsl\_multilarge\_linear\_alloc**(const *gsl\_multilarge\_linear\_type* \*T, const size\_t p)

This function allocates a workspace for solving large linear least squares systems. The least squares matrix  $X$  has  $p$  columns, but may have any number of rows.

type **gsl\_multilarge\_linear\_type**

The parameter  $T$  specifies the method to be used for solving the large least squares system and may be selected from the following choices

*gsl\_multilarge\_linear\_type* \***gsl\_multilarge\_linear\_normal**

This specifies the normal equations approach for solving the least squares system. This method is suitable in cases where performance is critical and it is known that the least squares matrix  $X$  is well conditioned. The size of this workspace is  $O(p^2)$ .

*gsl\_multilarge\_linear\_type* \***gsl\_multilarge\_linear\_tsqr**

This specifies the sequential Tall Skinny QR (TSQR) approach for solving the least squares system. This method is a good general purpose choice for large systems, but requires about twice as many operations as the normal equations method for  $n \gg p$ . The size of this workspace is  $O(p^2)$ .

void **gsl\_multilarge\_linear\_free**(*gsl\_multilarge\_linear\_workspace* \*w)

This function frees the memory associated with the workspace *w*.

const char \***gsl\_multilarge\_linear\_name**(*gsl\_multilarge\_linear\_workspace* \*w)

This function returns a string pointer to the name of the multilarge solver.

int **gsl\_multilarge\_linear\_reset**(*gsl\_multilarge\_linear\_workspace* \*w)

This function resets the workspace *w* so it can begin to accumulate a new least squares system.

int **gsl\_multilarge\_linear\_stdform1**(const *gsl\_vector* \*L, const *gsl\_matrix* \*X, const *gsl\_vector* \*y, *gsl\_matrix* \*Xs, *gsl\_vector* \*ys, *gsl\_multilarge\_linear\_workspace* \*work)

int **gsl\_multilarge\_linear\_wstdform1**(const *gsl\_vector* \*L, const *gsl\_matrix* \*X, const *gsl\_vector* \*w, const *gsl\_vector* \*y, *gsl\_matrix* \*Xs, *gsl\_vector* \*ys, *gsl\_multilarge\_linear\_workspace* \*work)

These functions define a regularization matrix  $L = \text{diag}(l_0, l_1, \dots, l_{p-1})$ . The diagonal matrix element  $l_i$  is provided by the  $i$ -th element of the input vector  $L$ . The block  $(X, y)$  is converted to standard form and the parameters  $(\tilde{X}, \tilde{y})$  are stored in  $Xs$  and  $ys$  on output.  $Xs$  and  $ys$  have the same dimensions as  $X$  and  $y$ . Optional data weights may be supplied in the vector  $w$ . In order to apply this transformation,  $L^{-1}$  must exist and so none of the  $l_i$  may be zero. After the standard form system has been solved, use [gsl\\_multilarge\\_linear\\_genform1\(\)](#) to recover the original solution vector. It is allowed to have  $X = Xs$  and  $y = ys$  for an in-place transform.

int **gsl\_multilarge\_linear\_L\_decomp**(*gsl\_matrix* \*L, *gsl\_vector* \*tau)

This function calculates the QR decomposition of the  $m$ -by- $p$  regularization matrix  $L$ .  $L$  must have  $m \geq p$ . On output, the Householder scalars are stored in the vector  $\tau$  of size  $p$ . These outputs will be used by [gsl\\_multilarge\\_linear\\_wstdform2\(\)](#) to complete the transformation to standard form.

int **gsl\_multilarge\_linear\_stdform2**(const *gsl\_matrix* \*LQR, const *gsl\_vector* \*Ltau, const *gsl\_matrix* \*X, const *gsl\_vector* \*y, *gsl\_matrix* \*Xs, *gsl\_vector* \*ys, *gsl\_multilarge\_linear\_workspace* \*work)

int **gsl\_multilarge\_linear\_wstdform2**(const *gsl\_matrix* \*LQR, const *gsl\_vector* \*Ltau, const *gsl\_matrix* \*X, const *gsl\_vector* \*w, const *gsl\_vector* \*y, *gsl\_matrix* \*Xs, *gsl\_vector* \*ys, *gsl\_multilarge\_linear\_workspace* \*work)

These functions convert a block of rows  $(X, y, w)$  to standard form  $(\tilde{X}, \tilde{y})$  which are stored in  $Xs$  and  $ys$  respectively.  $X$ ,  $y$ , and  $w$  must all have the same number of rows. The  $m$ -by- $p$  regularization matrix  $L$  is specified by the inputs  $LQR$  and  $Ltau$ , which are outputs from [gsl\\_multilarge\\_linear\\_L\\_decomp\(\)](#).  $Xs$  and  $ys$  have the same dimensions as  $X$  and  $y$ . After the standard form system has been solved, use [gsl\\_multilarge\\_linear\\_genform2\(\)](#) to recover the original solution vector. Optional data weights may be supplied in the vector  $w$ , where  $W = \text{diag}(w)$ .

int **gsl\_multilarge\_linear\_accumulate**(*gsl\_matrix* \*X, *gsl\_vector* \*y, *gsl\_multilarge\_linear\_workspace* \*w)

This function accumulates the standard form block  $(X, y)$  into the current least squares system.  $X$  and  $y$  have the same number of rows, which can be arbitrary.  $X$  must have  $p$  columns. For the TSQR method,  $X$  and  $y$  are destroyed on output. For the normal equations method, they are both unchanged.

int **gsl\_multilarge\_linear\_solve**(const double lambda, *gsl\_vector* \*c, double \*rnorm, double \*snorm, *gsl\_multilarge\_linear\_workspace* \*w)

After all blocks  $(X_i, y_i)$  have been accumulated into the large least squares system, this function will compute the solution vector which is stored in  $c$  on output. The regularization parameter  $\lambda$  is provided in  $lambda$ . On output,  $rnorm$  contains the residual norm  $\|y - Xc\|_W$  and  $snorm$  contains the solution norm  $\|Lc\|$ .

int **gsl\_multilarge\_linear\_genform1**(const *gsl\_vector* \*L, const *gsl\_vector* \*cs, *gsl\_vector* \*c, *gsl\_multilarge\_linear\_workspace* \*work)

After a regularized system has been solved with  $L = \text{diag}(l_0, l_1, \dots, l_{p-1})$ , this function backtransforms the standard form solution vector  $cs$  to recover the solution vector of the original problem  $c$ . The diagonal matrix elements  $l_i$  are provided in the vector  $L$ . It is allowed to have  $c = cs$  for an in-place transform.

int **gsl\_multilarge\_linear\_genform2**(const *gsl\_matrix* \*LQR, const *gsl\_vector* \*Ltau, const *gsl\_vector* \*cs, *gsl\_vector* \*c, *gsl\_multilarge\_linear\_workspace* \*work)

After a regularized system has been solved with a regularization matrix  $L$ , specified by ( $LQR$ ,  $L\tau$ ), this function backtransforms the standard form solution  $cs$  to recover the solution vector of the original problem, which is stored in  $c$ , of length  $p$ .

int **gsl\_multilarge\_linear\_lcurve**(*gsl\_vector* \*reg\_param, *gsl\_vector* \*rho, *gsl\_vector* \*eta, *gsl\_multilarge\_linear\_workspace* \*work)

This function computes the L-curve for a large least squares system after it has been fully accumulated into the workspace *work*. The output vectors *reg\_param*, *rho*, and *eta* must all be the same size, and will contain the regularization parameters  $\lambda_i$ , residual norms  $\|y - Xc_i\|$ , and solution norms  $\|Lc_i\|$  which compose the L-curve, where  $c_i$  is the regularized solution vector corresponding to  $\lambda_i$ . The user may determine the number of points on the L-curve by adjusting the size of these input arrays. For the TSQR method, the regularization parameters  $\lambda_i$  are estimated from the singular values of the triangular  $R$  factor. For the normal equations method, they are estimated from the eigenvalues of the  $X^T X$  matrix.

const *gsl\_matrix* \***gsl\_multilarge\_linear\_matrix\_ptr**(const *gsl\_multilarge\_linear\_workspace* \*work)

For the normal equations method, this function returns a pointer to the  $X^T X$  matrix. For the TSQR method, this function returns a pointer to the upper triangular  $R$  matrix.

const *gsl\_vector* \***gsl\_multilarge\_linear\_rhs\_ptr**(const *gsl\_multilarge\_linear\_workspace* \*work)

For the normal equations method, this function returns a pointer to the  $X^T y$  right hand side vector. For the TSQR method, this function returns a pointer to the  $Q^T y$  right hand side vector.

int **gsl\_multilarge\_linear\_rcond**(double \*rcond, *gsl\_multilarge\_linear\_workspace* \*work)

This function computes the reciprocal condition number, stored in *rcond*, of the least squares matrix after it has been accumulated into the workspace *work*. For the TSQR algorithm, this is accomplished by calculating the SVD of the  $R$  factor, which has the same singular values as the matrix  $X$ . For the normal equations method, this is done by computing the eigenvalues of  $X^T X$ , which could be inaccurate for ill-conditioned matrices  $X$ .

## 40.7 Troubleshooting

When using models based on polynomials, care should be taken when constructing the design matrix  $X$ . If the  $x$  values are large, then the matrix  $X$  could be ill-conditioned since its columns are powers of  $x$ , leading to unstable least-squares solutions. In this case it can often help to center and scale the  $x$  values using the mean and standard deviation:

$$x' = \frac{x - \mu(x)}{\sigma(x)}$$

and then construct the  $X$  matrix using the transformed values  $x'$ .

## 40.8 Examples

The example programs in this section demonstrate the various linear regression methods.



## 40.8.1 Simple Linear Regression Example

The following program computes a least squares straight-line fit to a simple dataset, and outputs the best-fit line and its associated one standard-deviation error bars.

```
#include <stdio.h>
#include <gsl/gsl_fit.h>

int
main (void)
{
    int i, n = 4;
    double x[4] = { 1970, 1980, 1990, 2000 };
    double y[4] = { 12, 11, 14, 13 };
    double w[4] = { 0.1, 0.2, 0.3, 0.4 };

    double c0, c1, cov00, cov01, cov11, chisq;

    gsl_fit_wlinear (x, 1, w, 1, y, 1, n,
                    &c0, &c1, &cov00, &cov01, &cov11,
                    &chisq);

    printf ("# best fit: Y = %g + %g X\n", c0, c1);
    printf ("# covariance matrix:\n");
    printf ("# [ %g, %g\n#   %g, %g]\n",
            cov00, cov01, cov01, cov11);
    printf ("# chisq = %g\n", chisq);

    for (i = 0; i < n; i++)
        printf ("data: %g %g %g\n",
                x[i], y[i], 1/sqrt(w[i]));

    printf ("\n");

    for (i = -30; i < 130; i++)
    {
        double xf = x[0] + (i/100.0) * (x[n-1] - x[0]);
        double yf, yf_err;

        gsl_fit_linear_est (xf,
                            c0, c1,
                            cov00, cov01, cov11,
                            &yf, &yf_err);

        printf ("fit: %g %g\n", xf, yf);
        printf ("hi : %g %g\n", xf, yf + yf_err);
        printf ("lo : %g %g\n", xf, yf - yf_err);
    }
    return 0;
}
```

The following commands extract the data from the output of the program and display it using the GNU plotutils “graph” utility:



```

$ ./demo > tmp
$ more tmp
# best fit: Y = -106.6 + 0.06 X
# covariance matrix:
# [ 39602, -19.9
#   -19.9, 0.01]
# chisq = 0.8

$ for n in data fit hi lo ;
do
  grep "^$n" tmp | cut -d: -f2 > $n ;
done
$ graph -T X -X x -Y y -y 0 20 -m 0 -S 2 -Ie data
-S 0 -I a -m 1 fit -m 2 hi -m 2 lo

```

The result is shown in [Fig. 40.1](#).

## 40.8.2 Multi-parameter Linear Regression Example

The following program performs a quadratic fit  $y = c_0 + c_1x + c_2x^2$  to a weighted dataset using the generalised linear fitting function `gsl_multifit_wlinear()`. The model matrix  $X$  for a quadratic fit is given by,

$$X = \begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \dots & \dots & \dots \end{pmatrix}$$

where the column of ones corresponds to the constant term  $c_0$ . The two remaining columns corresponds to the terms  $c_1x$  and  $c_2x^2$ .

The program reads  $n$  lines of data in the format  $(x, y, \text{err})$  where  $\text{err}$  is the error (standard deviation) in the value  $y$ .

```

#include <stdio.h>
#include <gsl/gsl_multifit.h>

int
main (int argc, char **argv)
{
  int i, n;
  double xi, yi, ei, chisq;
  gsl_matrix *X, *cov;
  gsl_vector *y, *w, *c;

  if (argc != 2)
    {
      fprintf (stderr, "usage: fit n < data\n");
      exit (-1);
    }

  n = atoi (argv[1]);

  X = gsl_matrix_alloc (n, 3);
  y = gsl_vector_alloc (n);

```

(continues on next page)

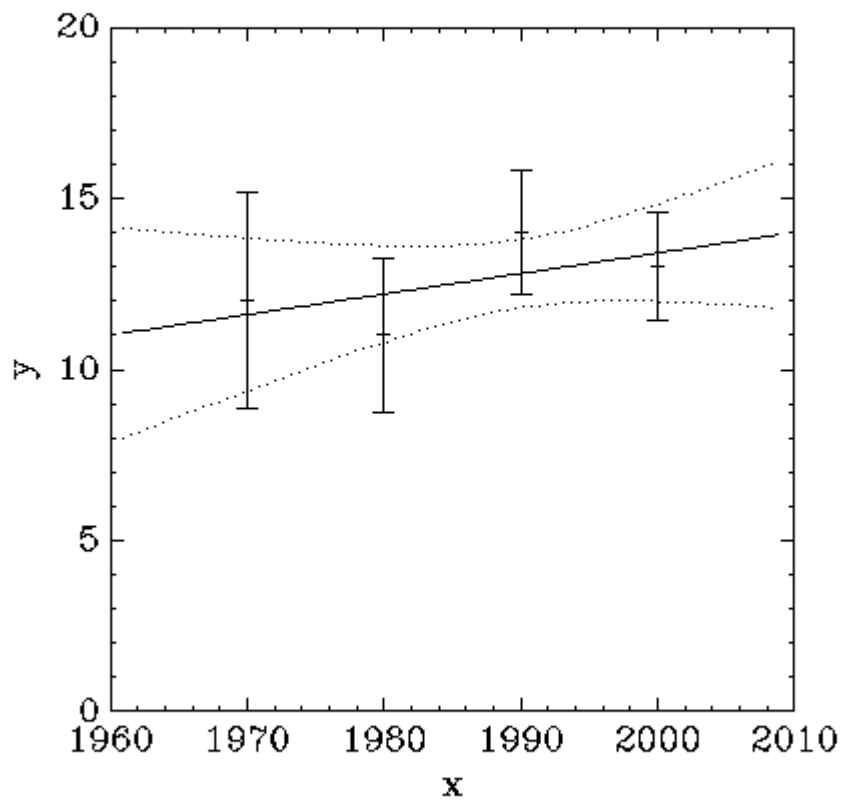


Fig. 40.1: Straight line fit with  $1\text{-}\sigma$  error bars

(continued from previous page)

```

w = gsl_vector_alloc (n);

c = gsl_vector_alloc (3);
cov = gsl_matrix_alloc (3, 3);

for (i = 0; i < n; i++)
{
    int count = fscanf (stdin, "%lg %lg %lg",
                        &xi, &yi, &ei);

    if (count != 3)
    {
        fprintf (stderr, "error reading file\n");
        exit (-1);
    }

    printf ("%g %g +/- %g\n", xi, yi, ei);

    gsl_matrix_set (X, i, 0, 1.0);
    gsl_matrix_set (X, i, 1, xi);
    gsl_matrix_set (X, i, 2, xi*xi);

    gsl_vector_set (y, i, yi);
    gsl_vector_set (w, i, 1.0/(ei*ei));
}

{
    gsl_multifit_linear_workspace * work
        = gsl_multifit_linear_alloc (n, 3);
    gsl_multifit_wlinear (X, w, y, c, cov,
                        &chisq, work);
    gsl_multifit_linear_free (work);
}

#define C(i) (gsl_vector_get(c,(i)))
#define COV(i,j) (gsl_matrix_get(cov,(i),(j)))

{
    printf ("# best fit: Y = %g + %g X + %g X^2\n",
           C(0), C(1), C(2));

    printf ("# covariance matrix:\n");
    printf ("[ %.5e, %.5e, %.5e \n",
           COV(0,0), COV(0,1), COV(0,2));
    printf ("  %.5e, %.5e, %.5e \n",
           COV(1,0), COV(1,1), COV(1,2));
    printf ("  %.5e, %.5e, %.5e ]\n",
           COV(2,0), COV(2,1), COV(2,2));
    printf ("# chisq = %g\n", chisq);
}

gsl_matrix_free (X);

```

(continues on next page)

(continued from previous page)

```

gsl_vector_free (y);
gsl_vector_free (w);
gsl_vector_free (c);
gsl_matrix_free (cov);

return 0;
}

```

A suitable set of data for fitting can be generated using the following program. It outputs a set of points with gaussian errors from the curve  $y = e^x$  in the region  $0 < x < 2$ .

```

#include <stdio.h>
#include <math.h>
#include <gsl/gsl_randist.h>

int
main (void)
{
    double x;
    const gsl_rng_type * T;
    gsl_rng * r;

    gsl_rng_env_setup ();

    T = gsl_rng_default;
    r = gsl_rng_alloc (T);

    for (x = 0.1; x < 2; x+= 0.1)
    {
        double y0 = exp (x);
        double sigma = 0.1 * y0;
        double dy = gsl_ran_gaussian (r, sigma);

        printf ("%g %g %g\n", x, y0 + dy, sigma);
    }

    gsl_rng_free(r);

    return 0;
}

```

The data can be prepared by running the resulting executable program:

```

$ GSL_RNG_TYPE=mt19937_1999 ./generate > exp.dat
$ more exp.dat
0.1 0.97935 0.110517
0.2 1.3359 0.12214
0.3 1.52573 0.134986
0.4 1.60318 0.149182
0.5 1.81731 0.164872
0.6 1.92475 0.182212
....

```

To fit the data use the previous program, with the number of data points given as the first argument. In this case there are 19 data points:

```
$ ./fit 19 < exp.dat
0.1 0.97935 +/- 0.110517
0.2 1.3359 +/- 0.12214
...
# best fit: Y = 1.02318 + 0.956201 X + 0.876796 X^2
# covariance matrix:
[ +1.25612e-02, -3.64387e-02, +1.94389e-02
  -3.64387e-02, +1.42339e-01, -8.48761e-02
  +1.94389e-02, -8.48761e-02, +5.60243e-02 ]
# chisq = 23.0987
```

The parameters of the quadratic fit match the coefficients of the expansion of  $e^x$ , taking into account the errors on the parameters and the  $O(x^3)$  difference between the exponential and quadratic functions for the larger values of  $x$ . The errors on the parameters are given by the square-root of the corresponding diagonal elements of the covariance matrix. The chi-squared per degree of freedom is 1.4, indicating a reasonable fit to the data.

Fig. 40.2 shows the resulting fit.

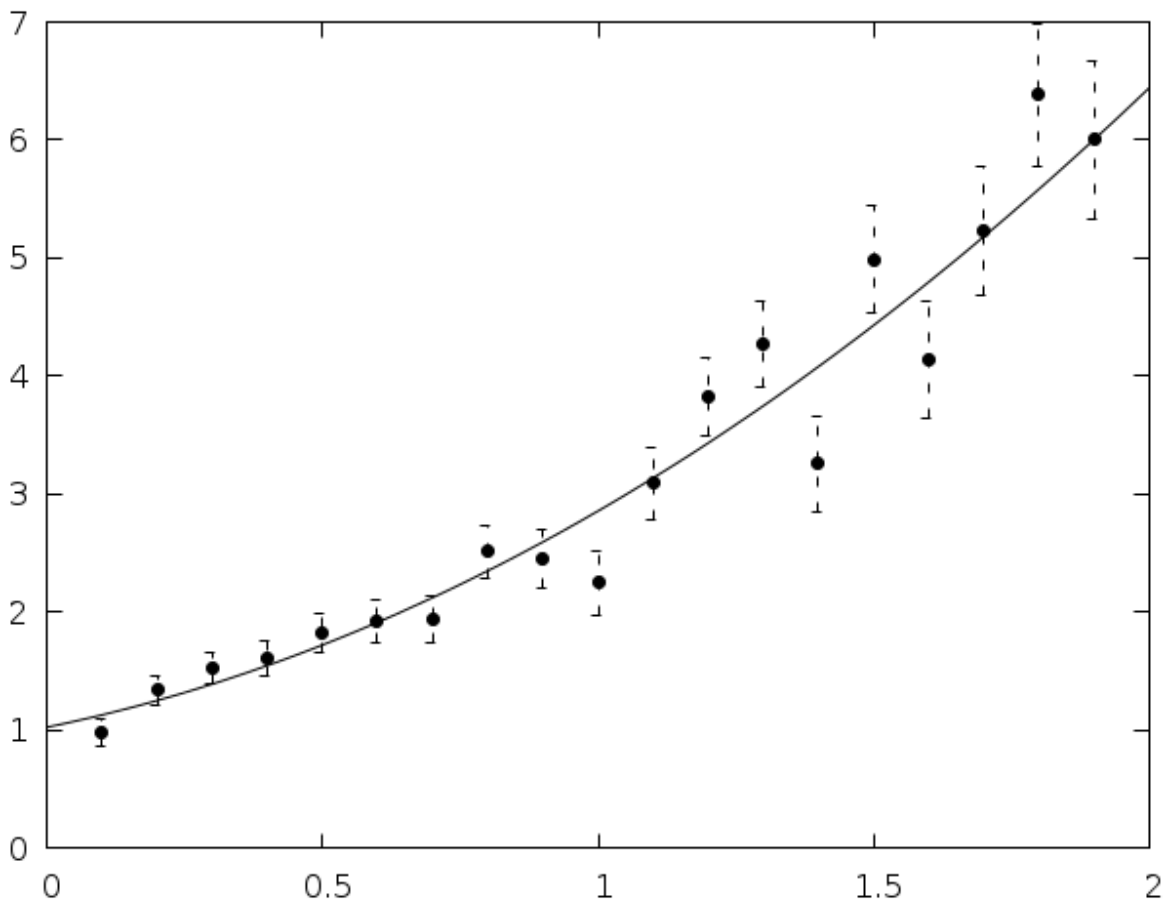


Fig. 40.2: Weighted fit example with error bars

### 40.8.3 Regularized Linear Regression Example 1

The next program demonstrates the difference between ordinary and regularized least squares when the design matrix is near-singular. In this program, we generate two random normally distributed variables  $u$  and  $v$ , with  $v = u + \text{noise}$  so that  $u$  and  $v$  are nearly colinear. We then set a third dependent variable  $y = u + v + \text{noise}$  and solve for the coefficients  $c_1, c_2$  of the model  $Y(c_1, c_2) = c_1u + c_2v$ . Since  $u \approx v$ , the design matrix  $X$  is nearly singular, leading to unstable ordinary least squares solutions.

Here is the program output:

```
matrix condition number = 1.025113e+04

=== Unregularized fit ===
best fit: y = -43.6588 u + 45.6636 v
residual norm = 31.6248
solution norm = 63.1764
chisq/dof = 1.00213

=== Regularized fit (L-curve) ===
optimal lambda: 4.51103
best fit: y = 1.00113 u + 1.0032 v
residual norm = 31.6547
solution norm = 1.41728
chisq/dof = 1.04499

=== Regularized fit (GCV) ===
optimal lambda: 0.0232029
best fit: y = -19.8367 u + 21.8417 v
residual norm = 31.6332
solution norm = 29.5051
chisq/dof = 1.00314
```

We see that the ordinary least squares solution is completely wrong, while the L-curve regularized method with the optimal  $\lambda = 4.51103$  finds the correct solution  $c_1 \approx c_2 \approx 1$ . The GCV regularized method finds a regularization parameter  $\lambda = 0.0232029$  which is too small to give an accurate solution, although it performs better than OLS. The L-curve and its computed corner, as well as the GCV curve and its minimum are plotted in Fig. 40.3.

The program is given below.

```
#include <gsl/gsl_math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_multifit.h>

int
main()
{
    const size_t n = 1000; /* number of observations */
    const size_t p = 2;   /* number of model parameters */
    size_t i;
    gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
    gsl_matrix *X = gsl_matrix_alloc(n, p);
    gsl_vector *y = gsl_vector_alloc(n);
```

(continues on next page)

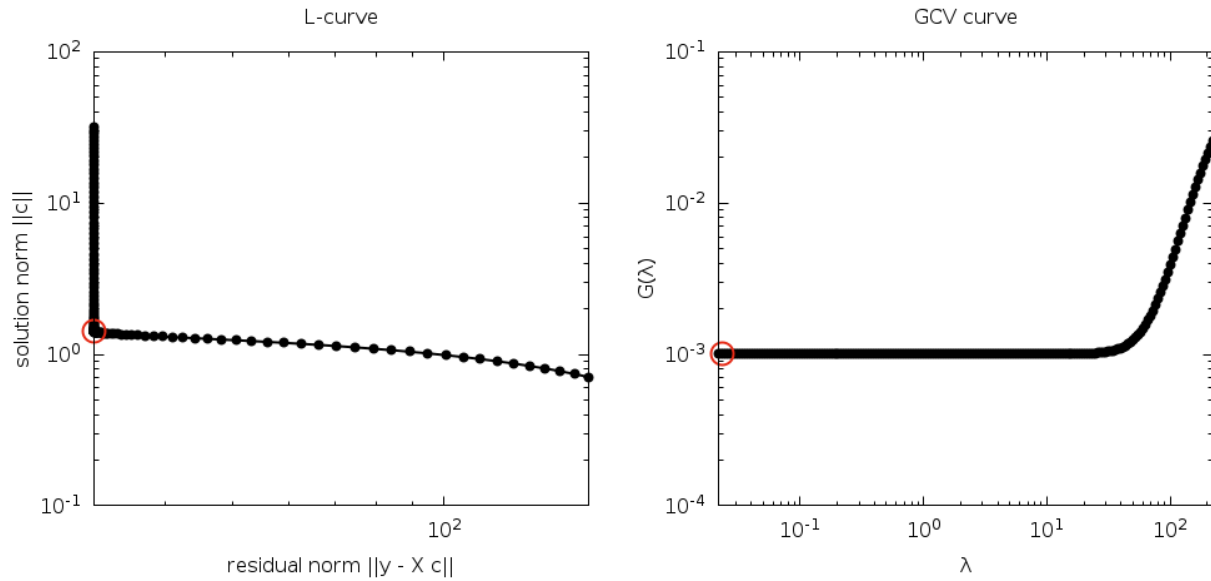


Fig. 40.3: L-curve and GCV curve for example program.

(continued from previous page)

```

for (i = 0; i < n; ++i)
{
    /* generate first random variable u */
    double ui = 5.0 * gsl_ran_gaussian(r, 1.0);

    /* set v = u + noise */
    double vi = ui + gsl_ran_gaussian(r, 0.001);

    /* set y = u + v + noise */
    double yi = ui + vi + gsl_ran_gaussian(r, 1.0);

    /* since u ≈ v, the matrix X is ill-conditioned */
    gsl_matrix_set(X, i, 0, ui);
    gsl_matrix_set(X, i, 1, vi);

    /* rhs vector */
    gsl_vector_set(y, i, yi);
}

{
    const size_t npoints = 200;           /* number of points on L-curve and GCV
↪curve */
    gsl_multifit_linear_workspace *w =
        gsl_multifit_linear_alloc(n, p);
    gsl_vector *c = gsl_vector_alloc(p); /* OLS solution */
    gsl_vector *c_lcurve = gsl_vector_alloc(p); /* regularized solution (L-curve) */
    gsl_vector *c_gcv = gsl_vector_alloc(p); /* regularized solution (GCV) */
    gsl_vector *reg_param = gsl_vector_alloc(npoints);
}

```

(continues on next page)

(continued from previous page)

```

gsl_vector *rho = gsl_vector_alloc(npoints); /* residual norms */
gsl_vector *eta = gsl_vector_alloc(npoints); /* solution norms */
gsl_vector *G = gsl_vector_alloc(npoints); /* GCV function values */
double lambda_l; /* optimal regularization parameter (L-
↳curve) */
double lambda_gcv; /* optimal regularization parameter
↳(GCV) */
double G_gcv; /* G(lambda_gcv) */
size_t reg_idx; /* index of optimal lambda */
double rcond; /* reciprocal condition number of X */
double chisq, rnorm, snorm;

/* compute SVD of X */
gsl_multifit_linear_svd(X, w);

rcond = gsl_multifit_linear_rcond(w);
fprintf(stderr, "matrix condition number = %e\n\n", 1.0 / rcond);

/* unregularized (standard) least squares fit, lambda = 0 */
gsl_multifit_linear_solve(0.0, X, y, c, &rnorm, &snorm, w);
chisq = pow(rnorm, 2.0);

fprintf(stderr, "=== Unregularized fit ===\n");
fprintf(stderr, "best fit: y = %g u + %g v\n",
    gsl_vector_get(c, 0), gsl_vector_get(c, 1));
fprintf(stderr, "residual norm = %g\n", rnorm);
fprintf(stderr, "solution norm = %g\n", snorm);
fprintf(stderr, "chisq/dof = %g\n", chisq / (n - p));

/* calculate L-curve and find its corner */
gsl_multifit_linear_lcurve(y, reg_param, rho, eta, w);
gsl_multifit_linear_lcorner(rho, eta, &reg_idx);

/* store optimal regularization parameter */
lambda_l = gsl_vector_get(reg_param, reg_idx);

/* regularize with lambda_l */
gsl_multifit_linear_solve(lambda_l, X, y, c_lcurve, &rnorm, &snorm, w);
chisq = pow(rnorm, 2.0) + pow(lambda_l * snorm, 2.0);

fprintf(stderr, "\n=== Regularized fit (L-curve) ===\n");
fprintf(stderr, "optimal lambda: %g\n", lambda_l);
fprintf(stderr, "best fit: y = %g u + %g v\n",
    gsl_vector_get(c_lcurve, 0), gsl_vector_get(c_lcurve, 1));
fprintf(stderr, "residual norm = %g\n", rnorm);
fprintf(stderr, "solution norm = %g\n", snorm);
fprintf(stderr, "chisq/dof = %g\n", chisq / (n - p));

/* calculate GCV curve and find its minimum */
gsl_multifit_linear_gcv(y, reg_param, G, &lambda_gcv, &G_gcv, w);

/* regularize with lambda_gcv */

```

(continues on next page)



(continued from previous page)

```

gsl_multifit_linear_solve(lambda_gcv, X, y, c_gcv, &rnorm, &snorm, w);
chisq = pow(rnorm, 2.0) + pow(lambda_gcv * snorm, 2.0);

fprintf(stderr, "\n=== Regularized fit (GCV) ===\n");
fprintf(stderr, "optimal lambda: %g\n", lambda_gcv);
fprintf(stderr, "best fit: y = %g u + %g v\n",
          gsl_vector_get(c_gcv, 0), gsl_vector_get(c_gcv, 1));
fprintf(stderr, "residual norm = %g\n", rnorm);
fprintf(stderr, "solution norm = %g\n", snorm);
fprintf(stderr, "chisq/dof = %g\n", chisq / (n - p));

/* output L-curve and GCV curve */
for (i = 0; i < npoints; ++i)
{
    printf("%e %e %e %e\n",
          gsl_vector_get(reg_param, i),
          gsl_vector_get(rho, i),
          gsl_vector_get(eta, i),
          gsl_vector_get(G, i));
}

/* output L-curve corner point */
printf("\n\n%f %f\n",
       gsl_vector_get(rho, reg_idx),
       gsl_vector_get(eta, reg_idx));

/* output GCV curve corner minimum */
printf("\n\n%e %e\n",
       lambda_gcv,
       G_gcv);

gsl_multifit_linear_free(w);
gsl_vector_free(c);
gsl_vector_free(c_lcurve);
gsl_vector_free(reg_param);
gsl_vector_free(rho);
gsl_vector_free(eta);
gsl_vector_free(G);
}

gsl_rng_free(r);
gsl_matrix_free(X);
gsl_vector_free(y);

return 0;
}

```

## 40.8.4 Regularized Linear Regression Example 2

The following example program minimizes the cost function

$$\|y - Xc\|^2 + \lambda^2 \|x\|^2$$

where  $X$  is the 10-by-8 Hilbert matrix whose entries are given by

$$X_{ij} = \frac{1}{i+j-1}$$

and the right hand side vector is given by  $y = [1, -1, 1, -1, 1, -1, 1, -1, 1, -1]^T$ . Solutions are computed for  $\lambda = 0$  (unregularized) as well as for optimal parameters  $\lambda$  chosen by analyzing the L-curve and GCV curve.

Here is the program output:

```
matrix condition number = 3.565872e+09

=== Unregularized fit ===
residual norm = 2.15376
solution norm = 2.92217e+09
chisq/dof = 2.31934

=== Regularized fit (L-curve) ===
optimal lambda: 7.11407e-07
residual norm = 2.60386
solution norm = 424507
chisq/dof = 3.43565

=== Regularized fit (GCV) ===
optimal lambda: 1.72278
residual norm = 3.1375
solution norm = 0.139357
chisq/dof = 4.95076
```

Here we see the unregularized solution results in a large solution norm due to the ill-conditioned matrix. The L-curve solution finds a small value of  $\lambda = 7.11e - 7$  which still results in a badly conditioned system and a large solution norm. The GCV method finds a parameter  $\lambda = 1.72$  which results in a well-conditioned system and small solution norm.

The L-curve and its computed corner, as well as the GCV curve and its minimum are plotted in Fig. 40.4.

The program is given below.

```
#include <gsl/gsl_math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_blas.h>

static int
hilbert_matrix(gsl_matrix * m)
{
    const size_t N = m->size1;
    const size_t M = m->size2;
    size_t i, j;
```

(continues on next page)

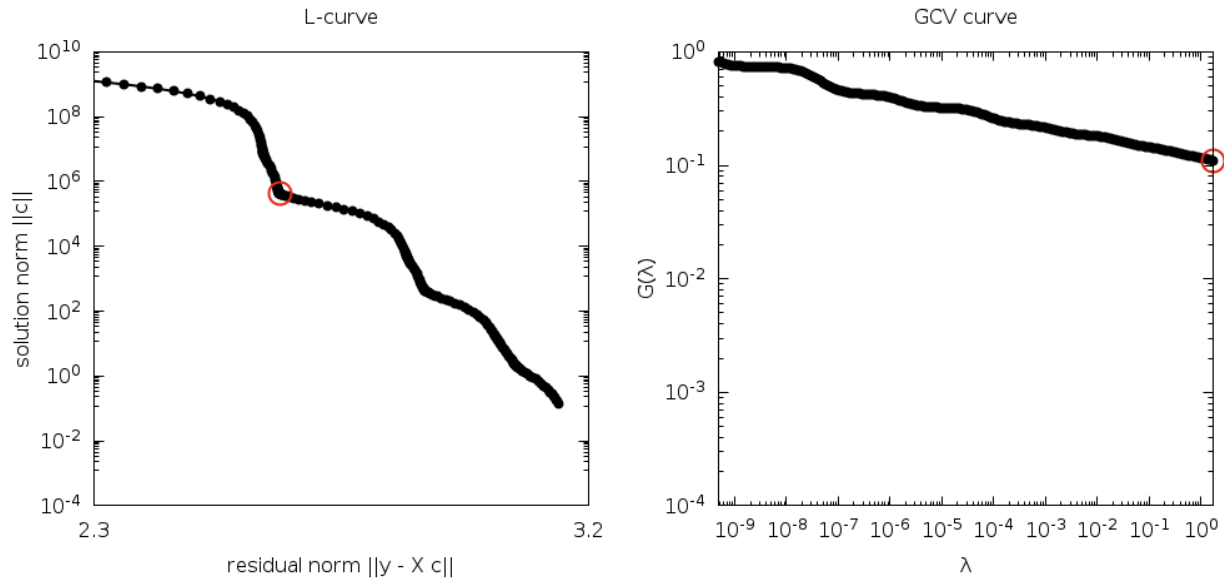


Fig. 40.4: L-curve and GCV curve for example program.

(continued from previous page)

```

for (i = 0; i < N; i++)
{
    for (j = 0; j < M; j++)
    {
        gsl_matrix_set(m, i, j, 1.0/(i+j+1.0));
    }
}

return GSL_SUCCESS;
}

int
main()
{
    const size_t n = 10; /* number of observations */
    const size_t p = 8; /* number of model parameters */
    size_t i;
    gsl_matrix *X = gsl_matrix_alloc(n, p);
    gsl_vector *y = gsl_vector_alloc(n);

    /* construct Hilbert matrix and rhs vector */
    hilbert_matrix(X);

    {
        double val = 1.0;
        for (i = 0; i < n; ++i)
        {
            gsl_vector_set(y, i, val);
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        val *= -1.0;
    }
}

{
    const size_t npoints = 200;           /* number of points on L-curve and GCV */
↪curve */
    gsl_multifit_linear_workspace *w =
        gsl_multifit_linear_alloc(n, p);
    gsl_vector *c = gsl_vector_alloc(p); /* OLS solution */
    gsl_vector *c_lcurve = gsl_vector_alloc(p); /* regularized solution (L-curve) */
    gsl_vector *c_gcv = gsl_vector_alloc(p); /* regularized solution (GCV) */
    gsl_vector *reg_param = gsl_vector_alloc(npoints);
    gsl_vector *rho = gsl_vector_alloc(npoints); /* residual norms */
    gsl_vector *eta = gsl_vector_alloc(npoints); /* solution norms */
    gsl_vector *G = gsl_vector_alloc(npoints); /* GCV function values */
    double lambda_l; /* optimal regularization parameter (L-
↪curve) */
    double lambda_gcv; /* optimal regularization parameter
↪(GCV) */
    double G_gcv; /* G(lambda_gcv) */
    size_t reg_idx; /* index of optimal lambda */
    double rcond; /* reciprocal condition number of X */
    double chisq, rnorm, snorm;

    /* compute SVD of X */
    gsl_multifit_linear_svd(X, w);

    rcond = gsl_multifit_linear_rcond(w);
    fprintf(stderr, "matrix condition number = %e\n", 1.0 / rcond);

    /* unregularized (standard) least squares fit, lambda = 0 */
    gsl_multifit_linear_solve(0.0, X, y, c, &rnorm, &snorm, w);
    chisq = pow(rnorm, 2.0);

    fprintf(stderr, "\n=== Unregularized fit ===\n");
    fprintf(stderr, "residual norm = %g\n", rnorm);
    fprintf(stderr, "solution norm = %g\n", snorm);
    fprintf(stderr, "chisq/dof = %g\n", chisq / (n - p));

    /* calculate L-curve and find its corner */
    gsl_multifit_linear_lcurve(y, reg_param, rho, eta, w);
    gsl_multifit_linear_lcorner(rho, eta, &reg_idx);

    /* store optimal regularization parameter */
    lambda_l = gsl_vector_get(reg_param, reg_idx);

    /* regularize with lambda_l */
    gsl_multifit_linear_solve(lambda_l, X, y, c_lcurve, &rnorm, &snorm, w);
    chisq = pow(rnorm, 2.0) + pow(lambda_l * snorm, 2.0);

    fprintf(stderr, "\n=== Regularized fit (L-curve) ===\n");

```

(continues on next page)

(continued from previous page)

```

fprintf(stderr, "optimal lambda: %g\n", lambda_l);
fprintf(stderr, "residual norm = %g\n", rnorm);
fprintf(stderr, "solution norm = %g\n", snorm);
fprintf(stderr, "chisq/dof = %g\n", chisq / (n - p));

/* calculate GCV curve and find its minimum */
gsl_multifit_linear_gcv(y, reg_param, G, &lambda_gcv, &G_gcv, w);

/* regularize with lambda_gcv */
gsl_multifit_linear_solve(lambda_gcv, X, y, c_gcv, &rnorm, &snorm, w);
chisq = pow(rnorm, 2.0) + pow(lambda_gcv * snorm, 2.0);

fprintf(stderr, "\n=== Regularized fit (GCV) ===\n");
fprintf(stderr, "optimal lambda: %g\n", lambda_gcv);
fprintf(stderr, "residual norm = %g\n", rnorm);
fprintf(stderr, "solution norm = %g\n", snorm);
fprintf(stderr, "chisq/dof = %g\n", chisq / (n - p));

/* output L-curve and GCV curve */
for (i = 0; i < npoints; ++i)
{
    printf("%e %e %e %e\n",
           gsl_vector_get(reg_param, i),
           gsl_vector_get(rho, i),
           gsl_vector_get(eta, i),
           gsl_vector_get(G, i));
}

/* output L-curve corner point */
printf("\n\n%f %f\n",
       gsl_vector_get(rho, reg_idx),
       gsl_vector_get(eta, reg_idx));

/* output GCV curve corner minimum */
printf("\n\n%e %e\n",
       lambda_gcv,
       G_gcv);

gsl_multifit_linear_free(w);
gsl_vector_free(c);
gsl_vector_free(c_lcurve);
gsl_vector_free(reg_param);
gsl_vector_free(rho);
gsl_vector_free(eta);
gsl_vector_free(G);
}

gsl_matrix_free(X);
gsl_vector_free(y);

return 0;
}

```

## 40.8.5 Robust Linear Regression Example

The next program demonstrates the advantage of robust least squares on a dataset with outliers. The program generates linear  $(x, y)$  data pairs on the line  $y = 1.45x + 3.88$ , adds some random noise, and inserts 3 outliers into the dataset. Both the robust and ordinary least squares (OLS) coefficients are computed for comparison.

```
#include <stdio.h>
#include <gsl/gsl_multifit.h>
#include <gsl/gsl_randist.h>

int
dofit(const gsl_multifit_robust_type *T,
      const gsl_matrix *X, const gsl_vector *y,
      gsl_vector *c, gsl_matrix *cov)
{
    int s;
    gsl_multifit_robust_workspace * work
        = gsl_multifit_robust_alloc (T, X->size1, X->size2);

    s = gsl_multifit_robust (X, y, c, cov, work);
    gsl_multifit_robust_free (work);

    return s;
}

int
main (int argc, char **argv)
{
    size_t i;
    size_t n;
    const size_t p = 2; /* linear fit */
    gsl_matrix *X, *cov;
    gsl_vector *x, *y, *c, *c_ols;
    const double a = 1.45; /* slope */
    const double b = 3.88; /* intercept */
    gsl_rng *r;

    if (argc != 2)
    {
        fprintf (stderr, "usage: roffit n\n");
        exit (-1);
    }

    n = atoi (argv[1]);

    X = gsl_matrix_alloc (n, p);
    x = gsl_vector_alloc (n);
    y = gsl_vector_alloc (n);

    c = gsl_vector_alloc (p);
    c_ols = gsl_vector_alloc (p);
    cov = gsl_matrix_alloc (p, p);
```

(continues on next page)

(continued from previous page)

```

r = gsl_rng_alloc(gsl_rng_default);

/* generate linear dataset */
for (i = 0; i < n - 3; i++)
{
    double dx = 10.0 / (n - 1.0);
    double ei = gsl_rng_uniform(r);
    double xi = -5.0 + i * dx;
    double yi = a * xi + b;

    gsl_vector_set (x, i, xi);
    gsl_vector_set (y, i, yi + ei);
}

/* add a few outliers */
gsl_vector_set(x, n - 3, 4.7);
gsl_vector_set(y, n - 3, -8.3);

gsl_vector_set(x, n - 2, 3.5);
gsl_vector_set(y, n - 2, -6.7);

gsl_vector_set(x, n - 1, 4.1);
gsl_vector_set(y, n - 1, -6.0);

/* construct design matrix X for linear fit */
for (i = 0; i < n; ++i)
{
    double xi = gsl_vector_get(x, i);

    gsl_matrix_set (X, i, 0, 1.0);
    gsl_matrix_set (X, i, 1, xi);
}

/* perform robust and OLS fit */
dofit(gsl_multifit_robust_ols, X, y, c_ols, cov);
dofit(gsl_multifit_robust_bisquare, X, y, c, cov);

/* output data and model */
for (i = 0; i < n; ++i)
{
    double xi = gsl_vector_get(x, i);
    double yi = gsl_vector_get(y, i);
    gsl_vector_view v = gsl_matrix_row(X, i);
    double y_ols, y_rob, y_err;

    gsl_multifit_robust_est(&v.vector, c, cov, &y_rob, &y_err);
    gsl_multifit_robust_est(&v.vector, c_ols, cov, &y_ols, &y_err);

    printf("%g %g %g %g\n", xi, yi, y_rob, y_ols);
}

#define C(i) (gsl_vector_get(c, (i)))

```

(continues on next page)

(continued from previous page)

```

#define COV(i,j) (gsl_matrix_get(cov,(i),(j)))

{
    printf ("# best fit: Y = %g + %g X\n",
           C(0), C(1));

    printf ("# covariance matrix:\n");
    printf ("# [  %+.5e, %+.5e\n",
           COV(0,0), COV(0,1));
    printf ("#   %+.5e, %+.5e\n",
           COV(1,0), COV(1,1));
}

gsl_matrix_free (X);
gsl_vector_free (x);
gsl_vector_free (y);
gsl_vector_free (c);
gsl_vector_free (c_ols);
gsl_matrix_free (cov);
gsl_rng_free(r);

return 0;
}

```

The output from the program is shown in Fig. 40.5.

## 40.8.6 Large Dense Linear Regression Example

The following program demonstrates the large dense linear least squares solvers. This example is adapted from Trefethen and Bau, and fits the function  $f(t) = \exp(\sin^3(10t))$  on the interval  $[0, 1]$  with a degree 15 polynomial. The program generates  $n = 50000$  equally spaced points  $t_i$  on this interval, calculates the function value and adds random noise to determine the observation value  $y_i$ . The entries of the least squares matrix are  $X_{ij} = t_i^j$ , representing a polynomial fit. The matrix is highly ill-conditioned, with a condition number of about  $2.4 \cdot 10^{11}$ . The program accumulates the matrix into the least squares system in 5 blocks, each with 10000 rows. This way the full matrix  $X$  is never stored in memory. We solve the system with both the normal equations and TSQR methods. The results are shown in Fig. 40.6. In the top left plot, the TSQR solution provides a reasonable agreement to the exact solution, while the normal equations method fails completely since the Cholesky factorization fails due to the ill-conditioning of the matrix. In the bottom left plot, we show the L-curve calculated from TSQR, which exhibits multiple corners. In the top right panel, we plot a regularized solution using  $\lambda = 10^{-5}$ . The TSQR and normal solutions now agree, however they are unable to provide a good fit due to the damping. This indicates that for some ill-conditioned problems, regularizing the normal equations does not improve the solution. This is further illustrated in the bottom right panel, where we plot the L-curve calculated from the normal equations. The curve agrees with the TSQR curve for larger damping parameters, but for small  $\lambda$ , the normal equations approach cannot provide accurate solution vectors leading to numerical inaccuracies in the left portion of the curve.

```

#include <gsl/gsl_math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_multifit.h>

```

(continues on next page)



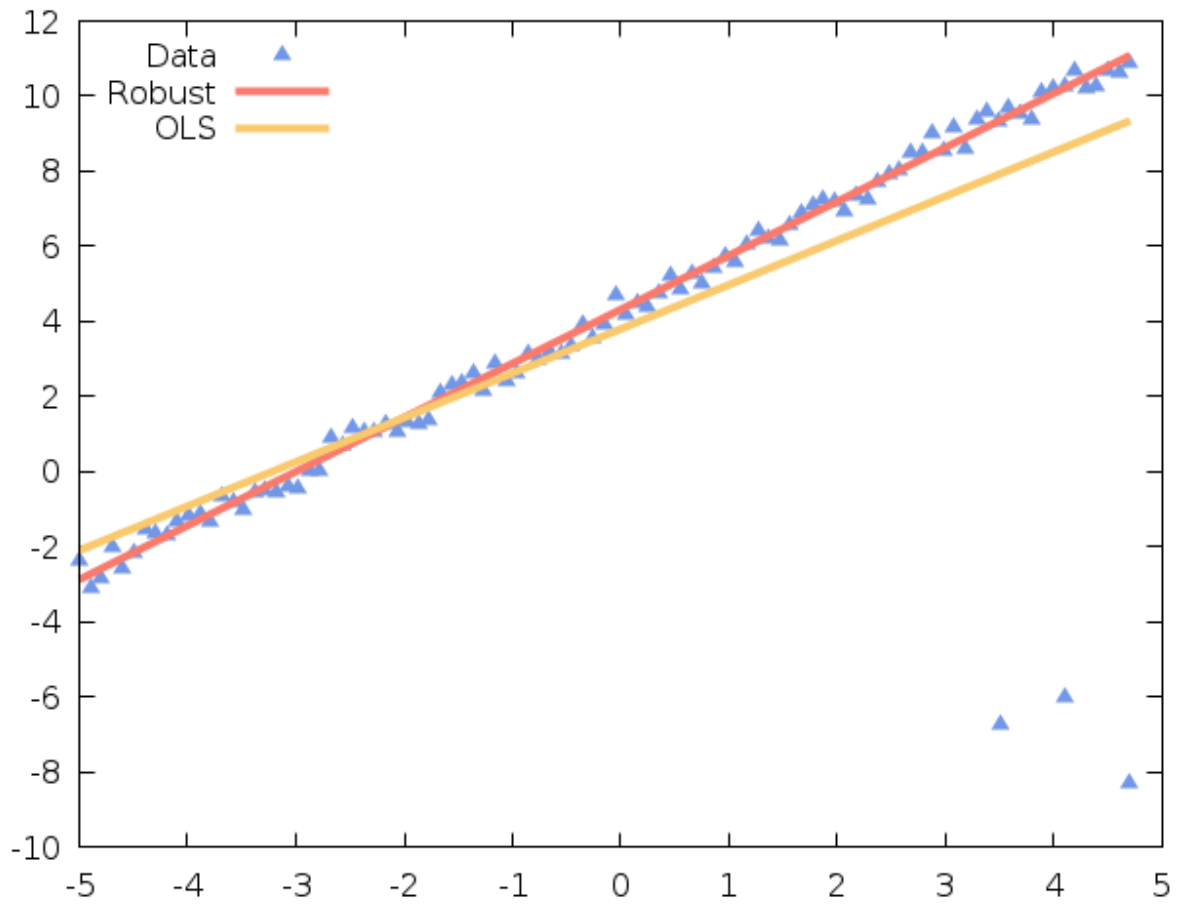


Fig. 40.5: Linear fit to dataset with outliers.

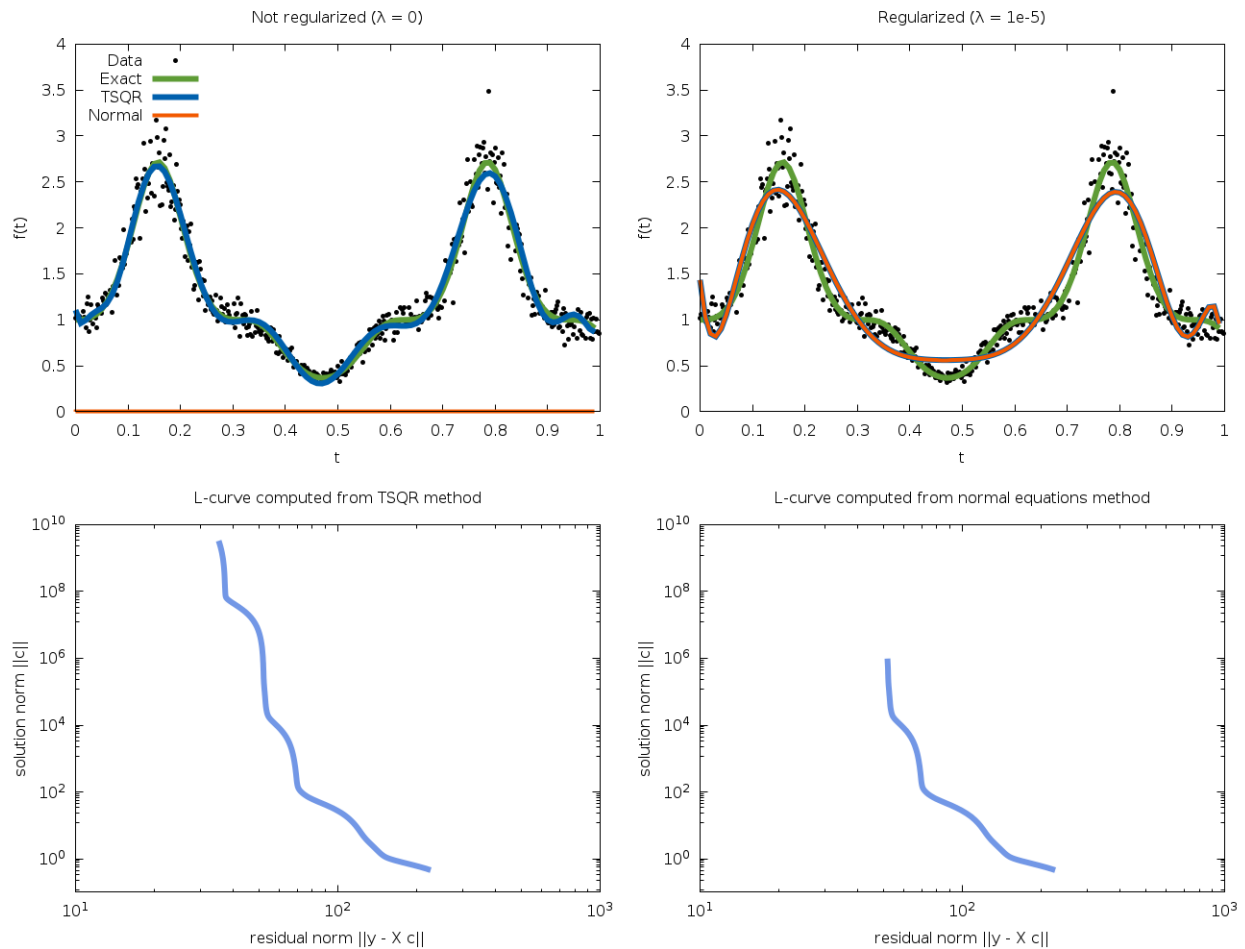


Fig. 40.6: Top left: unregularized solutions; top right: regularized solutions; bottom left: L-curve for TSQR method; bottom right: L-curve from normal equations method.

(continued from previous page)

```

#include <gsl/gsl_multilarge.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_errno.h>

/* function to be fitted */
double
func(const double t)
{
    double x = sin(10.0 * t);
    return exp(x*x*x);
}

/* construct a row of the least squares matrix */
int
build_row(const double t, gsl_vector *row)
{
    const size_t p = row->size;
    double Xj = 1.0;
    size_t j;

    for (j = 0; j < p; ++j)
    {
        gsl_vector_set(row, j, Xj);
        Xj *= t;
    }

    return 0;
}

int
solve_system(const int print_data, const gsl_multilarge_linear_type * T,
             const double lambda, const size_t n, const size_t p,
             gsl_vector * c)
{
    const size_t nblock = 5;          /* number of blocks to accumulate */
    const size_t nrows = n / nblock; /* number of rows per block */
    gsl_multilarge_linear_workspace * w =
        gsl_multilarge_linear_alloc(T, p);
    gsl_matrix *X = gsl_matrix_alloc(nrows, p);
    gsl_vector *y = gsl_vector_alloc(nrows);
    gsl_rng *r = gsl_rng_alloc(gsl_rng_default);
    const size_t nlcurve = 200;
    gsl_vector *reg_param = gsl_vector_alloc(nlcurve);
    gsl_vector *rho = gsl_vector_calloc(nlcurve);
    gsl_vector *eta = gsl_vector_calloc(nlcurve);
    size_t rowidx = 0;
    double rnorm, snorm, rcond;
    double t = 0.0;
    double dt = 1.0 / (n - 1.0);

    while (rowidx < n)
    {

```

(continues on next page)

(continued from previous page)

```

size_t nleft = n - rowidx;          /* number of rows left to accumulate */
size_t nr = GSL_MIN(nrows, nleft); /* number of rows in this block */
gsl_matrix_view Xv = gsl_matrix_submatrix(X, 0, 0, nr, p);
gsl_vector_view yv = gsl_vector_subvector(y, 0, nr);
size_t i;

/* build (X,y) block with 'nr' rows */
for (i = 0; i < nr; ++i)
{
    gsl_vector_view row = gsl_matrix_row(&Xv.matrix, i);
    double fi = func(t);
    double ei = gsl_ran_gaussian(r, 0.1 * fi); /* noise */
    double yi = fi + ei;

    /* construct this row of LS matrix */
    build_row(t, &row.vector);

    /* set right hand side value with added noise */
    gsl_vector_set(&yv.vector, i, yi);

    if (print_data && (i % 100 == 0))
        printf("%f %f\n", t, yi);

    t += dt;
}

/* accumulate (X,y) block into LS system */
gsl_multilarge_linear_accumulate(&Xv.matrix, &yv.vector, w);

rowidx += nr;
}

if (print_data)
    printf("\n\n");

/* compute L-curve */
gsl_multilarge_linear_lcurve(reg_param, rho, eta, w);

/* solve large LS system and store solution in c */
gsl_multilarge_linear_solve(lambda, c, &rnorm, &snorm, w);

/* compute reciprocal condition number */
gsl_multilarge_linear_rcond(&rcond, w);

fprintf(stderr, "=== Method %s ===\n", gsl_multilarge_linear_name(w));
fprintf(stderr, "condition number = %e\n", 1.0 / rcond);
fprintf(stderr, "residual norm    = %e\n", rnorm);
fprintf(stderr, "solution norm    = %e\n", snorm);

/* output L-curve */
{
    size_t i;

```

(continues on next page)

(continued from previous page)

```

    for (i = 0; i < nlcurve; ++i)
    {
        printf("%.12e %.12e %.12e\n",
            gsl_vector_get(reg_param, i),
            gsl_vector_get(rho, i),
            gsl_vector_get(eta, i));
    }
    printf("\n\n");
}

gsl_matrix_free(X);
gsl_vector_free(y);
gsl_multilarge_linear_free(w);
gsl_rng_free(r);
gsl_vector_free(reg_param);
gsl_vector_free(rho);
gsl_vector_free(eta);

return 0;
}

int
main(int argc, char *argv[])
{
    const size_t n = 50000;    /* number of observations */
    const size_t p = 16;     /* polynomial order + 1 */
    double lambda = 0.0;     /* regularization parameter */
    gsl_vector *c_tsqr = gsl_vector_calloc(p);
    gsl_vector *c_normal = gsl_vector_calloc(p);

    if (argc > 1)
        lambda = atof(argv[1]);

    /* turn off error handler so normal equations method won't abort */
    gsl_set_error_handler_off();

    /* solve system with TSQR method */
    solve_system(1, gsl_multilarge_linear_tsqr, lambda, n, p, c_tsqr);

    /* solve system with Normal equations method */
    solve_system(0, gsl_multilarge_linear_normal, lambda, n, p, c_normal);

    /* output solutions */
    {
        gsl_vector *v = gsl_vector_alloc(p);
        double t;

        for (t = 0.0; t <= 1.0; t += 0.01)
        {
            double f_exact = func(t);
            double f_tsqr, f_normal;

```

(continues on next page)

(continued from previous page)

```
    build_row(t, v);
    gsl_blas_ddot(v, c_tsqr, &f_tsqr);
    gsl_blas_ddot(v, c_normal, &f_normal);

    printf("%f %e %e %e\n", t, f_exact, f_tsqr, f_normal);
}

gsl_vector_free(v);
}

gsl_vector_free(c_tsqr);
gsl_vector_free(c_normal);

return 0;
}
```

## 40.9 References and Further Reading

A summary of formulas and techniques for least squares fitting can be found in the “Statistics” chapter of the Annual Review of Particle Physics prepared by the Particle Data Group,

- *Review of Particle Properties*, R.M. Barnett et al., Physical Review D54, 1 (1996) <http://pdg.lbl.gov>

The Review of Particle Physics is available online at the website given above.

The tests used to prepare these routines are based on the NIST Statistical Reference Datasets. The datasets and their documentation are available from NIST at the following website,

<http://www.nist.gov/itl/div898/strd/index.html>

More information on Tikhonov regularization can be found in

- Hansen, P. C. (1998), Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion. SIAM Monogr. on Mathematical Modeling and Computation, Society for Industrial and Applied Mathematics
- M. Rezghi and S. M. Hosseini (2009), A new variant of L-curve for Tikhonov regularization, Journal of Computational and Applied Mathematics, Volume 231, Issue 2, pages 914-924.

The GSL implementation of robust linear regression closely follows the publications

- DuMouchel, W. and F. O’Brien (1989), “Integrating a robust option into a multiple regression computing environment,” Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface, American Statistical Association
- Street, J.O., R.J. Carroll, and D. Ruppert (1988), “A note on computing robust regression estimates via iteratively reweighted least squares,” The American Statistician, v. 42, pp. 152-154.

More information about the normal equations and TSQR approach for solving large linear least squares systems can be found in the publications

- Trefethen, L. N. and Bau, D. (1997), “Numerical Linear Algebra”, SIAM.
- Demmel, J., Grigori, L., Hoemmen, M. F., and Langou, J. “Communication-optimal parallel and sequential QR and LU factorizations”, UCB Technical Report No. UCB/EECS-2008-89, 2008.

## NONLINEAR LEAST-SQUARES FITTING

This chapter describes functions for multidimensional nonlinear least-squares fitting. There are generally two classes of algorithms for solving nonlinear least squares problems, which fall under line search methods and trust region methods. GSL currently implements only trust region methods and provides the user with full access to intermediate steps of the iteration. The user also has the ability to tune a number of parameters which affect low-level aspects of the algorithm which can help to accelerate convergence for the specific problem at hand. GSL provides two separate interfaces for nonlinear least squares fitting. The first is designed for small to moderate sized problems, and the second is designed for very large problems, which may or may not have significant sparse structure.

The header file `gsl_multifit_nlinear.h` contains prototypes for the multidimensional nonlinear fitting functions and related declarations relating to the small to moderate sized systems.

The header file `gsl_multilarge_nlinear.h` contains prototypes for the multidimensional nonlinear fitting functions and related declarations relating to large systems.

### 41.1 Overview

The problem of multidimensional nonlinear least-squares fitting requires the minimization of the squared residuals of  $n$  functions,  $f_i$ , in  $p$  parameters,  $x_i$ ,

$$\begin{aligned}\Phi(x) &= \frac{1}{2} \|f(x)\|^2 \\ &= \frac{1}{2} \sum_{i=1}^n f_i(x_1, \dots, x_p)^2\end{aligned}$$

In trust region methods, the objective (or cost) function  $\Phi(x)$  is approximated by a model function  $m_k(\delta)$  in the vicinity of some point  $x_k$ . The model function is often simply a second order Taylor series expansion around the point  $x_k$ , ie:

$$\Phi(x_k + \delta) \approx m_k(\delta) = \Phi(x_k) + g_k^T \delta + \frac{1}{2} \delta^T B_k \delta$$

where  $g_k = \nabla \Phi(x_k) = J^T f$  is the gradient vector at the point  $x_k$ ,  $B_k = \nabla^2 \Phi(x_k)$  is the Hessian matrix at  $x_k$ , or some approximation to it, and  $J$  is the  $n$ -by- $p$  Jacobian matrix

$$J_{ij} = \partial f_i / \partial x_j$$

In order to find the next step  $\delta$ , we minimize the model function  $m_k(\delta)$ , but search for solutions only within a region where we trust that  $m_k(\delta)$  is a good approximation to the objective function  $\Phi(x_k + \delta)$ . In other words, we seek a solution of the trust region subproblem (TRS)

$$\min_{\delta \in R^p} m_k(\delta) = \Phi(x_k) + g_k^T \delta + \frac{1}{2} \delta^T B_k \delta, \quad \text{s.t.} \quad \|D_k \delta\| \leq \Delta_k$$

where  $\Delta_k > 0$  is the trust region radius and  $D_k$  is a scaling matrix. If  $D_k = I$ , then the trust region is a ball of radius  $\Delta_k$  centered at  $x_k$ . In some applications, the parameter vector  $x$  may have widely different scales. For example, one parameter might be a temperature on the order of  $10^3$  K, while another might be a length on the order of  $10^{-6}$  m. In such cases, a spherical trust region may not be the best choice, since if  $\Phi$  changes rapidly along directions with one scale, and more slowly along directions with a different scale, the model function  $m_k$  may be a poor approximation to  $\Phi$  along the rapidly changing directions. In such problems, it may be best to use an elliptical trust region, by setting  $D_k$  to a diagonal matrix whose entries are designed so that the scaled step  $D_k\delta$  has entries of approximately the same order of magnitude.

The trust region subproblem above normally amounts to solving a linear least squares system (or multiple systems) for the step  $\delta$ . Once  $\delta$  is computed, it is checked whether or not it reduces the objective function  $\Phi(x)$ . A useful statistic for this is to look at the ratio

$$\rho_k = \frac{\Phi(x_k) - \Phi(x_k + \delta_k)}{m_k(0) - m_k(\delta_k)}$$

where the numerator is the actual reduction of the objective function due to the step  $\delta_k$ , and the denominator is the predicted reduction due to the model  $m_k$ . If  $\rho_k$  is negative, it means that the step  $\delta_k$  increased the objective function and so it is rejected. If  $\rho_k$  is positive, then we have found a step which reduced the objective function and it is accepted. Furthermore, if  $\rho_k$  is close to 1, then this indicates that the model function is a good approximation to the objective function in the trust region, and so on the next iteration the trust region is enlarged in order to take more ambitious steps. When a step is rejected, the trust region is made smaller and the TRS is solved again. An outline for the general trust region method used by GSL can now be given.

### Trust Region Algorithm

1. Initialize: given  $x_0$ , construct  $m_0(\delta)$ ,  $D_0$  and  $\Delta_0 > 0$
2. For  $k = 0, 1, 2, \dots$ 
  - a. If converged, then stop
  - b. Solve TRS for trial step  $\delta_k$
  - c. Evaluate trial step by computing  $\rho_k$ 
    - 1). if step is accepted, set  $x_{k+1} = x_k + \delta_k$  and increase radius,  $\Delta_{k+1} = \alpha\Delta_k$
    - 2). if step is rejected, set  $x_{k+1} = x_k$  and decrease radius,  $\Delta_{k+1} = \frac{\Delta_k}{\beta}$ ; goto 2(b)
  - d. Construct  $m_{k+1}(\delta)$  and  $D_{k+1}$

GSL offers the user a number of different algorithms for solving the trust region subproblem in 2(b), as well as different choices of scaling matrices  $D_k$  and different methods of updating the trust region radius  $\Delta_k$ . Therefore, while reasonable default methods are provided, the user has a lot of control to fine-tune the various steps of the algorithm for their specific problem.

## 41.2 Solving the Trust Region Subproblem (TRS)

Below we describe the methods available for solving the trust region subproblem. The methods available provide either exact or approximate solutions to the trust region subproblem. In all algorithms below, the Hessian matrix  $B_k$  is approximated as  $B_k \approx J_k^T J_k$ , where  $J_k = J(x_k)$ . In all methods, the solution of the TRS involves solving a linear least squares system involving the Jacobian matrix. For small to moderate sized problems (`gsl_multifit_nlinear` interface), this is accomplished by factoring the full Jacobian matrix, which is provided by the user, with the Cholesky, QR, or SVD decompositions. For large systems (`gsl_multilarge_nlinear` interface), the user has two choices. One is to solve the system iteratively, without needing to store the full Jacobian matrix in memory. With this method, the user must provide a routine to calculate the matrix-vector products  $Ju$  or  $J^T u$  for a given vector  $u$ . This iterative method is particularly useful for systems where the Jacobian has sparse structure, since forming matrix-vector products



can be done cheaply. The second option for large systems involves forming the normal equations matrix  $J^T J$  and then factoring it using a Cholesky decomposition. The normal equations matrix is  $p$ -by- $p$ , typically much smaller than the full  $n$ -by- $p$  Jacobian, and can usually be stored in memory even if the full Jacobian matrix cannot. This option is useful for large, dense systems, or if the iterative method has difficulty converging.

### 41.2.1 Levenberg-Marquardt

There is a theorem which states that if  $\delta_k$  is a solution to the trust region subproblem given above, then there exists  $\mu_k \geq 0$  such that

$$(B_k + \mu_k D_k^T D_k) \delta_k = -g_k$$

with  $\mu_k(\Delta_k - \|D_k \delta_k\|) = 0$ . This forms the basis of the Levenberg-Marquardt algorithm, which controls the trust region size by adjusting the parameter  $\mu_k$  rather than the radius  $\Delta_k$  directly. For each radius  $\Delta_k$ , there is a unique parameter  $\mu_k$  which solves the TRS, and they have an inverse relationship, so that large values of  $\mu_k$  correspond to smaller trust regions, while small values of  $\mu_k$  correspond to larger trust regions.

With the approximation  $B_k \approx J_k^T J_k$ , on each iteration, in order to calculate the step  $\delta_k$ , the following linear least squares problem is solved:

$$\begin{bmatrix} J_k \\ \sqrt{\mu_k} D_k \end{bmatrix} \delta_k = - \begin{bmatrix} f_k \\ 0 \end{bmatrix}$$

If the step  $\delta_k$  is accepted, then  $\mu_k$  is decreased on the next iteration in order to take a larger step, otherwise it is increased to take a smaller step. The Levenberg-Marquardt algorithm provides an exact solution of the trust region subproblem, but typically has a higher computational cost per iteration than the approximate methods discussed below, since it may need to solve the least squares system above several times for different values of  $\mu_k$ .

### 41.2.2 Levenberg-Marquardt with Geodesic Acceleration

This method applies a so-called geodesic acceleration correction to the standard Levenberg-Marquardt step  $\delta_k$  (Transtrum et al, 2011). By interpreting  $\delta_k$  as a first order step along a geodesic in the model parameter space (ie: a velocity  $\delta_k = v_k$ ), the geodesic acceleration  $a_k$  is a second order correction along the geodesic which is determined by solving the linear least squares system

$$\begin{bmatrix} J_k \\ \sqrt{\mu_k} D_k \end{bmatrix} a_k = - \begin{bmatrix} f_{vv}(x_k) \\ 0 \end{bmatrix}$$

where  $f_{vv}$  is the second directional derivative of the residual vector in the velocity direction  $v$ ,  $f_{vv}(x) = D_v^2 f = \sum_{\alpha\beta} v_\alpha v_\beta \partial_\alpha \partial_\beta f(x)$ , where  $\alpha$  and  $\beta$  are summed over the  $p$  parameters. The new total step is then  $\delta'_k = v_k + \frac{1}{2} a_k$ . The second order correction  $a_k$  can be calculated with a modest additional cost, and has been shown to dramatically reduce the number of iterations (and expensive Jacobian evaluations) required to reach convergence on a variety of different problems. In order to utilize the geodesic acceleration, the user must supply a function which provides the second directional derivative vector  $f_{vv}(x)$ , or alternatively the library can use a finite difference method to estimate this vector with one additional function evaluation of  $f(x + hv)$  where  $h$  is a tunable step size (see the `h_fvv` parameter description).

### 41.2.3 Dogleg

This is Powell's dogleg method, which finds an approximate solution to the trust region subproblem, by restricting its search to a piecewise linear "dogleg" path, composed of the origin, the Cauchy point which represents the model minimizer along the steepest descent direction, and the Gauss-Newton point, which is the overall minimizer of the unconstrained model. The Gauss-Newton step is calculated by solving

$$J_k \delta_{gn} = -f_k$$

which is the main computational task for each iteration, but only needs to be performed once per iteration. If the Gauss-Newton point is inside the trust region, it is selected as the step. If it is outside, the method then calculates the Cauchy point, which is located along the gradient direction. If the Cauchy point is also outside the trust region, the method assumes that it is still far from the minimum and so proceeds along the gradient direction, truncating the step at the trust region boundary. If the Cauchy point is inside the trust region, with the Gauss-Newton point outside, the method uses a dogleg step, which is a linear combination of the gradient direction and the Gauss-Newton direction, stopping at the trust region boundary.

### 41.2.4 Double Dogleg

This method is an improvement over the classical dogleg algorithm, which attempts to include information about the Gauss-Newton step while the iteration is still far from the minimum. When the Cauchy point is inside the trust region and the Gauss-Newton point is outside, the method computes a scaled Gauss-Newton point and then takes a dogleg step between the Cauchy point and the scaled Gauss-Newton point. The scaling is calculated to ensure that the reduction in the model  $m_k$  is about the same as the reduction provided by the Cauchy point.

### 41.2.5 Two Dimensional Subspace

The dogleg methods restrict the search for the TRS solution to a 1D curve defined by the Cauchy and Gauss-Newton points. An improvement to this is to search for a solution using the full two dimensional subspace spanned by the Cauchy and Gauss-Newton directions. The dogleg path is of course inside this subspace, and so this method solves the TRS at least as accurately as the dogleg methods. Since this method searches a larger subspace for a solution, it can converge more quickly than dogleg on some problems. Because the subspace is only two dimensional, this method is very efficient and the main computation per iteration is to determine the Gauss-Newton point.

### 41.2.6 Steihaug-Toint Conjugate Gradient

One difficulty of the dogleg methods is calculating the Gauss-Newton step when the Jacobian matrix is singular. The Steihaug-Toint method also computes a generalized dogleg step, but avoids solving for the Gauss-Newton step directly, instead using an iterative conjugate gradient algorithm. This method performs well at points where the Jacobian is singular, and is also suitable for large-scale problems where factoring the Jacobian matrix could be prohibitively expensive.

## 41.3 Weighted Nonlinear Least-Squares

Weighted nonlinear least-squares fitting minimizes the function

$$\begin{aligned}\Phi(x) &= \frac{1}{2} \|f\|_W^2 \\ &= \frac{1}{2} \sum_{i=1}^n w_i f_i(x_1, \dots, x_p)^2\end{aligned}$$

where  $W = \text{diag}(w_1, w_2, \dots, w_n)$  is the weighting matrix, and  $\|f\|_W^2 = f^T W f$ . The weights  $w_i$  are commonly defined as  $w_i = 1/\sigma_i^2$ , where  $\sigma_i$  is the error in the  $i$ -th measurement. A simple change of variables  $\tilde{f} = W^{\frac{1}{2}} f$  yields  $\Phi(x) = \frac{1}{2} \|\tilde{f}\|^2$ , which is in the same form as the unweighted case. The user can either perform this transform directly on their function residuals and Jacobian, or use the `gsl_multifit_nlinear_winit()` interface which automatically performs the correct scaling. To manually perform this transformation, the residuals and Jacobian should be modified according to

$$\begin{aligned}\tilde{f}_i &= \sqrt{w_i} f_i = \frac{f_i}{\sigma_i} \\ \tilde{J}_{ij} &= \sqrt{w_i} \frac{\partial f_i}{\partial x_j} = \frac{1}{\sigma_i} \frac{\partial f_i}{\partial x_j}\end{aligned}$$

For large systems, the user must perform their own weighting.

## 41.4 Tunable Parameters

The user can tune nearly all aspects of the iteration at allocation time. For the `gsl_multifit_nlinear` interface, the user may modify the `gsl_multifit_nlinear_parameters` structure, which is defined as follows:

type `gsl_multifit_nlinear_parameters`

```
typedef struct
{
    const gsl_multifit_nlinear_trs *trs;          /* trust region subproblem method */
    const gsl_multifit_nlinear_scale *scale;     /* scaling method */
    const gsl_multifit_nlinear_solver *solver;   /* solver method */
    gsl_multifit_nlinear_fdtype fdtype;         /* finite difference method */
    double factor_up;                            /* factor for increasing trust radius */
    /* */
    double factor_down;                          /* factor for decreasing trust radius */
    /* */
    double avmax;                                /* max allowed |a|/|v| */
    double h_df;                                /* step size for finite difference */
    /* Jacobian */
    double h_fvv;                                /* step size for finite difference */
    /* fvv */
} gsl_multifit_nlinear_parameters;
```

For the `gsl_multilarge_nlinear` interface, the user may modify the `gsl_multilarge_nlinear_parameters` structure, which is defined as follows:

type `gsl_multilarge_nlinear_parameters`

```

typedef struct
{
    const gsl_multilarge_nlinear_trs *trs;          /* trust region subproblem method */
    const gsl_multilarge_nlinear_scale *scale;     /* scaling method */
    const gsl_multilarge_nlinear_solver *solver;   /* solver method */
    gsl_multilarge_nlinear_fdtype fdtype;         /* finite difference method */
    double factor_up;                             /* factor for increasing trust_
↪radius */
    double factor_down;                           /* factor for decreasing trust_
↪radius */
    double avmax;                                 /* max allowed |a|/|v| */
    double h_df;                                  /* step size for finite difference_
↪Jacobian */
    double h_fvv;                                 /* step size for finite difference_
↪fvv */
    size_t max_iter;                              /* maximum iterations for trs method_
↪*/
    double tol;                                   /* tolerance for solving trs */
} gsl_multilarge_nlinear_parameters;

```

Each of these parameters is discussed in further detail below.

type **gsl\_multifit\_nlinear\_trs**

type **gsl\_multilarge\_nlinear\_trs**

The parameter `trs` determines the method used to solve the trust region subproblem, and may be selected from the following choices,

*gsl\_multifit\_nlinear\_trs* \***gsl\_multifit\_nlinear\_trs\_lm**

*gsl\_multilarge\_nlinear\_trs* \***gsl\_multilarge\_nlinear\_trs\_lm**

This selects the Levenberg-Marquardt algorithm.

*gsl\_multifit\_nlinear\_trs* \***gsl\_multifit\_nlinear\_trs\_lmaccel**

*gsl\_multilarge\_nlinear\_trs* \***gsl\_multilarge\_nlinear\_trs\_lmaccel**

This selects the Levenberg-Marquardt algorithm with geodesic acceleration.

*gsl\_multifit\_nlinear\_trs* \***gsl\_multifit\_nlinear\_trs\_dogleg**

*gsl\_multilarge\_nlinear\_trs* \***gsl\_multilarge\_nlinear\_trs\_dogleg**

This selects the dogleg algorithm.

*gsl\_multifit\_nlinear\_trs* \***gsl\_multifit\_nlinear\_trs\_ddogleg**

*gsl\_multilarge\_nlinear\_trs* \***gsl\_multilarge\_nlinear\_trs\_ddogleg**

This selects the double dogleg algorithm.

*gsl\_multifit\_nlinear\_trs* \***gsl\_multifit\_nlinear\_trs\_subspace2D**

*gsl\_multilarge\_nlinear\_trs* \***gsl\_multilarge\_nlinear\_trs\_subspace2D**

This selects the 2D subspace algorithm.

*gsl\_multilarge\_nlinear\_trs* \***gsl\_multilarge\_nlinear\_trs\_cgst**

This selects the Steihaug-Toint conjugate gradient algorithm. This method is available only for large systems.

type **gsl\_multifit\_nlinear\_scale**

type **gsl\_multilarge\_nlinear\_scale**

The parameter `scale` determines the diagonal scaling matrix  $D$  and may be selected from the following choices,

*gsl\_multifit\_nlinear\_scale* \***gsl\_multifit\_nlinear\_scale\_more**

*gsl\_multilarge\_nlinear\_scale* \***gsl\_multilarge\_nlinear\_scale\_more**

This damping strategy was suggested by Moré, and corresponds to  $D^T D = \max(\text{diag}(J^T J))$ , in other words the maximum elements of  $\text{diag}(J^T J)$  encountered thus far in the iteration. This choice of  $D$  makes the problem scale-invariant, so that if the model parameters  $x_i$  are each scaled by an arbitrary constant,  $\tilde{x}_i = a_i x_i$ , then the sequence of iterates produced by the algorithm would be unchanged. This method can work very well in cases where the model parameters have widely different scales (ie: if some parameters are measured in nanometers, while others are measured in degrees Kelvin). This strategy has been proven effective on a large class of problems and so it is the library default, but it may not be the best choice for all problems.

*gsl\_multifit\_nlinear\_scale* \***gsl\_multifit\_nlinear\_scale\_levenberg**

*gsl\_multilarge\_nlinear\_scale* \***gsl\_multilarge\_nlinear\_scale\_levenberg**

This damping strategy was originally suggested by Levenberg, and corresponds to  $D^T D = I$ . This method has also proven effective on a large class of problems, but is not scale-invariant. However, some authors (e.g. Transtrum and Sethna 2012) argue that this choice is better for problems which are susceptible to parameter evaporation (ie: parameters go to infinity)

*gsl\_multifit\_nlinear\_scale* \***gsl\_multifit\_nlinear\_scale\_marquardt**

*gsl\_multilarge\_nlinear\_scale* \***gsl\_multilarge\_nlinear\_scale\_marquardt**

This damping strategy was suggested by Marquardt, and corresponds to  $D^T D = \text{diag}(J^T J)$ . This method is scale-invariant, but it is generally considered inferior to both the Levenberg and Moré strategies, though may work well on certain classes of problems.

type **gsl\_multifit\_nlinear\_solver**

type **gsl\_multilarge\_nlinear\_solver**

Solving the trust region subproblem on each iteration almost always requires the solution of the following linear least squares system

$$\begin{bmatrix} J \\ \sqrt{\mu}D \end{bmatrix} \delta = - \begin{bmatrix} f \\ 0 \end{bmatrix}$$

The solver parameter determines how the system is solved and can be selected from the following choices:

*gsl\_multifit\_nlinear\_solver* \***gsl\_multifit\_nlinear\_solver\_qr**

This method solves the system using a rank revealing QR decomposition of the Jacobian  $J$ . This method will produce reliable solutions in cases where the Jacobian is rank deficient or near-singular but does require about twice as many operations as the Cholesky method discussed below.

*gsl\_multifit\_nlinear\_solver* \***gsl\_multifit\_nlinear\_solver\_cholesky**

*gsl\_multilarge\_nlinear\_solver* \***gsl\_multilarge\_nlinear\_solver\_cholesky**

This method solves the alternate normal equations problem

$$(J^T J + \mu D^T D) \delta = -J^T f$$

by using a Cholesky decomposition of the matrix  $J^T J + \mu D^T D$ . This method is faster than the QR approach, however it is susceptible to numerical instabilities if the Jacobian matrix is rank deficient or near-singular. In these cases, an attempt is made to reduce the condition number of the matrix using Jacobi preconditioning, but for highly ill-conditioned problems the QR approach is better. If it is known that the Jacobian matrix is well conditioned, this method is accurate and will perform faster than the QR approach.

*gsl\_multifit\_nlinear\_solver* \***gsl\_multifit\_nlinear\_solver\_mcholesky**

*gsl\_multilarge\_nlinear\_solver* \***gsl\_multilarge\_nlinear\_solver\_mcholesky**

This method solves the alternate normal equations problem

$$(J^T J + \mu D^T D) \delta = -J^T f$$

by using a modified Cholesky decomposition of the matrix  $J^T J + \mu D^T D$ . This is more suitable for the dogleg methods where the parameter  $\mu = 0$ , and the matrix  $J^T J$  may be ill-conditioned or indefinite causing the standard Cholesky decomposition to fail. This method is based on Level 2 BLAS and is thus slower than the standard Cholesky decomposition, which is based on Level 3 BLAS.

*gsl\_multifit\_nlinear\_solver* \***gsl\_multifit\_nlinear\_solver\_svd**

This method solves the system using a singular value decomposition of the Jacobian  $J$ . This method will produce the most reliable solutions for ill-conditioned Jacobians but is also the slowest solver method.

type **gsl\_multifit\_nlinear\_fdtype**

The parameter `fdtype` specifies whether to use forward or centered differences when approximating the Jacobian. This is only used when an analytic Jacobian is not provided to the solver. This parameter may be set to one of the following choices.

**GSL\_MULTIFIT\_NLINEAR\_FWDIFF**

This specifies a forward finite difference to approximate the Jacobian matrix. The Jacobian matrix will be calculated as

$$J_{ij} = \frac{1}{\Delta_j} (f_i(x + \Delta_j e_j) - f_i(x))$$

where  $\Delta_j = h|x_j|$  and  $e_j$  is the standard  $j$ -th Cartesian unit basis vector so that  $x + \Delta_j e_j$  represents a small (forward) perturbation of the  $j$ -th parameter by an amount  $\Delta_j$ . The perturbation  $\Delta_j$  is proportional to the current value  $|x_j|$  which helps to calculate an accurate Jacobian when the various parameters have different scale sizes. The value of  $h$  is specified by the `h_df` parameter. The accuracy of this method is  $O(h)$ , and evaluating this matrix requires an additional  $p$  function evaluations.

**GSL\_MULTIFIT\_NLINEAR\_CTRDIFF**

This specifies a centered finite difference to approximate the Jacobian matrix. The Jacobian matrix will be calculated as

$$J_{ij} = \frac{1}{\Delta_j} \left( f_i(x + \frac{1}{2}\Delta_j e_j) - f_i(x - \frac{1}{2}\Delta_j e_j) \right)$$

See above for a description of  $\Delta_j$ . The accuracy of this method is  $O(h^2)$ , but evaluating this matrix requires an additional  $2p$  function evaluations.

double `factor_up`

When a step is accepted, the trust region radius will be increased by this factor. The default value is 3.

double `factor_down`

When a step is rejected, the trust region radius will be decreased by this factor. The default value is 2.

double `avmax`

When using geodesic acceleration to solve a nonlinear least squares problem, an important parameter to monitor is the ratio of the acceleration term to the velocity term,

$$\frac{\|a\|}{\|v\|}$$

If this ratio is small, it means the acceleration correction is contributing very little to the step. This could be because the problem is not “nonlinear” enough to benefit from the acceleration. If the ratio is large ( $> 1$ ) it means that the acceleration is larger than the velocity, which shouldn’t happen since the step represents a truncated series and so the second order term  $a$  should be smaller than the first order term  $v$  to guarantee convergence. Therefore any steps with a ratio larger than the parameter `avmax` are rejected. `avmax` is set to 0.75 by default. For problems which experience difficulty converging, this threshold could be lowered.

double `h_df`

This parameter specifies the step size for approximating the Jacobian matrix with finite differences. It is set to  $\sqrt{\epsilon}$  by default, where  $\epsilon$  is `GSL_DBL_EPSILON`.

```
double h_fvv
```

When using geodesic acceleration, the user must either supply a function to calculate  $f_{vv}(x)$  or the library can estimate this second directional derivative using a finite difference method. When using finite differences, the library must calculate  $f(x + hv)$  where  $h$  represents a small step in the velocity direction. The parameter `h_fvv` defines this step size and is set to 0.02 by default.

## 41.5 Initializing the Solver

```
type gsl_multifit_nlinear_type
```

This structure specifies the type of algorithm which will be used to solve a nonlinear least squares problem. It may be selected from the following choices,

```
gsl_multifit_nlinear_type *gsl_multifit_nlinear_trust
```

This specifies a trust region method. It is currently the only implemented nonlinear least squares method.

```
gsl_multifit_nlinear_workspace *gsl_multifit_nlinear_alloc(const gsl_multifit_nlinear_type *T, const
                                                         gsl_multifit_nlinear_parameters *params, const
                                                         size_t n, const size_t p)
```

```
gsl_multilarge_nlinear_workspace *gsl_multilarge_nlinear_alloc(const gsl_multilarge_nlinear_type *T,
                                                                const gsl_multilarge_nlinear_parameters
                                                                *params, const size_t n, const size_t p)
```

These functions return a pointer to a newly allocated instance of a derivative solver of type  $T$  for  $n$  observations and  $p$  parameters. The *params* input specifies a tunable set of parameters which will affect important details in each iteration of the trust region subproblem algorithm. It is recommended to start with the suggested default parameters (see `gsl_multifit_nlinear_default_parameters()` and `gsl_multilarge_nlinear_default_parameters()`) and then tune the parameters once the code is working correctly. See *Tunable Parameters*. for descriptions of the various parameters. For example, the following code creates an instance of a Levenberg-Marquardt solver for 100 data points and 3 parameters, using suggested defaults:

```
const gsl_multifit_nlinear_type * T = gsl_multifit_nlinear_trust;
gsl_multifit_nlinear_parameters params = gsl_multifit_nlinear_default_parameters();
gsl_multifit_nlinear_workspace * w = gsl_multifit_nlinear_alloc (T, &params, 100,
↪3);
```

The number of observations  $n$  must be greater than or equal to parameters  $p$ .

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

```
gsl_multifit_nlinear_parameters gsl_multifit_nlinear_default_parameters(void)
```

```
gsl_multilarge_nlinear_parameters gsl_multilarge_nlinear_default_parameters(void)
```

These functions return a set of recommended default parameters for use in solving nonlinear least squares problems. The user can tune each parameter to improve the performance on their particular problem, see *Tunable Parameters*.

```
int gsl_multifit_nlinear_init(const gsl_vector *x, gsl_multifit_nlinear_fdf *fdf,
                             gsl_multifit_nlinear_workspace *w)
```



```
int gsl_multifit_nlinear_winit(const gsl_vector *x, const gsl_vector *wts, gsl_multifit_nlinear_fdf *fdf,
                               gsl_multifit_nlinear_workspace *w)
```

```
int gsl_multilarge_nlinear_init(const gsl_vector *x, gsl_multilarge_nlinear_fdf *fdf,
                                gsl_multilarge_nlinear_workspace *w)
```

These functions initialize, or reinitialize, an existing workspace  $w$  to use the system  $fdf$  and the initial guess  $x$ . See *Providing the Function to be Minimized* for a description of the  $fdf$  structure.

Optionally, a weight vector  $wts$  can be given to perform a weighted nonlinear regression. Here, the weighting matrix is  $W = \text{diag}(w_1, w_2, \dots, w_n)$ .

```
void gsl_multifit_nlinear_free(gsl_multifit_nlinear_workspace *w)
```

```
void gsl_multilarge_nlinear_free(gsl_multilarge_nlinear_workspace *w)
```

These functions free all the memory associated with the workspace  $w$ .

```
const char *gsl_multifit_nlinear_name(const gsl_multifit_nlinear_workspace *w)
```

```
const char *gsl_multilarge_nlinear_name(const gsl_multilarge_nlinear_workspace *w)
```

These functions return a pointer to the name of the solver. For example:

```
printf ("w is a '%s' solver\n", gsl_multifit_nlinear_name (w));
```

would print something like  $w$  is a 'trust-region' solver.

```
const char *gsl_multifit_nlinear_trs_name(const gsl_multifit_nlinear_workspace *w)
```

```
const char *gsl_multilarge_nlinear_trs_name(const gsl_multilarge_nlinear_workspace *w)
```

These functions return a pointer to the name of the trust region subproblem method. For example:

```
printf ("w is a '%s' solver\n", gsl_multifit_nlinear_trs_name (w));
```

would print something like  $w$  is a 'levenberg-marquardt' solver.

## 41.6 Providing the Function to be Minimized

The user must provide  $n$  functions of  $p$  variables for the minimization algorithm to operate on. In order to allow for arbitrary parameters the functions are defined by the following data types:

```
type gsl_multifit_nlinear_fdf
```

This data type defines a general system of functions with arbitrary parameters, the corresponding Jacobian matrix of derivatives, and optionally the second directional derivative of the functions for geodesic acceleration.

```
int (* f) (const gsl_vector * x, void * params, gsl_vector * f)
```

This function should store the  $n$  components of the vector  $f(x)$  in  $f$  for argument  $x$  and arbitrary parameters  $params$ , returning an appropriate error code if the function cannot be computed.

```
int (* df) (const gsl_vector * x, void * params, gsl_matrix * J)
```

This function should store the  $n$ -by- $p$  matrix result

$$J_{ij} = \partial f_i(x) / \partial x_j$$

in  $J$  for argument  $x$  and arbitrary parameters  $params$ , returning an appropriate error code if the matrix cannot be computed. If an analytic Jacobian is unavailable, or too expensive to compute, this



function pointer may be set to `NULL`, in which case the Jacobian will be internally computed using finite difference approximations of the function `f`.

```
int (* fvv) (const gsl_vector * x, const gsl_vector * v, void * params, gsl_vector * fvv)
```

When geodesic acceleration is enabled, this function should store the  $n$  components of the vector  $f_{vv}(x) = \sum_{\alpha\beta} v_\alpha v_\beta \frac{\partial}{\partial x_\alpha} \frac{\partial}{\partial x_\beta} f(x)$ , representing second directional derivatives of the function to be minimized, into the output `fvv`. The parameter vector is provided in `x` and the velocity vector is provided in `v`, both of which have  $p$  components. The arbitrary parameters are given in `params`. If analytic expressions for  $f_{vv}(x)$  are unavailable or too difficult to compute, this function pointer may be set to `NULL`, in which case  $f_{vv}(x)$  will be computed internally using a finite difference approximation.

```
size_t n
```

the number of functions, i.e. the number of components of the vector `f`.

```
size_t p
```

the number of independent variables, i.e. the number of components of the vector `x`.

```
void * params
```

a pointer to the arbitrary parameters of the function.

```
size_t nevalf
```

This does not need to be set by the user. It counts the number of function evaluations and is initialized by the `_init` function.

```
size_t nevaldf
```

This does not need to be set by the user. It counts the number of Jacobian evaluations and is initialized by the `_init` function.

```
size_t nevalfvv
```

This does not need to be set by the user. It counts the number of  $f_{vv}(x)$  evaluations and is initialized by the `_init` function.

```
type gsl_multilarge_nlinear_fdf
```

This data type defines a general system of functions with arbitrary parameters, a function to compute  $Ju$  or  $J^T u$  for a given vector  $u$ , the normal equations matrix  $J^T J$ , and optionally the second directional derivative of the functions for geodesic acceleration.

```
int (* f) (const gsl_vector * x, void * params, gsl_vector * f)
```

This function should store the  $n$  components of the vector  $f(x)$  in `f` for argument `x` and arbitrary parameters `params`, returning an appropriate error code if the function cannot be computed.

```
int (* df) (CBLAS_TRANSPOSE_t TransJ, const gsl_vector * x, const gsl_vector * u, void * params, gsl_vector * v, gsl_matrix * JTJ)
```

If `TransJ` is equal to `CblasNoTrans`, then this function should compute the matrix-vector product  $Ju$  and store the result in `v`. If `TransJ` is equal to `CblasTrans`, then this function should compute the matrix-vector product  $J^T u$  and store the result in `v`. Additionally, the normal equations matrix  $J^T J$  should be stored in the lower half of `JTJ`. The input matrix `JTJ` could be set to `NULL`, for example by iterative methods which do not require this matrix, so the user should check for this prior to constructing the matrix. The input `params` contains the arbitrary parameters.

```
int (* fvv) (const gsl_vector * x, const gsl_vector * v, void * params, gsl_vector * fvv)
```

When geodesic acceleration is enabled, this function should store the  $n$  components of the vector  $f_{vv}(x) = \sum_{\alpha\beta} v_\alpha v_\beta \frac{\partial}{\partial x_\alpha} \frac{\partial}{\partial x_\beta} f(x)$ , representing second directional derivatives of the function to be minimized, into the output `fvv`. The parameter vector is provided in `x` and the velocity vector is provided in `v`, both of which have  $p$  components. The arbitrary parameters are given in `params`. If analytic expressions for  $f_{vv}(x)$  are unavailable or too difficult to compute, this function pointer may be set to `NULL`, in which case  $f_{vv}(x)$  will be computed internally using a finite difference approximation.

`size_t n`

the number of functions, i.e. the number of components of the vector `f`.

`size_t p`

the number of independent variables, i.e. the number of components of the vector `x`.

`void * params`

a pointer to the arbitrary parameters of the function.

`size_t nevalf`

This does not need to be set by the user. It counts the number of function evaluations and is initialized by the `_init` function.

`size_t nevaldfu`

This does not need to be set by the user. It counts the number of Jacobian matrix-vector evaluations ( $Ju$  or  $J^T u$ ) and is initialized by the `_init` function.

`size_t nevaldf2`

This does not need to be set by the user. It counts the number of  $J^T J$  evaluations and is initialized by the `_init` function.

`size_t nevalfvv`

This does not need to be set by the user. It counts the number of  $f_{vv}(x)$  evaluations and is initialized by the `_init` function.

Note that when fitting a non-linear model against experimental data, the data is passed to the functions above using the `params` argument and the trial best-fit parameters through the `x` argument.

## 41.7 Iteration

The following functions drive the iteration of each algorithm. Each function performs one iteration of the trust region method and updates the state of the solver.

```
int gsl_multifit_nlinear_iterate(gsl_multifit_nlinear_workspace *w)
```

```
int gsl_multilarge_nlinear_iterate(gsl_multilarge_nlinear_workspace *w)
```

These functions perform a single iteration of the solver `w`. If the iteration encounters an unexpected problem then an error code will be returned. The solver workspace maintains a current estimate of the best-fit parameters at all times.

The solver workspace `w` contains the following entries, which can be used to track the progress of the solution:

```
gsl_vector * x
```

The current position, length  $p$ .

```
gsl_vector * f
```

The function residual vector at the current position  $f(x)$ , length  $n$ .

`gsl_matrix * J`

The Jacobian matrix at the current position  $J(x)$ , size  $n$ -by- $p$  (only for `gsl_multifit_nlinear` interface).

`gsl_vector * dx`

The difference between the current position and the previous position, i.e. the last step  $\delta$ , taken as a vector, length  $p$ .

These quantities can be accessed with the following functions,

`gsl_vector *gsl_multifit_nlinear_position(const gsl_multifit_nlinear_workspace *w)`

`gsl_vector *gsl_multilarge_nlinear_position(const gsl_multilarge_nlinear_workspace *w)`

These functions return the current position  $x$  (i.e. best-fit parameters) of the solver  $w$ .

`gsl_vector *gsl_multifit_nlinear_residual(const gsl_multifit_nlinear_workspace *w)`

`gsl_vector *gsl_multilarge_nlinear_residual(const gsl_multilarge_nlinear_workspace *w)`

These functions return the current residual vector  $f(x)$  of the solver  $w$ . For weighted systems, the residual vector includes the weighting factor  $\sqrt{W}$ .

`gsl_matrix *gsl_multifit_nlinear_jac(const gsl_multifit_nlinear_workspace *w)`

This function returns a pointer to the  $n$ -by- $p$  Jacobian matrix for the current iteration of the solver  $w$ . This function is available only for the `gsl_multifit_nlinear` interface.

`size_t gsl_multifit_nlinear_niter(const gsl_multifit_nlinear_workspace *w)`

`size_t gsl_multilarge_nlinear_niter(const gsl_multilarge_nlinear_workspace *w)`

These functions return the number of iterations performed so far. The iteration counter is updated on each call to the `_iterate` functions above, and reset to 0 in the `_init` functions.

`int gsl_multifit_nlinear_rcond(double *rcond, const gsl_multifit_nlinear_workspace *w)`

`int gsl_multilarge_nlinear_rcond(double *rcond, const gsl_multilarge_nlinear_workspace *w)`

This function estimates the reciprocal condition number of the Jacobian matrix at the current position  $x$  and stores it in `rcond`. The computed value is only an estimate to give the user a guideline as to the conditioning of their particular problem. Its calculation is based on which factorization method is used (Cholesky, QR, or SVD).

- For the Cholesky solver, the matrix  $J^T J$  is factored at each iteration. Therefore this function will estimate the 1-norm condition number  $rcond^2 = 1/(\|J^T J\|_1 \cdot \|(J^T J)^{-1}\|_1)$
- For the QR solver,  $J$  is factored as  $J = QR$  at each iteration. For simplicity, this function calculates the 1-norm conditioning of only the  $R$  factor,  $rcond = 1/(\|R\|_1 \cdot \|R^{-1}\|_1)$ . This can be computed efficiently since  $R$  is upper triangular.
- For the SVD solver, in order to efficiently solve the trust region subproblem, the matrix which is factored is  $JD^{-1}$ , instead of  $J$  itself. The resulting singular values are used to provide the 2-norm reciprocal condition number, as  $rcond = \sigma_{min}/\sigma_{max}$ . Note that when using Moré scaling,  $D \neq I$  and the resulting `rcond` estimate may be significantly different from the true `rcond` of  $J$  itself.

`double gsl_multifit_nlinear_avratio(const gsl_multifit_nlinear_workspace *w)`

`double gsl_multilarge_nlinear_avratio(const gsl_multilarge_nlinear_workspace *w)`

This function returns the current ratio  $|a|/|v|$  of the acceleration correction term to the velocity step term. The acceleration term is computed only by the `gsl_multifit_nlinear_trs_lmaccel` and `gsl_multilarge_nlinear_trs_lmaccel` methods, so this ratio will be zero for other TRS methods.

## 41.8 Testing for Convergence

A minimization procedure should stop when one of the following conditions is true:

- A minimum has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the current estimate of the best-fit parameters in several standard ways.

```
int gsl_multifit_nlinear_test(const double xtol, const double gtol, const double ftol, int *info, const
                             gsl_multifit_nlinear_workspace *w)
```

```
int gsl_multilarge_nlinear_test(const double xtol, const double gtol, const double ftol, int *info, const
                                gsl_multilarge_nlinear_workspace *w)
```

These functions test for convergence of the minimization method using the following criteria:

- Testing for a small step size relative to the current parameter vector

$$|\delta_i| \leq xtol(|x_i| + xtol)$$

for each  $0 \leq i < p$ . Each element of the step vector  $\delta$  is tested individually in case the different parameters have widely different scales. Adding  $xtol$  to  $|x_i|$  helps the test avoid breaking down in situations where the true solution value  $x_i = 0$ . If this test succeeds, *info* is set to 1 and the function returns `GSL_SUCCESS`.

A general guideline for selecting the step tolerance is to choose  $xtol = 10^{-d}$  where  $d$  is the number of accurate decimal digits desired in the solution  $x$ . See Dennis and Schnabel for more information.

- Testing for a small gradient ( $g = \nabla\Phi(x) = J^T f$ ) indicating a local function minimum:

$$\max_i |g_i \times \max(x_i, 1)| \leq gtol \times \max(\Phi(x), 1)$$

This expression tests whether the ratio  $(\nabla\Phi)_i x_i / \Phi$  is small. Testing this scaled gradient is a better than  $\nabla\Phi$  alone since it is a dimensionless quantity and so independent of the scale of the problem. The `max` arguments help ensure the test doesn't break down in regions where  $x_i$  or  $\Phi(x)$  are close to 0. If this test succeeds, *info* is set to 2 and the function returns `GSL_SUCCESS`.

A general guideline for choosing the gradient tolerance is to set `gtol = GSL_DBL_EPSILON^(1/3)`. See Dennis and Schnabel for more information.

If none of the tests succeed, *info* is set to 0 and the function returns `GSL_CONTINUE`, indicating further iterations are required.

## 41.9 High Level Driver

These routines provide a high level wrapper that combines the iteration and convergence testing for easy use.

```
int gsl_multifit_nlinear_driver(const size_t maxiter, const double xtol, const double gtol, const double ftol,
                               void (*callback)(const size_t iter, void *params, const
                                                  gsl_multifit_linear_workspace *w), void *callback_params, int *info,
                               gsl_multifit_nlinear_workspace *w)
```

```
int gsl_multilarge_nlinear_driver(const size_t maxiter, const double xtol, const double gtol, const double
                                ftol, void (*callback)(const size_t iter, void *params, const
                                gsl_multilarge_linear_workspace *w), void *callback_params, int *info,
                                gsl_multilarge_nlinear_workspace *w)
```

These functions iterate the nonlinear least squares solver *w* for a maximum of *maxiter* iterations. After each iteration, the system is tested for convergence with the error tolerances *xtol*, *gtol* and *ftol*. Additionally, the user may supply a callback function *callback* which is called after each iteration, so that the user may save or print relevant quantities for each iteration. The parameter *callback\_params* is passed to the *callback* function. The parameters *callback* and *callback\_params* may be set to NULL to disable this feature. Upon successful convergence, the function returns GSL\_SUCCESS and sets *info* to the reason for convergence (see *gsl\_multifit\_nlinear\_test()*). If the function has not converged after *maxiter* iterations, GSL\_EMAXITER is returned. In rare cases, during an iteration the algorithm may be unable to find a new acceptable step  $\delta$  to take. In this case, GSL\_ENOPROG is returned indicating no further progress can be made. If your problem is having difficulty converging, see *Troubleshooting* for further guidance.

## 41.10 Covariance matrix of best fit parameters

```
int gsl_multifit_nlinear_covar(const gsl_matrix *J, const double epsrel, gsl_matrix *covar)
```

```
int gsl_multilarge_nlinear_covar(gsl_matrix *covar, gsl_multilarge_nlinear_workspace *w)
```

This function computes the covariance matrix of best-fit parameters using the Jacobian matrix *J* and stores it in *covar*. The parameter *epsrel* is used to remove linear-dependent columns when *J* is rank deficient.

The covariance matrix is given by,

$$C = (J^T J)^{-1}$$

or in the weighted case,

$$C = (J^T W J)^{-1}$$

and is computed using the factored form of the Jacobian (Cholesky, QR, or SVD). Any columns of *R* which satisfy

$$|R_{kk}| \leq \text{epsrel}|R_{11}|$$

are considered linearly-dependent and are excluded from the covariance matrix (the corresponding rows and columns of the covariance matrix are set to zero).

If the minimisation uses the weighted least-squares function  $f_i = (Y(x, t_i) - y_i)/\sigma_i$  then the covariance matrix above gives the statistical error on the best-fit parameters resulting from the Gaussian errors  $\sigma_i$  on the underlying data  $y_i$ . This can be verified from the relation  $\delta f = J\delta c$  and the fact that the fluctuations in *f* from the data  $y_i$  are normalised by  $\sigma_i$  and so satisfy

$$\langle \delta f \delta f^T \rangle = I$$

For an unweighted least-squares function  $f_i = (Y(x, t_i) - y_i)$  the covariance matrix above should be multiplied by the variance of the residuals about the best-fit  $\sigma^2 = \sum (y_i - Y(x, t_i))^2 / (n - p)$  to give the variance-covariance matrix  $\sigma^2 C$ . This estimates the statistical error on the best-fit parameters from the scatter of the underlying data.

For more information about covariance matrices see *Linear Least-Squares Overview*.

## 41.11 Troubleshooting

When developing a code to solve a nonlinear least squares problem, here are a few considerations to keep in mind.

1. The most common difficulty is the accurate implementation of the Jacobian matrix. If the analytic Jacobian is not properly provided to the solver, this can hinder and many times prevent convergence of the method. When developing a new nonlinear least squares code, it often helps to compare the program output with the internally computed finite difference Jacobian and the user supplied analytic Jacobian. If there is a large difference in coefficients, it is likely the analytic Jacobian is incorrectly implemented.
2. If your code is having difficulty converging, the next thing to check is the starting point provided to the solver. The methods of this chapter are local methods, meaning if you provide a starting point far away from the true minimum, the method may converge to a local minimum or not converge at all. Sometimes it is possible to solve a linearized approximation to the nonlinear problem, and use the linear solution as the starting point to the nonlinear problem.
3. If the various parameters of the coefficient vector  $x$  vary widely in magnitude, then the problem is said to be badly scaled. The methods of this chapter do attempt to automatically rescale the elements of  $x$  to have roughly the same order of magnitude, but in extreme cases this could still cause problems for convergence. In these cases it is recommended for the user to scale their parameter vector  $x$  so that each parameter spans roughly the same range, say  $[-1, 1]$ . The solution vector can be backscaled to recover the original units of the problem.

## 41.12 Examples

The following example programs demonstrate the nonlinear least squares fitting capabilities.

### 41.12.1 Exponential Fitting Example

The following example program fits a weighted exponential model with background to experimental data,  $Y = A \exp(-\lambda t) + b$ . The first part of the program sets up the functions `expb_f()` and `expb_df()` to calculate the model and its Jacobian. The appropriate fitting function is given by,

$$f_i = (A \exp(-\lambda t_i) + b) - y_i$$

where we have chosen  $t_i = iT/(N-1)$ , where  $N$  is the number of data points fitted, so that  $t_i \in [0, T]$ . The Jacobian matrix  $J$  is the derivative of these functions with respect to the three parameters  $(A, \lambda, b)$ . It is given by,

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

where  $x_0 = A$ ,  $x_1 = \lambda$  and  $x_2 = b$ . The  $i$ -th row of the Jacobian is therefore

$$J_{i.} = \left( \exp(-\lambda t_i) \quad -t_i A \exp(-\lambda t_i) \quad 1 \right)$$

The main part of the program sets up a Levenberg-Marquardt solver and some simulated random data. The data uses the known parameters (5.0,1.5,1.0) combined with Gaussian noise (standard deviation = 0.1) with a maximum time  $T = 3$  and  $N = 100$  timesteps. The initial guess for the parameters is chosen as (1.0, 1.0, 0.0). The iteration terminates when the relative change in  $x$  is smaller than  $10^{-8}$ , or when the magnitude of the gradient falls below  $10^{-8}$ . Here are the results of running the program:

```
iter 0: A = 1.0000, lambda = 1.0000, b = 0.0000, cond(J) =      inf, |f(x)| = 88.4448
iter 1: A = 4.5109, lambda = 2.5258, b = 1.0704, cond(J) = 26.2686, |f(x)| = 24.0646
iter 2: A = 4.8565, lambda = 1.7442, b = 1.1669, cond(J) = 23.7470, |f(x)| = 11.9797
```

(continues on next page)

(continued from previous page)

```

iter 3: A = 4.9356, lambda = 1.5713, b = 1.0767, cond(J) = 17.5849, |f(x)| = 10.7355
iter 4: A = 4.8678, lambda = 1.4838, b = 1.0252, cond(J) = 16.3428, |f(x)| = 10.5000
iter 5: A = 4.8118, lambda = 1.4481, b = 1.0076, cond(J) = 15.7925, |f(x)| = 10.4786
iter 6: A = 4.7983, lambda = 1.4404, b = 1.0041, cond(J) = 15.5840, |f(x)| = 10.4778
iter 7: A = 4.7967, lambda = 1.4395, b = 1.0037, cond(J) = 15.5396, |f(x)| = 10.4778
iter 8: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) = 15.5344, |f(x)| = 10.4778
iter 9: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) = 15.5339, |f(x)| = 10.4778
iter 10: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) = 15.5339, |f(x)| = 10.4778
iter 11: A = 4.7965, lambda = 1.4394, b = 1.0037, cond(J) = 15.5339, |f(x)| = 10.4778
summary from method 'trust-region/levenberg-marquardt'
number of iterations: 11
function evaluations: 16
Jacobian evaluations: 12
reason for stopping: small gradient
initial |f(x)| = 88.444756
final   |f(x)| = 10.477801
chisq/dof = 1.1318
A      = 4.79653 +/- 0.18704
lambda = 1.43937 +/- 0.07390
b      = 1.00368 +/- 0.03473
status = success

```

The approximate values of the parameters are found correctly, and the chi-squared value indicates a good fit (the chi-squared per degree of freedom is approximately 1). In this case the errors on the parameters can be estimated from the square roots of the diagonal elements of the covariance matrix. If the chi-squared value shows a poor fit (i.e.  $\chi^2/(n-p) \gg 1$ ) then the error estimates obtained from the covariance matrix will be too small. In the example program the error estimates are multiplied by  $\sqrt{\chi^2/(n-p)}$  in this case, a common way of increasing the errors for a poor fit. Note that a poor fit will result from the use of an inappropriate model, and the scaled error estimates may then be outside the range of validity for Gaussian errors.

Additionally, we see that the condition number of  $J(x)$  stays reasonably small throughout the iteration. This indicates we could safely switch to the Cholesky solver for speed improvement, although this particular system is too small to really benefit.

Fig. 41.1 shows the fitted curve with the original data.

```

#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>

#define N      100    /* number of data points to fit */
#define TMAX   (3.0) /* time variable in [0,TMAX] */

struct data {
    size_t n;
    double * t;
    double * y;
};

```

(continues on next page)

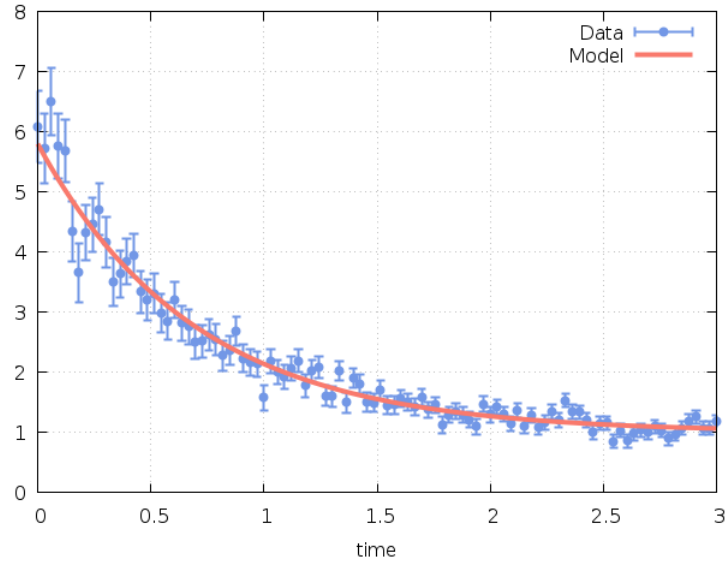


Fig. 41.1: Exponential fitted curve with data

(continued from previous page)

```

int
expb_f (const gsl_vector * x, void *data,
        gsl_vector * f)
{
    size_t n = ((struct data *)data)->n;
    double *t = ((struct data *)data)->t;
    double *y = ((struct data *)data)->y;

    double A = gsl_vector_get (x, 0);
    double lambda = gsl_vector_get (x, 1);
    double b = gsl_vector_get (x, 2);

    size_t i;

    for (i = 0; i < n; i++)
    {
        /* Model  $Y_i = A * \exp(-\lambda * t_i) + b$  */
        double Yi = A * exp (-lambda * t[i]) + b;
        gsl_vector_set (f, i, Yi - y[i]);
    }

    return GSL_SUCCESS;
}

int
expb_df (const gsl_vector * x, void *data,
         gsl_matrix * J)
{
    size_t n = ((struct data *)data)->n;

```

(continues on next page)



(continued from previous page)

```

double *t = ((struct data *)data)->t;

double A = gsl_vector_get (x, 0);
double lambda = gsl_vector_get (x, 1);

size_t i;

for (i = 0; i < n; i++)
  {
    /* Jacobian matrix J(i,j) = dfi / dxj, */
    /* where fi = (Yi - yi)/sigma[i],      */
    /*      Yi = A * exp(-lambda * t_i) + b */
    /* and the xj are the parameters (A,lambda,b) */
    double e = exp(-lambda * t[i]);
    gsl_matrix_set (J, i, 0, e);
    gsl_matrix_set (J, i, 1, -t[i] * A * e);
    gsl_matrix_set (J, i, 2, 1.0);
  }

return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
         const gsl_multifit_nlinear_workspace *w)
{
  gsl_vector *f = gsl_multifit_nlinear_residual(w);
  gsl_vector *x = gsl_multifit_nlinear_position(w);
  double rcond;

  /* compute reciprocal condition number of J(x) */
  gsl_multifit_nlinear_rcond(&rcond, w);

  fprintf(stderr, "iter %2zu: A = %.4f, lambda = %.4f, b = %.4f, cond(J) = %8.4f, |f(x)|_∞
↳ = %.4f\n",
          iter,
          gsl_vector_get(x, 0),
          gsl_vector_get(x, 1),
          gsl_vector_get(x, 2),
          1.0 / rcond,
          gsl_blas_dnrm2(f));
}

int
main (void)
{
  const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
  gsl_multifit_nlinear_workspace *w;
  gsl_multifit_nlinear_fdf fdf;
  gsl_multifit_nlinear_parameters fdf_params =
    gsl_multifit_nlinear_default_parameters();
  const size_t n = N;

```

(continues on next page)

```
const size_t p = 3;

gsl_vector *f;
gsl_matrix *J;
gsl_matrix *covar = gsl_matrix_alloc (p, p);
double t[N], y[N], weights[N];
struct data d = { n, t, y };
double x_init[3] = { 1.0, 1.0, 0.0 }; /* starting values */
gsl_vector_view x = gsl_vector_view_array (x_init, p);
gsl_vector_view wts = gsl_vector_view_array(weights, n);
gsl_rng * r;
double chisq, chisq0;
int status, info;
size_t i;

const double xtol = 1e-8;
const double gtol = 1e-8;
const double ftol = 0.0;

gsl_rng_env_setup();
r = gsl_rng_alloc(gsl_rng_default);

/* define the function to be minimized */
fdf.f = expb_f;
fdf.df = expb_df; /* set to NULL for finite-difference Jacobian */
fdf.fvv = NULL; /* not using geodesic acceleration */
fdf.n = n;
fdf.p = p;
fdf.params = &d;

/* this is the data to be fitted */
for (i = 0; i < n; i++)
{
    double ti = i * TMAX / (n - 1.0);
    double yi = 1.0 + 5 * exp (-1.5 * ti);
    double si = 0.1 * yi;
    double dy = gsl_ran_gaussian(r, si);

    t[i] = ti;
    y[i] = yi + dy;
    weights[i] = 1.0 / (si * si);
    printf ("data: %g %g %g\n", ti, y[i], si);
};

/* allocate workspace with default parameters */
w = gsl_multifit_nlinear_alloc (T, &fdf_params, n, p);

/* initialize solver with starting point and weights */
gsl_multifit_nlinear_winit (&x.vector, &wts.vector, &fdf, w);

/* compute initial cost function */
f = gsl_multifit_nlinear_residual(w);
```

(continues on next page)

(continued from previous page)

```

gsl_blas_ddot(f, f, &chisq0);

/* solve the system with a maximum of 100 iterations */
status = gsl_multifit_nlinear_driver(100, xtol, gtol, ftol,
                                   callback, NULL, &info, w);

/* compute covariance of best fit parameters */
J = gsl_multifit_nlinear_jac(w);
gsl_multifit_nlinear_covar (J, 0.0, covar);

/* compute final cost */
gsl_blas_ddot(f, f, &chisq);

#define FIT(i) gsl_vector_get(w->x, i)
#define ERR(i) sqrt(gsl_matrix_get(covar,i,i))

fprintf(stderr, "summary from method '%s/%s'\n",
         gsl_multifit_nlinear_name(w),
         gsl_multifit_nlinear_trs_name(w));
fprintf(stderr, "number of iterations: %zu\n",
         gsl_multifit_nlinear_niter(w));
fprintf(stderr, "function evaluations: %zu\n", fdf.nevalf);
fprintf(stderr, "Jacobian evaluations: %zu\n", fdf.nevaldf);
fprintf(stderr, "reason for stopping: %s\n",
         (info == 1) ? "small step size" : "small gradient");
fprintf(stderr, "initial |f(x)| = %f\n", sqrt(chisq0));
fprintf(stderr, "final   |f(x)| = %f\n", sqrt(chisq));

{
  double dof = n - p;
  double c = GSL_MAX_DBL(1, sqrt(chisq / dof));

  fprintf(stderr, "chisq/dof = %g\n", chisq / dof);

  fprintf (stderr, "A      = %.5f +/- %.5f\n", FIT(0), c*ERR(0));
  fprintf (stderr, "lambda = %.5f +/- %.5f\n", FIT(1), c*ERR(1));
  fprintf (stderr, "b      = %.5f +/- %.5f\n", FIT(2), c*ERR(2));
}

fprintf (stderr, "status = %s\n", gsl_strerror (status));

gsl_multifit_nlinear_free (w);
gsl_matrix_free (covar);
gsl_rng_free (r);

return 0;
}

```

### 41.12.2 Geodesic Acceleration Example 1

The following example program minimizes a modified Rosenbrock function, which is characterized by a narrow canyon with steep walls. The starting point is selected high on the canyon wall, so the solver must first find the canyon bottom and then navigate to the minimum. The problem is solved both with and without using geodesic acceleration for comparison. The cost function is given by

$$\begin{aligned}\Phi(x) &= \frac{1}{2}(f_1^2 + f_2^2) \\ f_1 &= 100(x_2 - x_1^2) \\ f_2 &= 1 - x_1\end{aligned}$$

The Jacobian matrix is

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} -200x_1 & 100 \\ -1 & 0 \end{pmatrix}$$

In order to use geodesic acceleration, the user must provide the second directional derivative of each residual in the velocity direction,  $D_v^2 f_i = \sum_{\alpha\beta} v_\alpha v_\beta \partial_\alpha \partial_\beta f_i$ . The velocity vector  $v$  is provided by the solver. For this example, these derivatives are

$$f_{vv} = D_v^2 \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} -200v_1^2 \\ 0 \end{pmatrix}$$

The solution of this minimization problem is

$$\begin{aligned}x^* &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \Phi(x^*) &= 0\end{aligned}$$

The program output is shown below:

```
=== Solving system without acceleration ===
NITER      = 53
NFEV      = 56
NJEV      = 54
NAEV      = 0
initial cost = 2.250225000000e+04
final cost  = 6.674986031430e-18
final x     = (9.999999974165e-01, 9.999999948328e-01)
final cond(J) = 6.000096055094e+02
=== Solving system with acceleration ===
NITER      = 15
NFEV      = 17
NJEV      = 16
NAEV      = 16
initial cost = 2.250225000000e+04
final cost  = 7.518932873279e-19
final x     = (9.999999991329e-01, 9.999999982657e-01)
final cond(J) = 6.000097233278e+02
```

We can see that enabling geodesic acceleration requires less than a third of the number of Jacobian evaluations in order to locate the minimum. The path taken by both methods is shown in [Fig. 41.2](#). The contours show the cost function  $\Phi(x_1, x_2)$ . We see that both methods quickly find the canyon bottom, but the geodesic acceleration method navigates along the bottom to the solution with significantly fewer iterations.

The program is given below.

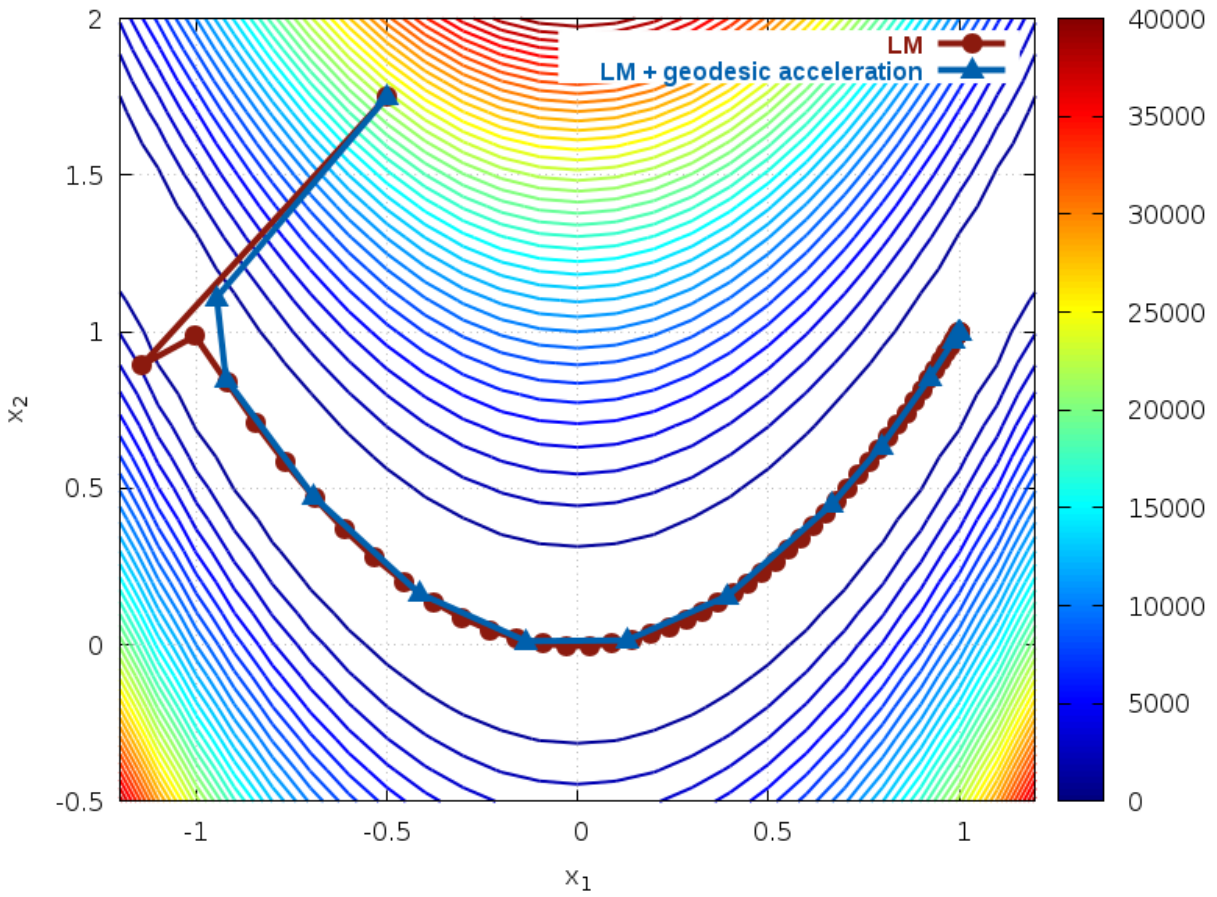


Fig. 41.2: Paths taken by solver for Rosenbrock function

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>

int
func_f (const gsl_vector * x, void *params, gsl_vector * f)
{
    double x1 = gsl_vector_get(x, 0);
    double x2 = gsl_vector_get(x, 1);

    gsl_vector_set(f, 0, 100.0 * (x2 - x1*x1));
    gsl_vector_set(f, 1, 1.0 - x1);

    return GSL_SUCCESS;
}

int
func_df (const gsl_vector * x, void *params, gsl_matrix * J)
{
    double x1 = gsl_vector_get(x, 0);

    gsl_matrix_set(J, 0, 0, -200.0*x1);
    gsl_matrix_set(J, 0, 1, 100.0);
    gsl_matrix_set(J, 1, 0, -1.0);
    gsl_matrix_set(J, 1, 1, 0.0);

    return GSL_SUCCESS;
}

int
func_fvv (const gsl_vector * x, const gsl_vector * v,
          void *params, gsl_vector * fvv)
{
    double v1 = gsl_vector_get(v, 0);

    gsl_vector_set(fvv, 0, -200.0 * v1 * v1);
    gsl_vector_set(fvv, 1, 0.0);

    return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
          const gsl_multifit_nlinear_workspace *w)
{
    gsl_vector * x = gsl_multifit_nlinear_position(w);

    /* print out current location */
    printf("%f %f\n",
           gsl_vector_get(x, 0),
```

(continues on next page)

(continued from previous page)

```

        gsl_vector_get(x, 1));
}

void
solve_system(gsl_vector *x0, gsl_multifit_nlinear_fdf *fdf,
             gsl_multifit_nlinear_parameters *params)
{
    const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
    const size_t max_iter = 200;
    const double xtol = 1.0e-8;
    const double gtol = 1.0e-8;
    const double ftol = 1.0e-8;
    const size_t n = fdf->n;
    const size_t p = fdf->p;
    gsl_multifit_nlinear_workspace *work =
        gsl_multifit_nlinear_alloc(T, params, n, p);
    gsl_vector * f = gsl_multifit_nlinear_residual(work);
    gsl_vector * x = gsl_multifit_nlinear_position(work);
    int info;
    double chisq0, chisq, rcond;

    /* initialize solver */
    gsl_multifit_nlinear_init(x0, fdf, work);

    /* store initial cost */
    gsl_blas_ddot(f, f, &chisq0);

    /* iterate until convergence */
    gsl_multifit_nlinear_driver(max_iter, xtol, gtol, ftol,
                               callback, NULL, &info, work);

    /* store final cost */
    gsl_blas_ddot(f, f, &chisq);

    /* store cond(J(x)) */
    gsl_multifit_nlinear_rcond(&rcond, work);

    /* print summary */

    fprintf(stderr, "NITER      = %zu\n", gsl_multifit_nlinear_niter(work));
    fprintf(stderr, "NFEV       = %zu\n", fdf->nevalf);
    fprintf(stderr, "NJEV       = %zu\n", fdf->nevaldf);
    fprintf(stderr, "NAEV       = %zu\n", fdf->nevalfv);
    fprintf(stderr, "initial cost = %.12e\n", chisq0);
    fprintf(stderr, "final cost   = %.12e\n", chisq);
    fprintf(stderr, "final x     = (%.12e, %.12e)\n",
                gsl_vector_get(x, 0), gsl_vector_get(x, 1));
    fprintf(stderr, "final cond(J) = %.12e\n", 1.0 / rcond);

    printf("\n\n");

    gsl_multifit_nlinear_free(work);
}

```

(continues on next page)

```

}

int
main (void)
{
  const size_t n = 2;
  const size_t p = 2;
  gsl_vector *f = gsl_vector_alloc(n);
  gsl_vector *x = gsl_vector_alloc(p);
  gsl_multifit_nlinear_fdf fdf;
  gsl_multifit_nlinear_parameters fdf_params =
    gsl_multifit_nlinear_default_parameters();

  /* print map of Phi(x1, x2) */
  {
    double x1, x2, chisq;
    double *f1 = gsl_vector_ptr(f, 0);
    double *f2 = gsl_vector_ptr(f, 1);

    for (x1 = -1.2; x1 < 1.3; x1 += 0.1)
      {
        for (x2 = -0.5; x2 < 2.1; x2 += 0.1)
          {
            gsl_vector_set(x, 0, x1);
            gsl_vector_set(x, 1, x2);
            func_f(x, NULL, f);

            chisq = (*f1) * (*f1) + (*f2) * (*f2);
            printf("%f %f %f\n", x1, x2, chisq);
          }
        printf("\n");
      }
    printf("\n\n");
  }

  /* define function to be minimized */
  fdf.f = func_f;
  fdf.df = func_df;
  fdf.fvv = func_fvv;
  fdf.n = n;
  fdf.p = p;
  fdf.params = NULL;

  /* starting point */
  gsl_vector_set(x, 0, -0.5);
  gsl_vector_set(x, 1, 1.75);

  fprintf(stderr, "=== Solving system without acceleration ===\n");
  fdf_params.trs = gsl_multifit_nlinear_trs_lm;
  solve_system(x, &fdf, &fdf_params);

  fprintf(stderr, "=== Solving system with acceleration ===\n");

```

(continues on next page)



(continued from previous page)

```

fdf_params.trs = gsl_multifit_nlinear_trs_lmaccel;
solve_system(x, &fdf, &fdf_params);

gsl_vector_free(f);
gsl_vector_free(x);

return 0;
}

```

### 41.12.3 Geodesic Acceleration Example 2

The following example fits a set of data to a Gaussian model using the Levenberg-Marquardt method with geodesic acceleration. The cost function is

$$\Phi(x) = \frac{1}{2} \sum_i f_i^2$$

$$f_i = y_i - Y(a, b, c; t_i)$$

where  $y_i$  is the measured data point at time  $t_i$ , and the model is specified by

$$Y(a, b, c; t) = a \exp \left[ -\frac{1}{2} \left( \frac{t - b}{c} \right)^2 \right]$$

The parameters  $a, b, c$  represent the amplitude, mean, and width of the Gaussian respectively. The program below generates the  $y_i$  data on  $[0, 1]$  using the values  $a = 5, b = 0.4, c = 0.15$  and adding random noise. The  $i$ -th row of the Jacobian is

$$J_{i,:} = \left( \frac{\partial f_i}{\partial a} \quad \frac{\partial f_i}{\partial b} \quad \frac{\partial f_i}{\partial c} \right) = \left( -e_i \quad -\frac{a}{c} z_i e_i \quad -\frac{a}{c} z_i^2 e_i \right)$$

where

$$z_i = \frac{t_i - b}{c}$$

$$e_i = \exp \left( -\frac{1}{2} z_i^2 \right)$$

In order to use geodesic acceleration, we need the second directional derivative of the residuals in the velocity direction,  $D_v^2 f_i = \sum_{\alpha\beta} v_\alpha v_\beta \partial_\alpha \partial_\beta f_i$ , where  $v$  is provided by the solver. To compute this, it is helpful to make a table of all second derivatives of the residuals  $f_i$  with respect to each combination of model parameters. This table is

$\frac{\partial}{\partial a}$	$\frac{\partial}{\partial b}$	$\frac{\partial}{\partial c}$
$0$	$-\frac{z_i}{c} e_i$	$-\frac{z_i}{c} e_i$
$\frac{a}{c^2}$	$\frac{a}{c^2} (1 - z_i^2) e_i$	$\frac{a}{c^2} z_i (2 - z_i^2) e_i$
$\frac{a}{c^2}$	$\frac{a}{c^2} z_i^2 (3 - z_i^2) e_i$	$\frac{a}{c^2} z_i^2 (3 - z_i^2) e_i$

The lower half of the table is omitted since it is symmetric. Then, the second directional derivative of  $f_i$  is

$$D_v^2 f_i = v_a^2 \partial_a^2 f_i + 2v_a v_b \partial_a \partial_b f_i + 2v_a v_c \partial_a \partial_c f_i + v_b^2 \partial_b^2 f_i + 2v_b v_c \partial_b \partial_c f_i + v_c^2 \partial_c^2 f_i$$

The factors of 2 come from the symmetry of the mixed second partial derivatives. The iteration is started using the initial guess  $a = 1, b = 0, c = 1$ . The program output is shown below:

```

iter 0: a = 1.0000, b = 0.0000, c = 1.0000, |a|/|v| = 0.0000 cond(J) = inf, |f(x)|
↳= 35.4785
iter 1: a = 1.5708, b = 0.5321, c = 0.5219, |a|/|v| = 0.3093 cond(J) = 29.0443, |f(x)|
↳= 31.1042
iter 2: a = 1.7387, b = 0.4040, c = 0.4568, |a|/|v| = 0.1199 cond(J) = 3.5256, |f(x)|
↳= 28.7217
iter 3: a = 2.2340, b = 0.3829, c = 0.3053, |a|/|v| = 0.3308 cond(J) = 4.5121, |f(x)|
↳= 23.8074
iter 4: a = 3.2275, b = 0.3952, c = 0.2243, |a|/|v| = 0.2784 cond(J) = 8.6499, |f(x)|
↳= 15.6003
iter 5: a = 4.3347, b = 0.3974, c = 0.1752, |a|/|v| = 0.2029 cond(J) = 15.1732, |f(x)|
↳= 7.5908
iter 6: a = 4.9352, b = 0.3992, c = 0.1536, |a|/|v| = 0.1001 cond(J) = 26.6621, |f(x)|
↳= 4.8402
iter 7: a = 5.0716, b = 0.3994, c = 0.1498, |a|/|v| = 0.0166 cond(J) = 34.6922, |f(x)|
↳= 4.7103
iter 8: a = 5.0828, b = 0.3994, c = 0.1495, |a|/|v| = 0.0012 cond(J) = 36.5422, |f(x)|
↳= 4.7095
iter 9: a = 5.0831, b = 0.3994, c = 0.1495, |a|/|v| = 0.0000 cond(J) = 36.6929, |f(x)|
↳= 4.7095
iter 10: a = 5.0831, b = 0.3994, c = 0.1495, |a|/|v| = 0.0000 cond(J) = 36.6975, |f(x)|
↳= 4.7095
iter 11: a = 5.0831, b = 0.3994, c = 0.1495, |a|/|v| = 0.0000 cond(J) = 36.6976, |f(x)|
↳= 4.7095
NITER      = 11
NFEV      = 18
NJEV      = 12
NAEV      = 17
initial cost = 1.258724737288e+03
final cost  = 2.217977560180e+01
final x     = (5.083101559156e+00, 3.994484109594e-01, 1.494898e-01)
final cond(J) = 3.669757713403e+01

```

We see the method converges after 11 iterations. For comparison the standard Levenberg-Marquardt method requires 26 iterations and so the Gaussian fitting problem benefits substantially from the geodesic acceleration correction. The column marked  $|a|/|v|$  above shows the ratio of the acceleration term to the velocity term as the iteration progresses. Larger values of this ratio indicate that the geodesic acceleration correction term is contributing substantial information to the solver relative to the standard LM velocity step.

The data and fitted model are shown in [Fig. 41.3](#).

The program is given below.

```

#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

struct data
{

```

(continues on next page)

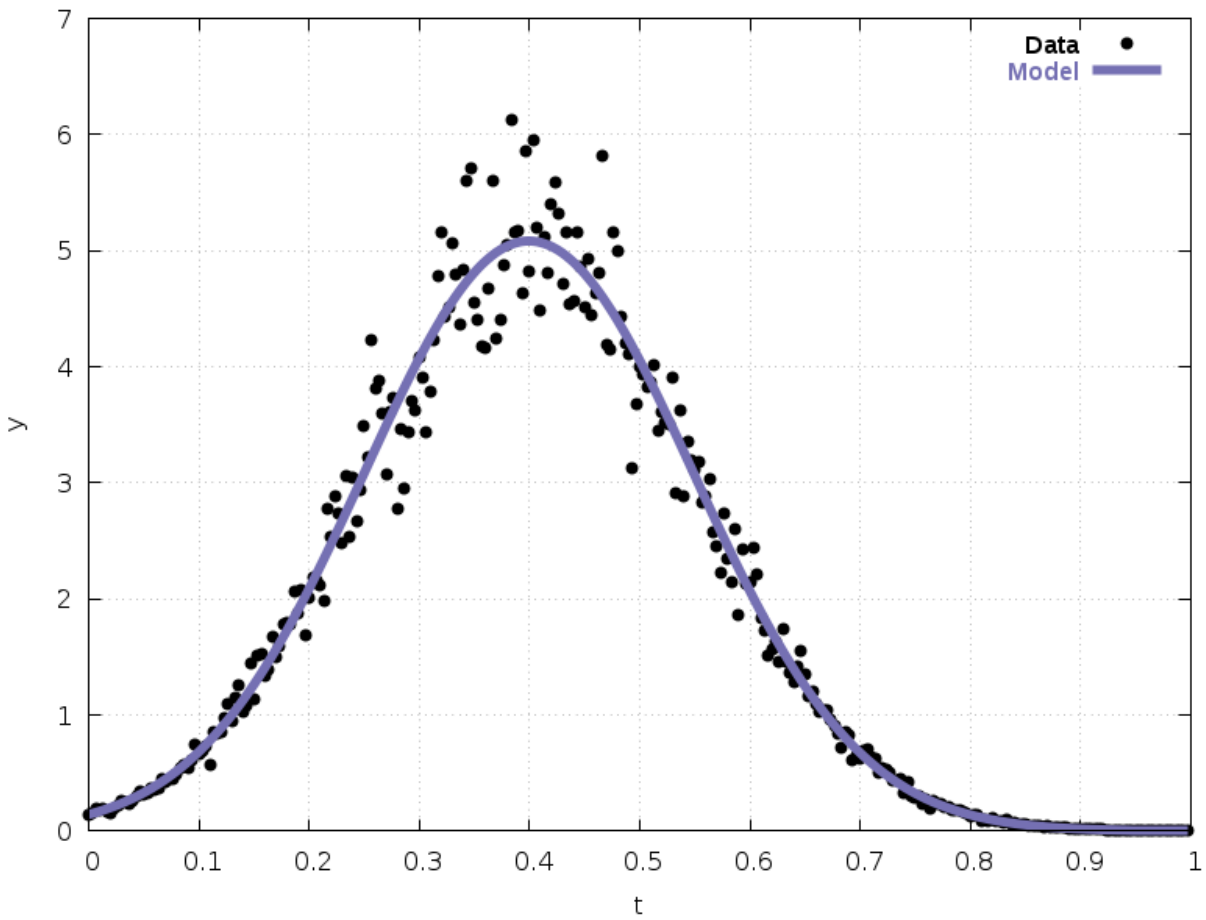


Fig. 41.3: Gaussian model fitted to data

(continued from previous page)

```
double *t;
double *y;
size_t n;
};

/* model function: a * exp( -1/2 * [ (t - b) / c ]^2 ) */
double
gaussian(const double a, const double b, const double c, const double t)
{
    const double z = (t - b) / c;
    return (a * exp(-0.5 * z * z));
}

int
func_f (const gsl_vector * x, void *params, gsl_vector * f)
{
    struct data *d = (struct data *) params;
    double a = gsl_vector_get(x, 0);
    double b = gsl_vector_get(x, 1);
    double c = gsl_vector_get(x, 2);
    size_t i;

    for (i = 0; i < d->n; ++i)
    {
        double ti = d->t[i];
        double yi = d->y[i];
        double y = gaussian(a, b, c, ti);

        gsl_vector_set(f, i, yi - y);
    }

    return GSL_SUCCESS;
}

int
func_df (const gsl_vector * x, void *params, gsl_matrix * J)
{
    struct data *d = (struct data *) params;
    double a = gsl_vector_get(x, 0);
    double b = gsl_vector_get(x, 1);
    double c = gsl_vector_get(x, 2);
    size_t i;

    for (i = 0; i < d->n; ++i)
    {
        double ti = d->t[i];
        double zi = (ti - b) / c;
        double ei = exp(-0.5 * zi * zi);

        gsl_matrix_set(J, i, 0, -ei);
        gsl_matrix_set(J, i, 1, -(a / c) * ei * zi);
        gsl_matrix_set(J, i, 2, -(a / c) * ei * zi * zi);
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    return GSL_SUCCESS;
}

int
func_fvv (const gsl_vector * x, const gsl_vector * v,
          void *params, gsl_vector * fvv)
{
    struct data *d = (struct data *) params;
    double a = gsl_vector_get(x, 0);
    double b = gsl_vector_get(x, 1);
    double c = gsl_vector_get(x, 2);
    double va = gsl_vector_get(v, 0);
    double vb = gsl_vector_get(v, 1);
    double vc = gsl_vector_get(v, 2);
    size_t i;

    for (i = 0; i < d->n; ++i)
    {
        double ti = d->t[i];
        double zi = (ti - b) / c;
        double ei = exp(-0.5 * zi * zi);
        double Dab = -zi * ei / c;
        double Dac = -zi * zi * ei / c;
        double Dbb = a * ei / (c * c) * (1.0 - zi*zi);
        double Dbc = a * zi * ei / (c * c) * (2.0 - zi*zi);
        double Dcc = a * zi * zi * ei / (c * c) * (3.0 - zi*zi);
        double sum;

        sum = 2.0 * va * vb * Dab +
              2.0 * va * vc * Dac +
              vb * vb * Dbb +
              2.0 * vb * vc * Dbc +
              vc * vc * Dcc;

        gsl_vector_set(fvv, i, sum);
    }

    return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
          const gsl_multifit_nlinear_workspace *w)
{
    gsl_vector *f = gsl_multifit_nlinear_residual(w);
    gsl_vector *x = gsl_multifit_nlinear_position(w);
    double avratio = gsl_multifit_nlinear_avratio(w);
    double rcond;

    (void) params; /* not used */
}

```

(continues on next page)

(continued from previous page)

```

/* compute reciprocal condition number of J(x) */
gsl_multifit_nlinear_rcond(&rcond, w);

fprintf(stderr, "iter %2zu: a = %.4f, b = %.4f, c = %.4f, |a|/|v| = %.4f cond(J) = %8.
→4f, |f(x)| = %.4f\n",
    iter,
    gsl_vector_get(x, 0),
    gsl_vector_get(x, 1),
    gsl_vector_get(x, 2),
    avratio,
    1.0 / rcond,
    gsl_blas_dnorm2(f));
}

void
solve_system(gsl_vector *x, gsl_multifit_nlinear_fdf *fdf,
             gsl_multifit_nlinear_parameters *params)
{
    const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
    const size_t max_iter = 200;
    const double xtol = 1.0e-8;
    const double gtol = 1.0e-8;
    const double ftol = 1.0e-8;
    const size_t n = fdf->n;
    const size_t p = fdf->p;
    gsl_multifit_nlinear_workspace *work =
        gsl_multifit_nlinear_alloc(T, params, n, p);
    gsl_vector * f = gsl_multifit_nlinear_residual(work);
    gsl_vector * y = gsl_multifit_nlinear_position(work);
    int info;
    double chisq0, chisq, rcond;

    /* initialize solver */
    gsl_multifit_nlinear_init(x, fdf, work);

    /* store initial cost */
    gsl_blas_ddot(f, f, &chisq0);

    /* iterate until convergence */
    gsl_multifit_nlinear_driver(max_iter, xtol, gtol, ftol,
                               callback, NULL, &info, work);

    /* store final cost */
    gsl_blas_ddot(f, f, &chisq);

    /* store cond(J(x)) */
    gsl_multifit_nlinear_rcond(&rcond, work);

    gsl_vector_memcpy(x, y);

    /* print summary */

```

(continues on next page)

(continued from previous page)

```

fprintf(stderr, "NITER      = %zu\n", gsl_multifit_nlinear_niter(work));
fprintf(stderr, "NFEV      = %zu\n", fdf->nevalf);
fprintf(stderr, "NJEV      = %zu\n", fdf->nevaldf);
fprintf(stderr, "NAEV      = %zu\n", fdf->nevalfvv);
fprintf(stderr, "initial cost = %.12e\n", chisq0);
fprintf(stderr, "final cost  = %.12e\n", chisq);
fprintf(stderr, "final x      = (%.12e, %.12e, %.12e)\n",
        gsl_vector_get(x, 0), gsl_vector_get(x, 1), gsl_vector_get(x, 2));
fprintf(stderr, "final cond(J) = %.12e\n", 1.0 / rcond);

gsl_multifit_nlinear_free(work);
}

int
main (void)
{
  const size_t n = 300; /* number of data points to fit */
  const size_t p = 3;  /* number of model parameters */
  const double a = 5.0; /* amplitude */
  const double b = 0.4; /* center */
  const double c = 0.15; /* width */
  const gsl_rng_type * T = gsl_rng_default;
  gsl_vector *f = gsl_vector_alloc(n);
  gsl_vector *x = gsl_vector_alloc(p);
  gsl_multifit_nlinear_fdf fdf;
  gsl_multifit_nlinear_parameters fdf_params =
    gsl_multifit_nlinear_default_parameters();
  struct data fit_data;
  gsl_rng * r;
  size_t i;

  gsl_rng_env_setup ();
  r = gsl_rng_alloc (T);

  fit_data.t = malloc(n * sizeof(double));
  fit_data.y = malloc(n * sizeof(double));
  fit_data.n = n;

  /* generate synthetic data with noise */
  for (i = 0; i < n; ++i)
  {
    double t = (double)i / (double) n;
    double y0 = gaussian(a, b, c, t);
    double dy = gsl_ran_gaussian (r, 0.1 * y0);

    fit_data.t[i] = t;
    fit_data.y[i] = y0 + dy;
  }

  /* define function to be minimized */
  fdf.f = func_f;

```

(continues on next page)

(continued from previous page)

```

fdf.df = func_df;
fdf.fvv = func_fvv;
fdf.n = n;
fdf.p = p;
fdf.params = &fit_data;

/* starting point */
gsl_vector_set(x, 0, 1.0);
gsl_vector_set(x, 1, 0.0);
gsl_vector_set(x, 2, 1.0);

fdf_params.trs = gsl_multifit_nlinear_trs_lmaccel;
solve_system(x, &fdf, &fdf_params);

/* print data and model */
{
  double A = gsl_vector_get(x, 0);
  double B = gsl_vector_get(x, 1);
  double C = gsl_vector_get(x, 2);

  for (i = 0; i < n; ++i)
  {
    double ti = fit_data.t[i];
    double yi = fit_data.y[i];
    double fi = gaussian(A, B, C, ti);

    printf("%f %f %f\n", ti, yi, fi);
  }
}

gsl_vector_free(f);
gsl_vector_free(x);
gsl_rng_free(r);

return 0;
}

```

#### 41.12.4 Comparing TRS Methods Example

The following program compares all available nonlinear least squares trust-region subproblem (TRS) methods on the Branin function, a common optimization test problem. The cost function is

$$\Phi(x) = \frac{1}{2}(f_1^2 + f_2^2)$$

$$f_1 = x_2 + a_1x_1^2 + a_2x_1 + a_3$$

$$f_2 = \sqrt{a_4}\sqrt{1 + (1 - a_5)\cos x_1}$$

with  $a_1 = -\frac{5.1}{4\pi^2}$ ,  $a_2 = \frac{5}{\pi}$ ,  $a_3 = -6$ ,  $a_4 = 10$ ,  $a_5 = \frac{1}{8\pi}$ . There are three minima of this function in the range  $(x_1, x_2) \in [-5, 15] \times [-5, 15]$ . The program below uses the starting point  $(x_1, x_2) = (6, 14.5)$  and calculates the solution with all available nonlinear least squares TRS methods. The program output is shown below:



Method	NITER	NFEV	NJEV	Initial Cost	Final cost	Final cond(J)	
↪Final x							
levenberg-marquardt	20	27	21	1.9874e+02	3.9789e-01	6.1399e+07	(-
↪3.14e+00, 1.23e+01)							
levenberg-marquardt+accel	27	36	28	1.9874e+02	3.9789e-01	1.4465e+07	(3.
↪14e+00, 2.27e+00)							
dogleg	23	64	23	1.9874e+02	3.9789e-01	5.0692e+08	(3.
↪14e+00, 2.28e+00)							
double-dogleg	24	69	24	1.9874e+02	3.9789e-01	3.4879e+07	(3.
↪14e+00, 2.27e+00)							
2D-subspace	23	54	24	1.9874e+02	3.9789e-01	2.5142e+07	(3.
↪14e+00, 2.27e+00)							

The first row of output above corresponds to standard Levenberg-Marquardt, while the second row includes geodesic acceleration. We see that the standard LM method converges to the minimum at  $(-\pi, 12.275)$  and also uses the least number of iterations and Jacobian evaluations. All other methods converge to the minimum  $(\pi, 2.275)$  and perform similarly in terms of number of Jacobian evaluations. We see that  $J$  is fairly ill-conditioned at both minima, indicating that the QR (or SVD) solver is the best choice for this problem. Since there are only two parameters in this optimization problem, we can easily visualize the paths taken by each method, which are shown in Fig. 41.4. The figure shows contours of the cost function  $\Phi(x_1, x_2)$  which exhibits three global minima in the range  $[-5, 15] \times [-5, 15]$ . The paths taken by each solver are shown as colored lines.

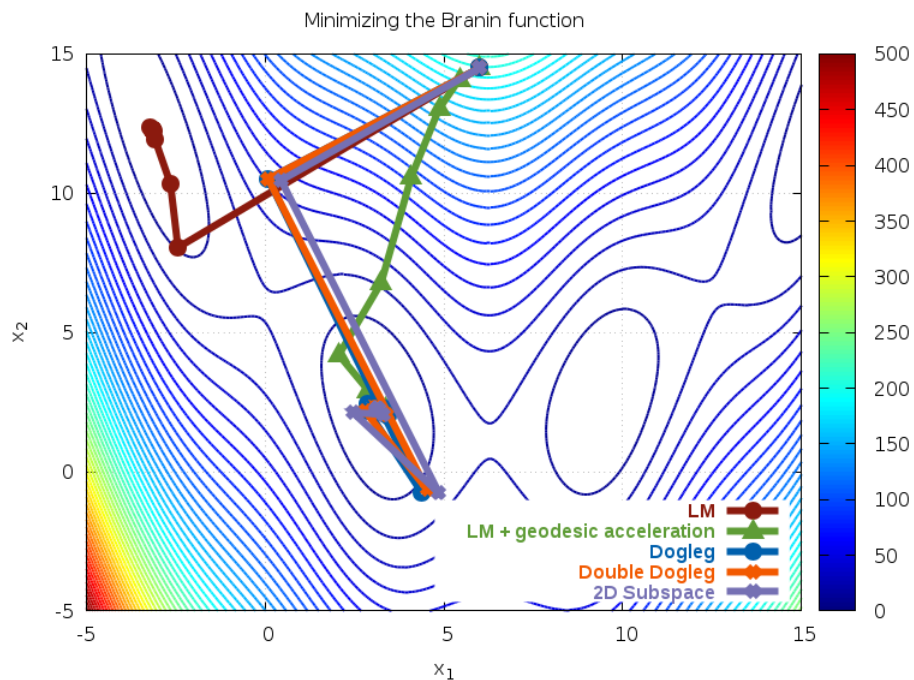


Fig. 41.4: Paths taken for different TRS methods for the Branin function

The program is given below.

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
```

(continues on next page)

```
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multifit_nlinear.h>

/* parameters to model */
struct model_params
{
    double a1;
    double a2;
    double a3;
    double a4;
    double a5;
};

/* Branin function */
int
func_f (const gsl_vector * x, void *params, gsl_vector * f)
{
    struct model_params *par = (struct model_params *) params;
    double x1 = gsl_vector_get(x, 0);
    double x2 = gsl_vector_get(x, 1);
    double f1 = x2 + par->a1 * x1 * x1 + par->a2 * x1 + par->a3;
    double f2 = sqrt(par->a4) * sqrt(1.0 + (1.0 - par->a5) * cos(x1));

    gsl_vector_set(f, 0, f1);
    gsl_vector_set(f, 1, f2);

    return GSL_SUCCESS;
}

int
func_df (const gsl_vector * x, void *params, gsl_matrix * J)
{
    struct model_params *par = (struct model_params *) params;
    double x1 = gsl_vector_get(x, 0);
    double f2 = sqrt(par->a4) * sqrt(1.0 + (1.0 - par->a5) * cos(x1));

    gsl_matrix_set(J, 0, 0, 2.0 * par->a1 * x1 + par->a2);
    gsl_matrix_set(J, 0, 1, 1.0);

    gsl_matrix_set(J, 1, 0, -0.5 * par->a4 / f2 * (1.0 - par->a5) * sin(x1));
    gsl_matrix_set(J, 1, 1, 0.0);

    return GSL_SUCCESS;
}

int
func_fvv (const gsl_vector * x, const gsl_vector * v,
          void *params, gsl_vector * fvv)
{
    struct model_params *par = (struct model_params *) params;
    double x1 = gsl_vector_get(x, 0);
```

(continues on next page)

(continued from previous page)

```

double v1 = gsl_vector_get(v, 0);
double c = cos(x1);
double s = sin(x1);
double f2 = sqrt(par->a4) * sqrt(1.0 + (1.0 - par->a5) * c);
double t = 0.5 * par->a4 * (1.0 - par->a5) / f2;

gsl_vector_set(fvv, 0, 2.0 * par->a1 * v1 * v1);
gsl_vector_set(fvv, 1, -t * (c + s*s/f2) * v1 * v1);

return GSL_SUCCESS;
}

void
callback(const size_t iter, void *params,
         const gsl_multifit_nlinear_workspace *w)
{
    gsl_vector * x = gsl_multifit_nlinear_position(w);
    double x1 = gsl_vector_get(x, 0);
    double x2 = gsl_vector_get(x, 1);

    /* print out current location */
    printf("%f %f\n", x1, x2);
}

void
solve_system(gsl_vector *x0, gsl_multifit_nlinear_fdf *fdf,
             gsl_multifit_nlinear_parameters *params)
{
    const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
    const size_t max_iter = 200;
    const double xtol = 1.0e-8;
    const double gtol = 1.0e-8;
    const double ftol = 1.0e-8;
    const size_t n = fdf->n;
    const size_t p = fdf->p;
    gsl_multifit_nlinear_workspace *work =
        gsl_multifit_nlinear_alloc(T, params, n, p);
    gsl_vector * f = gsl_multifit_nlinear_residual(work);
    gsl_vector * x = gsl_multifit_nlinear_position(work);
    int info;
    double chisq0, chisq, rcond;

    printf("# %s/%s\n",
           gsl_multifit_nlinear_name(work),
           gsl_multifit_nlinear_trs_name(work));

    /* initialize solver */
    gsl_multifit_nlinear_init(x0, fdf, work);

    /* store initial cost */
    gsl_blas_ddot(f, f, &chisq0);

```

(continues on next page)

(continued from previous page)

```

/* iterate until convergence */
gsl_multifit_nlinear_driver(max_iter, xtol, gtol, ftol,
                           callback, NULL, &info, work);

/* store final cost */
gsl_blas_ddot(f, f, &chisq);

/* store cond(J(x)) */
gsl_multifit_nlinear_rcond(&rcond, work);

/* print summary */
fprintf(stderr, "%-25s %-6zu %-5zu %-5zu %-13.4e %-12.4e %-13.4e (%.2e, %.2e)\n",
        gsl_multifit_nlinear_trs_name(work),
        gsl_multifit_nlinear_niter(work),
        fdf->nevalf,
        fdf->nevaldf,
        chisq0,
        chisq,
        1.0 / rcond,
        gsl_vector_get(x, 0),
        gsl_vector_get(x, 1));

printf("\n\n");

gsl_multifit_nlinear_free(work);
}

int
main (void)
{
    const size_t n = 2;
    const size_t p = 2;
    gsl_vector *f = gsl_vector_alloc(n);
    gsl_vector *x = gsl_vector_alloc(p);
    gsl_multifit_nlinear_fdf fdf;
    gsl_multifit_nlinear_parameters fdf_params =
        gsl_multifit_nlinear_default_parameters();
    struct model_params params;

    params.a1 = -5.1 / (4.0 * M_PI * M_PI);
    params.a2 = 5.0 / M_PI;
    params.a3 = -6.0;
    params.a4 = 10.0;
    params.a5 = 1.0 / (8.0 * M_PI);

    /* print map of Phi(x1, x2) */
    {
        double x1, x2, chisq;

        for (x1 = -5.0; x1 < 15.0; x1 += 0.1)
            {
                for (x2 = -5.0; x2 < 15.0; x2 += 0.1)

```

(continues on next page)

(continued from previous page)

```

    {
        gsl_vector_set(x, 0, x1);
        gsl_vector_set(x, 1, x2);
        func_f(x, &params, f);

        gsl_blas_ddot(f, f, &chisq);

        printf("%f %f %f\n", x1, x2, chisq);
    }
    printf("\n");
}
printf("\n\n");
}

/* define function to be minimized */
fdf.f = func_f;
fdf.df = func_df;
fdf.fvv = func_fvv;
fdf.n = n;
fdf.p = p;
fdf.params = &params;

/* starting point */
gsl_vector_set(x, 0, 6.0);
gsl_vector_set(x, 1, 14.5);

fprintf(stderr, "%-25s %-6s %-5s %-5s %-13s %-12s %-13s %-15s\n",
        "Method", "NITER", "NFEV", "NJEV", "Initial Cost",
        "Final cost", "Final cond(J)", "Final x");

fdf_params.trs = gsl_multifit_nlinear_trs_lm;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multifit_nlinear_trs_lmaccel;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multifit_nlinear_trs_dogleg;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multifit_nlinear_trs_ddogleg;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multifit_nlinear_trs_subspace2D;
solve_system(x, &fdf, &fdf_params);

gsl_vector_free(f);
gsl_vector_free(x);

return 0;
}

```

### 41.12.5 Large Nonlinear Least Squares Example

The following program illustrates the large nonlinear least squares solvers on a system with significant sparse structure in the Jacobian. The cost function is

$$\begin{aligned}\Phi(x) &= \frac{1}{2} \sum_{i=1}^{p+1} f_i^2 \\ f_i &= \sqrt{\alpha}(x_i - 1), \quad 1 \leq i \leq p \\ f_{p+1} &= \|x\|^2 - \frac{1}{4}\end{aligned}$$

with  $\alpha = 10^{-5}$ . The residual  $f_{p+1}$  imposes a constraint on the  $p$  parameters  $x$ , to ensure that  $\|x\|^2 \approx \frac{1}{4}$ . The  $(p+1)$ -by- $p$  Jacobian for this system is

$$J(x) = \begin{pmatrix} \sqrt{\alpha}I_p \\ 2x^T \end{pmatrix}$$

and the normal equations matrix is

$$J^T J = \alpha I_p + 4xx^T$$

Finally, the second directional derivative of  $f$  for the geodesic acceleration method is

$$f_{vv} = D_v^2 f = \begin{pmatrix} 0 \\ 2\|v\|^2 \end{pmatrix}$$

Since the upper  $p$ -by- $p$  block of  $J$  is diagonal, this sparse structure should be exploited in the nonlinear solver. For comparison, the following program solves the system for  $p = 2000$  using the dense direct Cholesky solver based on the normal equations matrix  $J^T J$ , as well as the iterative Steihaug-Toint solver, based on sparse matrix-vector products  $Ju$  and  $J^T u$ . The program output is shown below:

Method	NITER	NFEV	NJUEV	NJTJEV	NAEV	Init Cost	Final cost	cond(J)	␣
↪Final  x ^2 Time (s)									
levenberg-marquardt	25	31	26	26	0	7.1218e+18	1.9555e-02	447.50	2.
↪5044e-01 46.28									
levenberg-marquardt+accel	22	23	45	23	22	7.1218e+18	1.9555e-02	447.64	2.
↪5044e-01 33.92									
dogleg	37	87	36	36	0	7.1218e+18	1.9555e-02	447.59	2.
↪5044e-01 56.05									
double-dogleg	35	88	34	34	0	7.1218e+18	1.9555e-02	447.62	2.
↪5044e-01 52.65									
2D-subspace	37	88	36	36	0	7.1218e+18	1.9555e-02	447.71	2.
↪5044e-01 59.75									
stehaug-toint	35	88	345	0	0	7.1218e+18	1.9555e-02	inf	2.
↪5044e-01 0.09									

The first five rows use methods based on factoring the dense  $J^T J$  matrix while the last row uses the iterative Steihaug-Toint method. While the number of Jacobian matrix-vector products (NJUEV) is less for the dense methods, the added time to construct and factor the  $J^T J$  matrix (NJTJEV) results in a much larger runtime than the iterative method (see last column).

The program is given below.

```
#include <stdlib.h>
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```

#include <sys/time.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_multilarge_nlinear.h>
#include <gsl/gsl_splblas.h>
#include <gsl/gsl_spmatrix.h>

/* parameters for functions */
struct model_params
{
    double alpha;
    gsl_spmatrix *J;
};

/* penalty function */
int
penalty_f (const gsl_vector * x, void *params, gsl_vector * f)
{
    struct model_params *par = (struct model_params *) params;
    const double sqrt_alpha = sqrt(par->alpha);
    const size_t p = x->size;
    size_t i;
    double sum = 0.0;

    for (i = 0; i < p; ++i)
    {
        double xi = gsl_vector_get(x, i);

        gsl_vector_set(f, i, sqrt_alpha*(xi - 1.0));

        sum += xi * xi;
    }

    gsl_vector_set(f, p, sum - 0.25);

    return GSL_SUCCESS;
}

int
penalty_df (CBLAS_TRANSPOSE_t TransJ, const gsl_vector * x,
            const gsl_vector * u, void * params, gsl_vector * v,
            gsl_matrix * JTJ)
{
    struct model_params *par = (struct model_params *) params;
    const size_t p = x->size;
    size_t j;

    /* store 2*x in last row of J */
    for (j = 0; j < p; ++j)
    {
        double xj = gsl_vector_get(x, j);

```

(continues on next page)

(continued from previous page)

```

    gsl_spmatrix_set(par->J, p, j, 2.0 * xj);
}

/* compute v = op(J) u */
if (v)
    gsl_splblas_dgemv(TransJ, 1.0, par->J, u, 0.0, v);

if (JTJ)
{
    gsl_vector_view diag = gsl_matrix_diagonal(JTJ);

    /* compute J^T J = [ alpha*I_p + 4 x x^T ] */
    gsl_matrix_set_zero(JTJ);

    /* store 4 x x^T in lower half of JTJ */
    gsl_blas_dsyr(CblasLower, 4.0, x, JTJ);

    /* add alpha to diag(JTJ) */
    gsl_vector_add_constant(&diag.vector, par->alpha);
}

return GSL_SUCCESS;
}

int
penalty_fvv (const gsl_vector * x, const gsl_vector * v,
            void *params, gsl_vector * fvv)
{
    const size_t p = x->size;
    double normv = gsl_blas_dnrm2(v);

    gsl_vector_set_zero(fvv);
    gsl_vector_set(fvv, p, 2.0 * normv * normv);

    (void)params; /* avoid unused parameter warning */

    return GSL_SUCCESS;
}

void
solve_system(const gsl_vector *x0, gsl_multilarge_nlinear_fdf *fdf,
            gsl_multilarge_nlinear_parameters *params)
{
    const gsl_multilarge_nlinear_type *T = gsl_multilarge_nlinear_trust;
    const size_t max_iter = 200;
    const double xtol = 1.0e-8;
    const double gtol = 1.0e-8;
    const double ftol = 1.0e-8;
    const size_t n = fdf->n;
    const size_t p = fdf->p;
    gsl_multilarge_nlinear_workspace *work =
        gsl_multilarge_nlinear_alloc(T, params, n, p);

```

(continues on next page)



(continued from previous page)

```

gsl_vector * f = gsl_multilarge_nlinear_residual(work);
gsl_vector * x = gsl_multilarge_nlinear_position(work);
int info;
double chisq0, chisq, rcond, xsq;
struct timeval tv0, tv1;

gettimeofday(&tv0, NULL);

/* initialize solver */
gsl_multilarge_nlinear_init(x0, fdf, work);

/* store initial cost */
gsl_blas_ddot(f, f, &chisq0);

/* iterate until convergence */
gsl_multilarge_nlinear_driver(max_iter, xtol, gtol, ftol,
                             NULL, NULL, &info, work);

gettimeofday(&tv1, NULL);

/* store final cost */
gsl_blas_ddot(f, f, &chisq);

/* compute final ||x||^2 */
gsl_blas_ddot(x, x, &xsq);

/* store cond(J(x)) */
gsl_multilarge_nlinear_rcond(&rcond, work);

/* print summary */
fprintf(stderr, "%-25s %-5zu %-4zu %-5zu %-6zu %-4zu %-10.4e %-10.4e %-7.2f %-11.4e %
↪2f\n",
        gsl_multilarge_nlinear_trs_name(work),
        gsl_multilarge_nlinear_niter(work),
        fdf->nevalf,
        fdf->nevaldfu,
        fdf->nevaldf2,
        fdf->nevalfvv,
        chisq0,
        chisq,
        1.0 / rcond,
        xsq,
        (tv1.tv_sec - tv0.tv_sec) + 1.0e-6 * (tv1.tv_usec - tv0.tv_usec));

gsl_multilarge_nlinear_free(work);
}

int
main (void)
{
    const size_t p = 2000;
    const size_t n = p + 1;

```

(continues on next page)

(continued from previous page)

```

gsl_vector *f = gsl_vector_alloc(n);
gsl_vector *x = gsl_vector_alloc(p);

/* allocate sparse Jacobian matrix with 2*p non-zero elements in triplet format */
gsl_spmatrix *J = gsl_spmatrix_alloc_nzmax(n, p, 2 * p, GSL_SPMATRIX_TRIPLET);

gsl_multilarge_nlinear_fdf fdf;
gsl_multilarge_nlinear_parameters fdf_params =
    gsl_multilarge_nlinear_default_parameters();
struct model_params params;
size_t i;

params.alpha = 1.0e-5;
params.J = J;

/* define function to be minimized */
fdf.f = penalty_f;
fdf.df = penalty_df;
fdf.fvv = penalty_fvv;
fdf.n = n;
fdf.p = p;
fdf.params = &params;

for (i = 0; i < p; ++i)
{
    /* starting point */
    gsl_vector_set(x, i, i + 1.0);

    /* store sqrt(alpha)*I_p in upper p-by-p block of J */
    gsl_spmatrix_set(J, i, i, sqrt(params.alpha));
}

fprintf(stderr, "%-25s %-4s %-4s %-5s %-6s %-4s %-10s %-10s %-7s %-11s %-10s\n",
    "Method", "NITER", "NFEV", "NJUEV", "NJTJEV", "NAEV", "Init Cost",
    "Final cost", "cond(J)", "Final |x|^2", "Time (s)");

fdf_params.scale = gsl_multilarge_nlinear_scale_levenberg;

fdf_params.trs = gsl_multilarge_nlinear_trs_lm;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multilarge_nlinear_trs_lmaccel;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multilarge_nlinear_trs_dogleg;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multilarge_nlinear_trs_ddogleg;
solve_system(x, &fdf, &fdf_params);

fdf_params.trs = gsl_multilarge_nlinear_trs_subspace2D;
solve_system(x, &fdf, &fdf_params);

```

(continues on next page)

(continued from previous page)

```
fdf_params.trs = gsl_multilarge_nlinear_trs_cgst;
solve_system(x, &fdf, &fdf_params);

gsl_vector_free(f);
gsl_vector_free(x);
gsl_spmatrix_free(J);

return 0;
}
```

## 41.13 References and Further Reading

The following publications are relevant to the algorithms described in this section,

- J.J. Moré, *The Levenberg-Marquardt Algorithm: Implementation and Theory*, Lecture Notes in Mathematics, v630 (1978), ed G. Watson.
- H. B. Nielsen, “Damping Parameter in Marquardt’s Method”, IMM Department of Mathematical Modeling, DTU, Tech. Report IMM-REP-1999-05 (1999).
- K. Madsen and H. B. Nielsen, “Introduction to Optimization and Data Fitting”, IMM Department of Mathematical Modeling, DTU, 2010.
- J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, SIAM, 1996.
- M. K. Transtrum, B. B. Machta, and J. P. Sethna, Geometry of nonlinear least squares with applications to sloppy models and optimization, *Phys. Rev. E* 83, 036701, 2011.
- M. K. Transtrum and J. P. Sethna, Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization, arXiv:1201.5885, 2012.
- J.J. Moré, B.S. Garbow, K.E. Hillstom, “Testing Unconstrained Optimization Software”, *ACM Transactions on Mathematical Software*, Vol 7, No 1 (1981), p 17–41.
- H. B. Nielsen, “UCTP Test Problems for Unconstrained Optimization”, IMM Department of Mathematical Modeling, DTU, Tech. Report IMM-REP-2000-17 (2000).



## BASIS SPLINES

This chapter describes functions for the computation of smoothing basis splines (B-splines). A smoothing spline differs from an interpolating spline in that the resulting curve is not required to pass through each data point. For information about interpolating splines, see *Interpolation*.

The header file `gsl_bspline.h` contains the prototypes for the bspline functions and related declarations.

### 42.1 Overview

Basis splines are commonly used to fit smooth curves to data sets. They are defined on some interval  $[a, b]$  which is sub-divided into smaller intervals by defining a non-decreasing sequence of *knots*. Between any two adjacent knots, a B-spline of order  $k$  is a polynomial of degree  $k - 1$ , and so B-splines defined over the whole interval  $[a, b]$  are called *piecewise polynomials*. In order to generate a smooth spline spanning the full interval, the different polynomial pieces are often required to match one or more derivatives at the knots.

B-splines of order  $k$  are basis functions for spline functions of the same order, so that all possible spline functions on some interval  $[a, b]$  can be expressed as linear combinations of B-splines,

$$f(x; \mathbf{t}) = \sum_{i=1}^n c_i B_{i,k}(x; \mathbf{t}), \quad a \leq x \leq b$$

Here, the  $n$  coefficients  $c_i$  are known as *control points* and are often determined from a least squares fit. The  $B_{i,k}(x; \mathbf{t})$  are the basis splines of order  $k$ , and depend on a knot vector  $\mathbf{t}$ , which is a non-decreasing sequence of  $n + k$  knots

$$\mathbf{t} = \{t_1, t_2, \dots, t_{n+k}\}$$

On each knot interval  $t_i \leq x < t_{i+1}$ , the B-splines  $B_{i,k}(x; \mathbf{t})$  are polynomials of degree  $k - 1$ . The  $n$  basis splines of order  $k$  are defined by

$$B_{i,1}(x) = \begin{cases} 1, & t_i \leq x < t_{i+1} \\ 0, & \text{else} \end{cases}$$
$$B_{i,k}(x) = \frac{(x - t_i)}{(t_{i+k-1} - t_i)} B_{i,k-1}(x) + \frac{(t_{i+k} - x)}{(t_{i+k} - t_{i+1})} B_{i+1,k-1}(x)$$

for  $i = 1, \dots, n$ . The common case of cubic B-splines is given by  $k = 4$ . The above recurrence relation can be evaluated in a numerically stable way by the de Boor algorithm.

In what follows, we will drop the  $\mathbf{t}$  and simply refer to the spline functions as  $f(x)$  and  $B_{i,k}(x)$  for simplicity, with the understanding that each set of spline basis functions depends on the chosen knot vector  $\mathbf{t}$ . We will also occasionally drop the order  $k$  and use  $B_i(x)$  and  $B_{i,k}(x)$  interchangeably.

### 42.1.1 Vector Splines

The spline above can be generalized to vector control points,

$$\mathbf{f}(x) = \sum_{i=1}^n \mathbf{c}_i B_{i,k}(x)$$

where the  $\mathbf{c}_i$  may have some length  $m$ .

### 42.1.2 Knot vector

In practice, one will often identify locations inside the interval  $[a, b]$  where it makes sense to end one polynomial piece and begin another. These locations are called *interior knots* or *breakpoints*. It may be desirable to increase the density of knots in regions with rapid changes, and decrease the density when the underlying data is changing slowly. For some applications it may be simply best to distribute the knots with equal spacing on the interval, which are called *uniform knots*.

The full knot vector will look like

$$\mathbf{t} = \underbrace{\{t_1, \dots, t_{k-1}\}}_{k-1}, \underbrace{\{t_k, \dots, t_{n+1}\}}_{n-k+2 \text{ interior knots}}, \underbrace{\{t_{n+2}, \dots, t_{n+k}\}}_{k-1}$$

The  $n - k + 2$  interior knots are in  $[a, b]$ . In order to have a full basis of B-splines, we require an additional  $2(k - 1)$  *exterior knots*,  $\{t_1, \dots, t_{k-1}\}$  and  $\{t_{n+2}, \dots, t_{n+k}\}$ . These knots must satisfy

$$\begin{aligned} t_1 &\leq \dots \leq t_{k-1} = a \\ b &= t_{n+2} \leq \dots \leq t_{n+k} \end{aligned}$$

but can otherwise be arbitrary.

## 42.2 Allocation of B-splines

type `gsl_bspline_workspace`

The computation of B-spline functions requires a preallocated workspace.

`gsl_bspline_workspace *gsl_bspline_alloc(const size_t k, const size_t nbreak)`

`gsl_bspline_workspace *gsl_bspline_alloc_ncontrol(const size_t k, const size_t ncontrol)`

These functions allocate a workspace for computing B-splines of order  $k$ . The number of breakpoints is given by `nbreak`. Alternatively, the number of control points  $n$  may be specified in `ncontrol`. The relationship is `nbreak = n - k + 2`. Cubic B-splines are specified by  $k = 4$ . The size of the workspace is  $O(2k^2 + 5k + \text{nbreak})$ .

void `gsl_bspline_free(gsl_bspline_workspace *w)`

This function frees the memory associated with the workspace  $w$ .

## 42.3 Initializing the knots vector

int **gsl\_bspline\_init\_augment**(const *gsl\_vector* \*tau, *gsl\_bspline\_workspace* \*w)

This function initializes the knot vector  $\mathbf{t}$  using the non-decreasing interior knot vector  $\mathbf{tau}$  of length  $n - k + 2$  by choosing the exterior knots to be equal to the first and last interior knot. The total knot vector is

$$\mathbf{t} = \left\{ \underbrace{\tau_1, \dots, \tau_1}_{k-1 \text{ copies}}, \tau_1, \tau_2, \dots, \tau_{n-k+2}, \underbrace{\tau_{n-k+2}, \dots, \tau_{n-k+2}}_{k-1 \text{ copies}} \right\}$$

where  $\tau_i$  is specified by the  $i$ -th element of  $\mathbf{tau}$ .

This function does not verify that the  $\mathbf{tau}$  vector is non-decreasing, so the user must ensure this.

int **gsl\_bspline\_init\_uniform**(const double a, const double b, *gsl\_bspline\_workspace* \*w)

This function constructs a knot vector using uniformly spaced interior knots on  $[a, b]$  and sets the exterior knots to the interval endpoints. The total knot vector is

$$\mathbf{t} = \left\{ \underbrace{a, \dots, a}_{k-1 \text{ copies}}, \tau_1, \tau_2, \dots, \tau_{n-k+2}, \underbrace{b, \dots, b}_{k-1 \text{ copies}} \right\}$$

with

$$\tau_i = a + \frac{i-1}{n-k+1}(b-a), \quad i = 1, \dots, n-k+2$$

int **gsl\_bspline\_init\_periodic**(const double a, const double b, *gsl\_bspline\_workspace* \*w)

This function constructs a knot vector suitable for defining a periodic B-spline on  $[a, b]$  with a period  $P = b - a$ . There are several ways to choose a periodic knot vector. Here, we adopt the approach of Dierckx, 1982, and require

$$\begin{aligned} t_i &= t_{i+n-k+1} - P, & i &= 1, \dots, k-1 \\ t_{n-k+1+i} &= t_i + P, & i &= k+1, \dots, 2k-1 \end{aligned}$$

We further define a uniform spacing between each consecutive knot pair. The knot values are given by

$$t_i = \frac{i-k}{n-k+1}(b-a) + a, \quad i = 1, \dots, n+k$$

int **gsl\_bspline\_init\_interp**(const *gsl\_vector* \*x, *gsl\_bspline\_workspace* \*w)

This function calculates a knot vector suitable for interpolating data at the  $n$  abscissa values provided in  $\mathbf{x}$ , which must be in ascending order. The knot vector for interpolation must satisfy the *Schoenberg-Whitney* conditions,

$$t_i < x_i < t_{i+k}, \quad i = 1, \dots, n$$

This function selects the knots according to

$$t_i = \begin{cases} x_1 & i = 1, \dots, k \\ \frac{1}{k-1} \sum_{j=i-k+1}^{i-1} x_j & i = k+1, \dots, n \\ x_n & i = n+1, \dots, n+k \end{cases}$$

which satisfy the Schoenberg-Whitney conditions, although this may not be the optimal knot vector for a given interpolation problem.

int **gsl\_bspline\_init**(const *gsl\_vector* \*t, *gsl\_bspline\_workspace* \*w)

This function sets the knot vector equal to the user-supplied vector  $\mathbf{t}$  which must have length  $n + k$  and be non-decreasing.

## 42.4 Properties of B-splines

size\_t **gsl\_bspline\_ncontrol**(const *gsl\_bspline\_workspace* \*w)

This function returns the number of B-spline control points given by  $n = \text{nbreak} + k - 2$ .

size\_t **gsl\_bspline\_order**(const *gsl\_bspline\_workspace* \*w)

This function returns the B-spline order  $k$ .

size\_t **gsl\_bspline\_nbreak**(const *gsl\_bspline\_workspace* \*w)

This function returns the number of breakpoints (interior knots) defined for the B-spline.

## 42.5 Evaluation of B-splines

int **gsl\_bspline\_calc**(const double x, const *gsl\_vector* \*c, double \*result, *gsl\_bspline\_workspace* \*w)

This function evaluates the B-spline

$$f(x) = \sum_{i=1}^n c_i B_i(x)$$

at the point  $x$  given the  $n$  control points  $c$ . The result  $f(x)$  is stored in *result*. If  $x$  lies outside the knot interval, a Taylor series approximation about the closest knot endpoint is used to extrapolate the spline.

int **gsl\_bspline\_vector\_calc**(const double x, const *gsl\_matrix* \*c, *gsl\_vector* \*result, *gsl\_bspline\_workspace* \*w)

This function evaluates the vector valued B-spline

$$\mathbf{f}(x) = \sum_{i=1}^n \mathbf{c}_i B_i(x)$$

at the point  $x$  given the  $n$  control vectors  $\mathbf{c}_i$ . The vector  $\mathbf{c}_i$  is stored in the  $i$ -th column of  $c$ . The result  $\mathbf{f}(x)$  is stored in *result*. The parameters  $c$  and *result* must have the same number of rows. If  $x$  lies outside the knot interval, a Taylor series approximation about the closest knot endpoint is used to extrapolate the spline.

int **gsl\_bspline\_basis**(const double x, *gsl\_vector* \*B, size\_t \*istart, *gsl\_bspline\_workspace* \*w)

This function evaluates all potentially nonzero B-spline basis functions at the position  $x$  and stores them in the vector  $B$  of length  $k$ , so that the  $i$ -th element is  $B_{(istart+i)}(x)$  for  $i = 0, 1, \dots, k - 1$ . The output *istart* is guaranteed to be in  $[0, n - k]$ . By returning only the nonzero basis functions, this function allows quantities involving linear combinations of the  $B_i(x)$  to be computed without unnecessary terms (such linear combinations occur, for example, when evaluating an interpolated function).

## 42.6 Evaluation of B-spline derivatives

int **gsl\_bspline\_calc\_deriv**(const double x, const *gsl\_vector* \*c, const size\_t nderiv, double \*result, *gsl\_bspline\_workspace* \*w)

This function evaluates the B-spline derivative

$$\frac{d^j}{dx^j} f(x) = \sum_{i=1}^n c_i \frac{d^j}{dx^j} B_i(x)$$



at the point  $x$  given the  $n$  control points  $c$ . The derivative order  $j$  is specified by `nderiv`. The result  $d^j f(x)/dx^j$  is stored in `result`. If  $x$  lies outside the knot interval, a Taylor series approximation about the closest knot endpoint is used to extrapolate the spline.

```
int gsl_bspline_vector_calc_deriv(const double x, const gsl_matrix *c, const size_t nderiv, gsl_vector *result,
                                gsl_bspline_workspace *w)
```

This function evaluates the vector valued B-spline derivative

$$\frac{d^j}{dx^j} \mathbf{f}(x) = \sum_{i=1}^n \mathbf{c}_i \frac{d^j}{dx^j} B_i(x)$$

at the point  $x$  given the  $n$  control vectors  $c_i$ . The vector  $c_i$  is stored in the  $i$ -th column of  $c$ . The result  $d^j \mathbf{f}(x)/dx^j$  is stored in `result`. The parameters  $c$  and `result` must have the same number of rows. The derivative order  $j$  is specified in `nderiv`. If  $x$  lies outside the knot interval, a Taylor series approximation about the closest knot endpoint is used to extrapolate the spline.

```
int gsl_bspline_basis_deriv(const double x, const size_t nderiv, gsl_matrix *dB, size_t *istart,
                            gsl_bspline_workspace *w)
```

This function evaluates all potentially nonzero B-spline basis function derivatives of orders 0 through `nderiv` (inclusive) at the position  $x$  and stores them in the matrix `dB`. The  $(i, j)$ -th element of `dB` is  $d^j B_{(istart+i)}(x)/dx^j$ . The last row of `dB` contains  $d^j B_{istart+k-1}(x)/dx^j$ . The matrix `dB` must be of size  $k$  by at least `nderiv` + 1. The output `istart` is guaranteed to be in  $[0, n - k]$ . Note that function evaluations are included as the zeroth order derivatives in `dB`. By returning only the nonzero basis functions, this function allows quantities involving linear combinations of the  $B_i(x)$  and their derivatives to be computed without unnecessary terms.

## 42.7 Evaluation of B-spline integrals

```
int gsl_bspline_calc_integ(const double a, const double b, const gsl_vector *c, double *result,
                           gsl_bspline_workspace *w)
```

This function computes the integral

$$\int_a^b f(x) dx = \sum_{i=1}^n c_i \int_a^b B_i(x) dx$$

using the coefficients stored in  $c$  and the integration limits  $(a, b)$ . The integral value is stored in `result` on output.

```
int gsl_bspline_basis_integ(const double a, const double b, gsl_vector *y, gsl_bspline_workspace *w)
```

This function computes the integral from  $a$  to  $b$  of each basis function  $B_i(x)$ ,

$$y_i = \int_a^b B_i(x) dx$$

storing the results in the vector  $y$ , of length `ncontrol`. The algorithm uses Gauss-Legendre quadrature on each knot interval.

## 42.8 Least Squares Fitting with B-splines

A common application is to fit a B-spline to a set of  $m$  data points  $(x_i, y_i)$  without requiring the spline to pass through each point, but rather the spline fits the data in a least square sense. In this case, the number of control points can be much less than  $m$  and the result is a smooth spline which minimizes the cost function,

$$\chi^2 = \sum_{i=1}^m w_i (y_i - f(x_i))^2 = \|y - Xc\|_W^2$$

where  $f(x) = \sum_{j=1}^n c_j B_{j,k}(x)$  is the B-spline defined above,  $n$  is the number of control points for the spline,  $W = \text{diag}(w)$ , and  $X_{ij} = B_{j,k}(x_i)$  is the least squares design matrix. The parameters  $w_i$  are optional weights which can be assigned to each data point  $(x_i, y_i)$ . Because each basis spline  $B_{j,k}(x)$  has local support ( $B_{j,k}(x) = 0$  for  $x \notin [t_j, \dots, t_{j+k}]$ ), the least squares design matrix  $X$  has only  $k$  nonzero entries per row. Therefore, the normal equations matrix  $X^T W X$  is *symmetric and banded*, with lower bandwidth  $k - 1$ . The GSL routines below use a banded Cholesky factorization to solve the normal equations system for the unknown coefficient vector,

$$c = (X^T W X)^{-1} X^T W y$$

The normal equations approach is often avoided in least squares problems, since if  $X$  is badly conditioned, then  $X^T W X$  is much worse conditioned, which can lead to loss of accuracy in the solution vector. However, a theorem of de Boor [de Boor (2001), XI(4)] states that the condition number of the B-spline basis is independent of the knot vector  $\mathbf{t}$  and bounded, depending only on  $k$ . Numerical evidence has shown that for reasonably small  $k$  the normal equations matrix is well conditioned, and a banded Cholesky factorization can be used safely.

```
int gsl_bspline_lssolve(const gsl_vector *x, const gsl_vector *y, gsl_vector *c, double *chisq,
                       gsl_bspline_workspace *w)
```

```
int gsl_bspline_wlssolve(const gsl_vector *x, const gsl_vector *y, const gsl_vector *wts, gsl_vector *c, double
                          *chisq, gsl_bspline_workspace *w)
```

These functions fit a B-spline to the input data  $(x, y)$  which must be the same length  $m$ . Optional weights may be given in the *wts* vector. The banded normal equations matrix is assembled and factored with a Cholesky factorization, and then solved, storing the coefficients in the vector  $c$ , which has length  $n$ . The cost function  $\chi^2$  is output in *chisq*. If the normal equations matrix is singular, the error code [GSL\\_EDOM](#) is returned. This could occur, for example, if there is a large gap in the data vectors so that no data points constrain a particular  $B_{i,k}(x)$  basis function.

On output,  $w \rightarrow \text{XTX}$  contains the Cholesky factor of the symmetric banded normal equations matrix, and this can be passed directly to the functions [gsl\\_bspline\\_covariance\(\)](#) and [gsl\\_bspline\\_rcond\(\)](#).

```
int gsl_bspline_lsnormal(const gsl_vector *x, const gsl_vector *y, const gsl_vector *wts, gsl_vector *XTy,
                          gsl_matrix *XTX, gsl_bspline_workspace *w)
```

This function builds the normal equations matrix  $X^T W X$  and right hand side vector  $X^T W y$ , which are stored in *XTX* and *XTy* on output. The diagonal weight matrix  $W$  is specified by the input vector *wts*, which may be set to NULL, in which case  $W = I$ . The inputs  $x$ ,  $y$ , and *wts* represent the data and weights to be fitted in a least-squares sense, and they all have length  $m$ . While the full normal equations matrix has dimension  $n$ -by- $n$ , it has a significant number of zeros, and so the output matrix *XTX* is stored in *banded symmetric format*. Therefore, its dimensions are  $n$ -by- $k$ . The vector *XTy* has length  $n$ . The output matrix *XTX* can be passed directly into a banded Cholesky factorization routine, such as [gsl\\_linalg\\_cholesky\\_band\\_decomp\(\)](#).

This function is normally called by [gsl\\_bspline\\_wlssolve\(\)](#) but is available for users with more specialized applications.

```
int gsl_bspline_lsnormalm(const gsl_vector *x, const gsl_matrix *Y, const gsl_vector *wts, gsl_matrix *XTY,
                           gsl_matrix *XTX, gsl_bspline_workspace *w)
```

This function builds the normal equations matrix  $X^T W X$  and right hand side vectors  $X^T W Y$ , which are stored in *XTX* and *XTY* on output. The diagonal weight matrix  $W$  is specified by the input vector *wts*, which may be

set to NULL, in which case  $W = I$ . The input  $Y$  is a matrix of the right hand side vectors, of size  $m$ -by- $n$ -rhs. The inputs  $x$  and  $wts$  represent the data and weights to be fitted in a least-squares sense, and they have length  $m$ . While the full normal equations matrix has dimension  $n$ -by- $n$ , it has a significant number of zeros, and so the output matrix  $XTX$  is stored in *banded symmetric format*. Therefore, its dimensions are  $n$ -by- $k$ . The matrix  $XTY$  has size  $n$ -by- $n$ -rhs. The output matrix  $XTX$  can be passed directly into a banded Cholesky factorization routine, such as `gsl_linalg_cholesky_band_decomp()`.

```
int gsl_bspline_residuals(const gsl_vector *x, const gsl_vector *y, const gsl_vector *c, gsl_vector *r,
                        gsl_bspline_workspace *w)
```

This function computes the residual vector  $r_i = y_i - f(x_i)$  by evaluating the spline  $f(x)$  using the coefficients  $c$ .

```
int gsl_bspline_covariance(const gsl_matrix *cholesky, gsl_matrix *cov, gsl_bspline_workspace *w)
```

This function computes the  $n$ -by- $n$  covariance matrix  $(X^T W X)^{-1}$  using the banded symmetric Cholesky decomposition stored in `cholesky`, which has dimensions  $n$ -by- $k$ . The output is stored in `cov`. The function `gsl_linalg_cholesky_band_decomp()` must be called on the normal equations matrix to compute the Cholesky factor prior to calling this function.

```
int gsl_bspline_err(const double x, const size_t nderiv, const gsl_matrix *cov, double *err,
                  gsl_bspline_workspace *w)
```

This function computes the standard deviation error of a spline (or its derivative),

$$f^{(q)}(x) = \sum_{j=1}^n c_j B_j^{(q)}(x)$$

at the point  $x$  using the covariance matrix `cov` calculated previously with `gsl_bspline_covariance()`. The derivative order is specified by `nderiv`. The error is stored in `err` on output, which is defined as,

$$\delta f = \sqrt{\mathbf{B}^{(q)}(x)^T C \mathbf{B}^{(q)}(x)}$$

where  $\mathbf{B}^{(q)}(x) = (B_1^{(q)}(x), \dots, B_n^{(q)}(x))$  is the vector of B-spline values evaluated at  $x$ , and  $C$  is the covariance matrix.

```
int gsl_bspline_rcond(const gsl_matrix *XTX, double *rcond, gsl_bspline_workspace *w)
```

This function estimates the reciprocal condition number (using the 1-norm) of the normal equations matrix  $X^T W X$ , using its Cholesky decomposition, which must be computed previously by calling `gsl_linalg_cholesky_band_decomp()`. The reciprocal condition number estimate, defined as  $1/(\|X^T W X\|_1 \cdot \|(X^T W X)^{-1}\|_1)$ , is stored in `rcond` on output.

### 42.8.1 Periodic Spline Fitting

Some applications require fitting a spline to a dataset which has an underlying periodicity associated with it. In these cases it is desirable to construct a spline which exhibits the same periodicity. A spline  $f(x)$  is called periodic on  $[a, b]$  if it satisfies the conditions

$$f^{(j)}(a) = f^{(j)}(b), \quad j = 0, \dots, k - 2$$

These conditions can be satisfied with the following constraints on the knot vector and the control points (Dierckx, 1982),

$$\begin{aligned} t_i &= t_{i+n-k+1} - P, & i &= 1, \dots, k - 1 \\ t_{n-k+1+i} &= t_i + P, & i &= k + 1, \dots, 2k - 1 \\ c_i &= c_{n-k+1+i}, & i &= 1, \dots, k - 1 \end{aligned}$$

where  $P = b - a$  is the period. The constraint on the control points  $c_i$  means that there are only  $n - k + 1$  independent control points, and then the remaining  $k - 1$  are determined. The independent control points  $c^* = \{c_1, \dots, c_{n-k+1}\}$  can be determined by solving the following least squares problem (Dierckx, 1982):

$$\min_{c^*} \|y - X^* c^*\|_W^2$$

where  $X_{ij}^* = B_{j,k}^*(x_i)$  and

$$B_{j,k}^*(x) = \begin{cases} B_{j,k}(x) + B_{n-k+j+1,k}(x), & j = 1, \dots, k - 1 \\ B_{j,k}(x), & j = k, \dots, n - k + 1 \end{cases}$$

```
int gsl_bspline_plssolve(const gsl_vector *x, const gsl_vector *y, gsl_vector *c, double *chisq,
                        gsl_bspline_workspace *w)
```

```
int gsl_bspline_pwlssolve(const gsl_vector *x, const gsl_vector *y, const gsl_vector *wts, gsl_vector *c,
                          double *chisq, gsl_bspline_workspace *w)
```

These functions fit a periodic B-spline to the input data  $(x, y)$  which must be the same length  $m$ . Optional weights may be given in the *wts* vector. The input vector  $x$  must be sorted in ascending order, in order to efficiently compute the  $QR$  decomposition of the least squares system. The algorithm computes the  $QR$  decomposition of the least squares matrix, one row at a time using Givens transformations, taking advantage of sparse structure due to the local properties of the B-spline basis. On output, the coefficients are stored in the vector  $c$ , which has length  $n$ . The cost function  $\chi^2 = \|y - Xc\|_W^2$  is output in *chisq*.

```
int gsl_bspline_plsqqr(const gsl_vector *x, const gsl_vector *y, const gsl_vector *wts, gsl_matrix *R, gsl_vector
                      *QTy, double *rnorm, gsl_bspline_workspace *w)
```

This function calculates the  $R$  factor and  $Q^T y$  vector for a periodic spline fitting problem on the input data  $(x, y)$  with optional weights *wts*. The input vector  $x$  must be sorted in ascending order. On output, the  $(n - k + 1)$ -by- $(n - k + 1)$   $R$  factor is stored in  $R$ , while the first  $n - k + 1$  elements of the vector  $Q^T y$  are stored in  $QTy$ . The residual norm  $\|y - X^* c^*\|_W$  is stored in *rnorm*.

## 42.9 Gram Matrix

Let  $\mathbf{B}(x) = (B_1(x), B_2(x), \dots, B_n(x))^T$  be the  $n$ -vector whose elements are the B-spline basis functions evaluated at  $x$ . Then the  $n$ -by- $n$  outer product matrix of the B-spline basis at  $x$  is defined as

$$A(x) = \mathbf{B}(x) \otimes \mathbf{B}(x) = \mathbf{B}(x)\mathbf{B}(x)^T, \quad A_{ij}(x) = B_i(x)B_j(x)$$

This can be generalized to higher order derivatives,

$$A_{ij}^{(q)}(x) = \left( \frac{d^q}{dx^q} B_i(x) \right) \left( \frac{d^q}{dx^q} B_j(x) \right)$$

Then, the  $n$ -by- $n$  Gram matrix of the B-spline basis is defined as

$$G_{ij}^{(q)} = \int_a^b A_{ij}^{(q)}(x) dx = \int_a^b \left( \frac{d^q}{dx^q} B_i(x) \right) \left( \frac{d^q}{dx^q} B_j(x) \right) dx$$

where the integral is usually taken over the full support of the B-splines, i.e. from  $\min(\mathbf{t})$  to  $\max(\mathbf{t})$ . However it can be computed over any interval  $[a, b]$ .

The B-spline outer product and Gram matrices are *symmetric and banded* with lower bandwidth  $k - 1$ . These matrices are positive semi-definite, and in the case of  $q = 0$ ,  $A^{(0)}$  and  $G^{(0)}$  are positive definite. These matrices have applications in least-squares fitting.

int **gsl\_bspline\_oprod**(const size\_t nderiv, const double x, *gsl\_matrix* \*A, *gsl\_bspline\_workspace* \*w)

This function calculates the outer product matrix  $A^{(q)}(x)$  at the location  $x$ , where the derivative order  $q$  is specified by *nderiv*. The matrix is stored in *A* in *symmetric banded* format, and so *A* has dimensions  $n$ -by- $k$ . Since the B-spline functions have local support, many elements of the outer product matrix could be zero.

int **gsl\_bspline\_gram**(const size\_t nderiv, *gsl\_matrix* \*G, *gsl\_bspline\_workspace* \*w)

This function calculates the positive semi-definite Gram matrix  $G_{ij}^{(q)}$ , where the derivative order  $q$  is specified by *nderiv*. The matrix is stored in *G* in *symmetric banded* format, and so *G* has dimensions  $n$ -by- $k$ . The integrals are computed over the full support of the B-spline basis.

The algorithm uses Gauss-Legendre quadrature on each knot interval. See algorithm 5.22 of L. Schumaker, “Spline Functions: Basic Theory”.

int **gsl\_bspline\_gram\_interval**(const double a, const double b, const size\_t nderiv, *gsl\_matrix* \*G, *gsl\_bspline\_workspace* \*w)

This function calculates the positive semi-definite Gram matrix  $G_{ij}^{(q)}$ , where the derivative order  $q$  is specified by *nderiv*. The matrix is stored in *G* in *symmetric banded* format, and so *G* has dimensions  $n$ -by- $k$ . The integrals are computed over the interval  $[a, b]$ .

The algorithm uses Gauss-Legendre quadrature on each knot interval. See algorithm 5.22 of L. Schumaker, “Spline Functions: Basic Theory”.

## 42.10 Interpolation with B-splines

Given a set of  $n$  data points,  $(x_i, y_i)$ , we can define an interpolating spline with  $n$  control points which passes through each data point,

$$f(x_i) = \sum_{j=1}^n c_j B_j(x_i) = y_i, \quad i = 1, \dots, n$$

This is a square system of equations which can be solved by inverting the  $n$ -by- $n$  *collocation matrix*, whose elements are  $B_j(x_i)$ . In order for the collocation matrix to be invertible, the  $x_i$  and spline knots must satisfy the Schoenberg-Whitney conditions, which are

$$t_i < x_i < t_{i+k}, \quad i = 1, \dots, n$$

If these conditions are met, then the collocation matrix is banded, with both the upper and lower bandwidths equal to  $k - 1$ . This property leads to savings in storage and computation when solving interpolation problems. To construct a knot vector which satisfies these conditions, see the function *gsl\_bspline\_init\_interp()*.

int **gsl\_bspline\_col\_interp**(const *gsl\_vector* \*x, *gsl\_matrix* \*XB, *gsl\_bspline\_workspace* \*w)

This function constructs the banded collocation matrix to interpolate the  $n$  abscissa values provided in *x*. The values in *x* must be sorted in ascending order, although this is not checked explicitly by the function. The collocation matrix is stored in *XB* on output in banded storage format suitable for use with the banded LU routines. Therefore, *XB* must have dimension  $n$ -by- $3(k - 1) + 1$ .

## 42.11 Projection onto the B-spline Basis

A function  $f(x)$  can be projected onto the subspace  $V = \text{span}(B_i)$  spanned by the B-splines as follows,

$$f(x) \approx \text{proj}_V(f) = \sum_j c_j B_j(x)$$

If  $f(x)$  is a piecewise polynomial of degree less than  $k$ , then  $f(x)$  will lie in  $V$  and  $f(x) = \text{proj}_V(f)$ . The expansion coefficients  $c_j$  can be calculated as follows,

$$\begin{aligned} f(x)B_i(x) &= \sum_j c_j B_j(x)B_i(x) \\ \underbrace{\int f(x)B_i(x)dx}_{y_i} &= \sum_j c_j \underbrace{\int B_i(x)B_j(x)dx}_{G_{ij}} \\ Gc &= y \end{aligned}$$

where  $G$  is the *B-spline Gram Matrix*. The above equation can be solved using a Cholesky factorization of  $G$ .

int **gsl\_bspline\_proj\_rhs**(const *gsl\_function* \*F, *gsl\_vector* \*y, *gsl\_bspline\_workspace* \*w)

This function computes the right hand side vector  $y_i = \int f(x)B_i(x)dx$  for projecting a function  $F$  onto the B-spline basis. The output is stored in  $y$  which must have size `ncontrol`. The integrals are computed using Gauss-Legendre quadrature, which will produce exact results if  $f(x)$  is a piecewise polynomial of degree less than  $k$ .

## 42.12 Greville abscissae

The Greville abscissae are defined to be the mean location of  $k - 1$  consecutive knots in the knot vector for each basis spline function of order  $k$ . With the first and last knots in the knot vector excluded, there are  $n$  Greville abscissae for any given B-spline basis. These values are often used in B-spline collocation applications and may also be called Marsden-Schoenberg points.

double **gsl\_bspline\_greville\_abcissa**(const size\_t i, *gsl\_bspline\_workspace* \*w)

Returns the location of the  $i$ -th Greville abscissa for the given B-spline basis. For the ill-defined case when  $k = 1$ , the implementation chooses to return breakpoint interval midpoints.

## 42.13 Examples

### 42.13.1 Example 1: Basis Splines and Uniform Knots

The following program computes and outputs linear, quadratic, cubic, and quartic basis splines using uniform knots on the interval  $[0, 1]$ . The knot vector is

$$\mathbf{t} = \left( \underbrace{0, \dots, 0}_k, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, \underbrace{1, \dots, 1}_k \right)$$

where the first and last knot are repeated  $k$  times, where  $k$  is the spline order. The output is shown in [Fig. 42.1](#).

This figure makes it clear that each B-spline has local support and spans  $k + 1$  knots. The B-splines near the interval endpoints appear to span less knots, but note that the endpoint knots have multiplicities of  $k$ .

The program is given below.

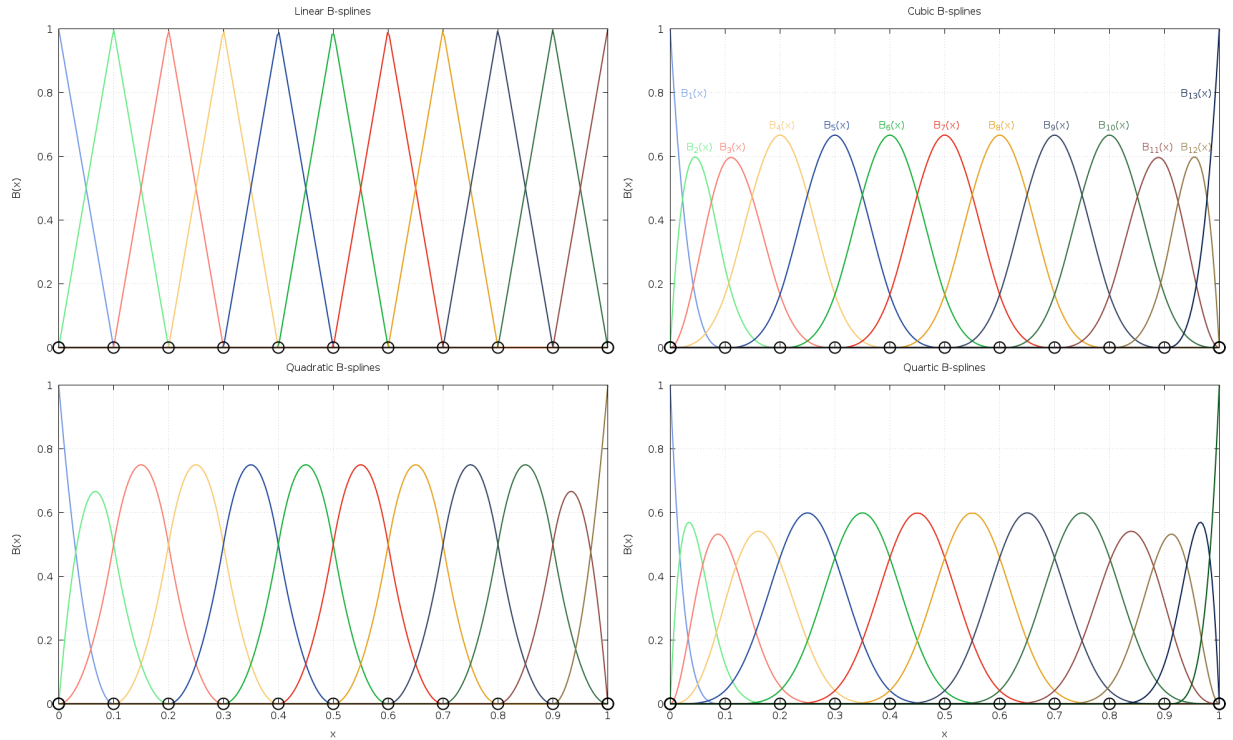


Fig. 42.1: Linear, quadratic, cubic, and quartic basis splines defined on the interval  $[0, 1]$  with uniform knots. Black circles denote the location of the knots.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_vector.h>

void
print_basis(FILE *fp_knots, FILE *fp_spline, const size_t order, const size_t nbreak)
{
    gsl_bspline_workspace *w = gsl_bspline_alloc(order, nbreak);
    const size_t p = gsl_bspline_ncontrol(w);
    const size_t n = 300;
    const double a = 0.0;
    const double b = 1.0;
    const double dx = (b - a) / (n - 1.0);
    gsl_vector *B = gsl_vector_alloc(p);
    size_t i, j;

    /* use uniform breakpoints on [a, b] */
    gsl_bspline_init_uniform(a, b, w);

    gsl_vector_fprintf(fp_knots, w->knots, "%f");
}
```

(continues on next page)



(continued from previous page)

```

for (i = 0; i < n; ++i)
{
    double xi = i * dx;

    gsl_bspline_eval_basis(xi, B, w);

    fprintf(fp_spline, "%f ", xi);

    for (j = 0; j < p; ++j)
        fprintf(fp_spline, "%f ", gsl_vector_get(B, j));

    fprintf(fp_spline, "\n");
}

fprintf(fp_knots, "\n\n");
fprintf(fp_spline, "\n\n");

gsl_vector_free(B);
gsl_bspline_free(w);
}

int
main (void)
{
    const size_t nbreak = 11; /* number of breakpoints */
    FILE *fp_knots = fopen("bspline1_knots.txt", "w");
    FILE *fp_spline = fopen("bspline1.txt", "w");

    print_basis(fp_knots, fp_spline, 2, nbreak); /* linear splines */
    print_basis(fp_knots, fp_spline, 3, nbreak); /* quadratic splines */
    print_basis(fp_knots, fp_spline, 4, nbreak); /* cubic splines */
    print_basis(fp_knots, fp_spline, 5, nbreak); /* quartic splines */

    fclose(fp_knots);
    fclose(fp_spline);

    return 0;
}

```

### 42.13.2 Example 2: Derivatives of Basis Splines

The following program computes and outputs cubic B-splines and their derivatives using 6 breakpoints and uniform knots on the interval  $[0, 1]$ . All derivatives up to order 3 are computed. The output is shown in [Fig. 42.2](#).

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

```

(continues on next page)



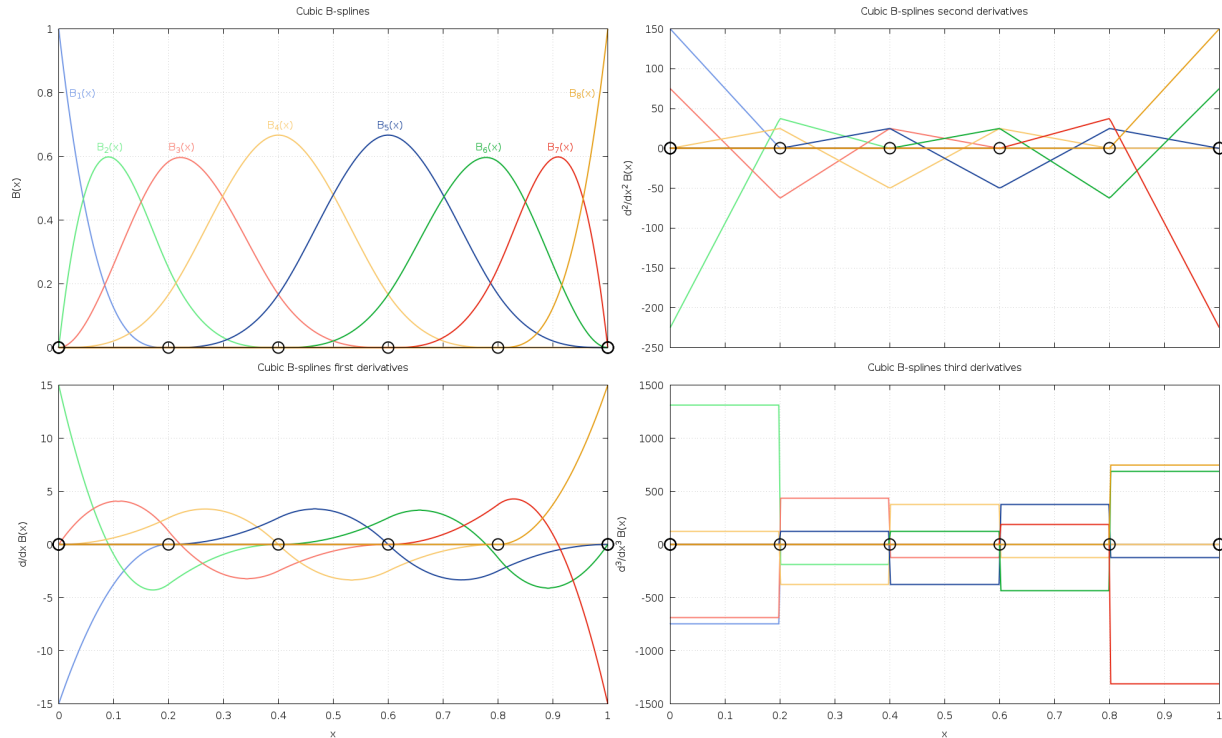


Fig. 42.2: Cubic B-splines and derivatives up to order 3. Black circles denote the location of the knots.

(continued from previous page)

```
#include <gsl/gsl_math.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_bspline.h>

int
main (void)
{
  const size_t nbreak = 6;
  const size_t spline_order = 4;
  gsl_bspline_workspace *w = gsl_bspline_alloc(spline_order, nbreak);
  const size_t p = gsl_bspline_ncontrol(w);
  const size_t n = 300;
  const double a = 0.0;
  const double b = 1.0;
  const double dx = (b - a) / (n - 1.0);
  gsl_matrix *dB = gsl_matrix_alloc(p, spline_order);
  size_t i, j, k;

  /* uniform breakpoints on [a, b] */
  gsl_bspline_init_uniform(a, b, w);

  /* output knot vector */
  gsl_vector_fprintf(stdout, w->knots, "%f");
  printf("\n\n");
}
```

(continues on next page)

(continued from previous page)

```

for (i = 0; i < spline_order; ++i)
{
    for (j = 0; j < n; ++j)
    {
        double xj = j * dx;

        gsl_bspline_eval_deriv_basis(xj, i, dB, w);

        printf("%f ", xj);

        for (k = 0; k < p; ++k)
            printf("%f ", gsl_matrix_get(dB, k, i));

        printf("\n");
    }

    printf("\n\n");
}

gsl_matrix_free(dB);
gsl_bspline_free(w);

return 0;
}

```

### 42.13.3 Example 3: Least squares and breakpoints

The following program computes a linear least squares fit to data using cubic B-splines with uniform breakpoints. The data is generated from the curve  $y(x) = \cos x \exp(-x/10)$  with Gaussian noise added. Splines are fitted with 10 and 40 breakpoints for comparison.

The program output is shown below:

```

$ ./a.out > bspline_lsbreak.txt
40 breakpoints: chisq/dof = 1.008999e+00
10 breakpoints: chisq/dof = 1.014761e+00

```

The data and fitted models are shown in [Fig. 42.3](#).

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_statistics.h>

int

```

(continues on next page)

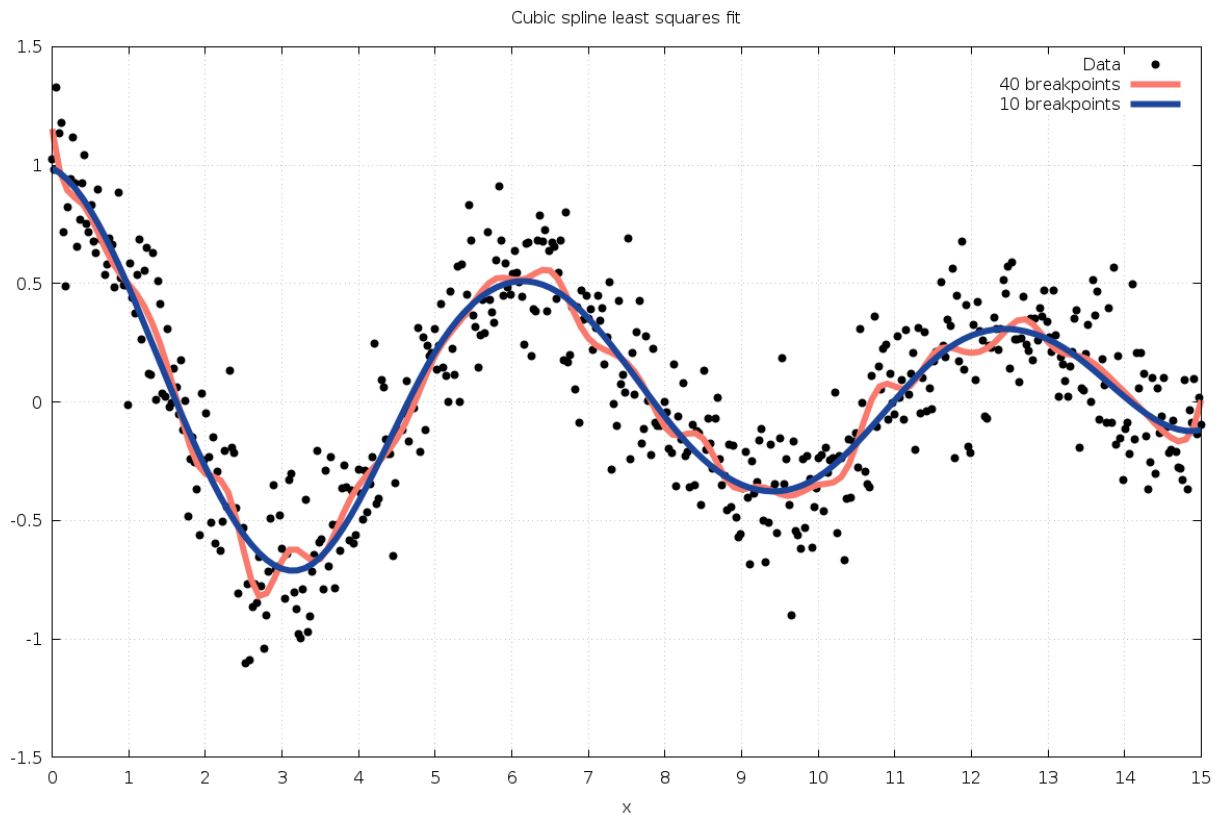


Fig. 42.3: Cubic B-spline least squares fit

(continued from previous page)

```

main (void)
{
  const size_t n = 500;      /* number of data points to fit */
  const size_t k = 4;       /* spline order */
  const double a = 0.0;     /* data interval [a,b] */
  const double b = 15.0;
  const double sigma = 0.2; /* noise */
  gsl_bspline_workspace *work1 = gsl_bspline_alloc(k, 40); /* 40 breakpoints */
  gsl_bspline_workspace *work2 = gsl_bspline_alloc(k, 10); /* 10 breakpoints */
  const size_t p1 = gsl_bspline_ncontrol(work1); /* number of control points.
↪ */
  const size_t p2 = gsl_bspline_ncontrol(work2); /* number of control points.
↪ */
  const size_t dof1 = n - p1; /* degrees of freedom */
  const size_t dof2 = n - p2; /* degrees of freedom */
  gsl_vector *c1 = gsl_vector_alloc(p1);
  gsl_vector *c2 = gsl_vector_alloc(p2);
  gsl_vector *x = gsl_vector_alloc(n);
  gsl_vector *y = gsl_vector_alloc(n);
  gsl_vector *w = gsl_vector_alloc(n);
  size_t i;
  gsl_rng *r;
  double chisq1, chisq2;

  gsl_rng_env_setup();
  r = gsl_rng_alloc(gsl_rng_default);

  /* this is the data to be fitted */
  for (i = 0; i < n; ++i)
  {
    double xi = (b - a) / (n - 1.0) * i + a;
    double yi = cos(xi) * exp(-0.1 * xi);
    double dyi = gsl_ran_gaussian(r, sigma);

    yi += dyi;

    gsl_vector_set(x, i, xi);
    gsl_vector_set(y, i, yi);
    gsl_vector_set(w, i, 1.0 / (sigma * sigma));

    printf("%f %f\n", xi, yi);
  }

  /* use uniform breakpoints on [a, b] */
  gsl_bspline_init_uniform(a, b, work1);
  gsl_bspline_init_uniform(a, b, work2);

  /* solve least squares problem */
  gsl_bspline_wlssolve(x, y, w, c1, &chisq1, work1);
  gsl_bspline_wlssolve(x, y, w, c2, &chisq2, work2);

  fprintf(stderr, "40 breakpoints: chisq/dof = %e\n", chisq1 / dof1);

```

(continues on next page)

(continued from previous page)

```

fprintf(stderr, "10 breakpoints: chisq/dof = %e\n", chisq2 / dof2);

printf("\n\n");

/* output the spline curves */
{
    double xi;

    for (xi = a; xi <= b; xi += 0.1)
        {
            double result1, result2;

            gsl_bspline_calc(xi, c1, &result1, work1);
            gsl_bspline_calc(xi, c2, &result2, work2);
            printf("%f %f %f\n", xi, result1, result2);
        }
}

gsl_rng_free(r);
gsl_vector_free(x);
gsl_vector_free(y);
gsl_vector_free(w);
gsl_vector_free(c1);
gsl_vector_free(c2);
gsl_bspline_free(work1);
gsl_bspline_free(work2);

return 0;
}

```

#### 42.13.4 Example 4: Least squares and spline order

The following program computes least squares fits to the same dataset in the previous example, using 10 uniform breakpoints and computing splines of order 1, 2, 3, 4, 5.

The data and fitted models are shown in Fig. 42.4.

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_statistics.h>

int
main (void)
{

```

(continues on next page)

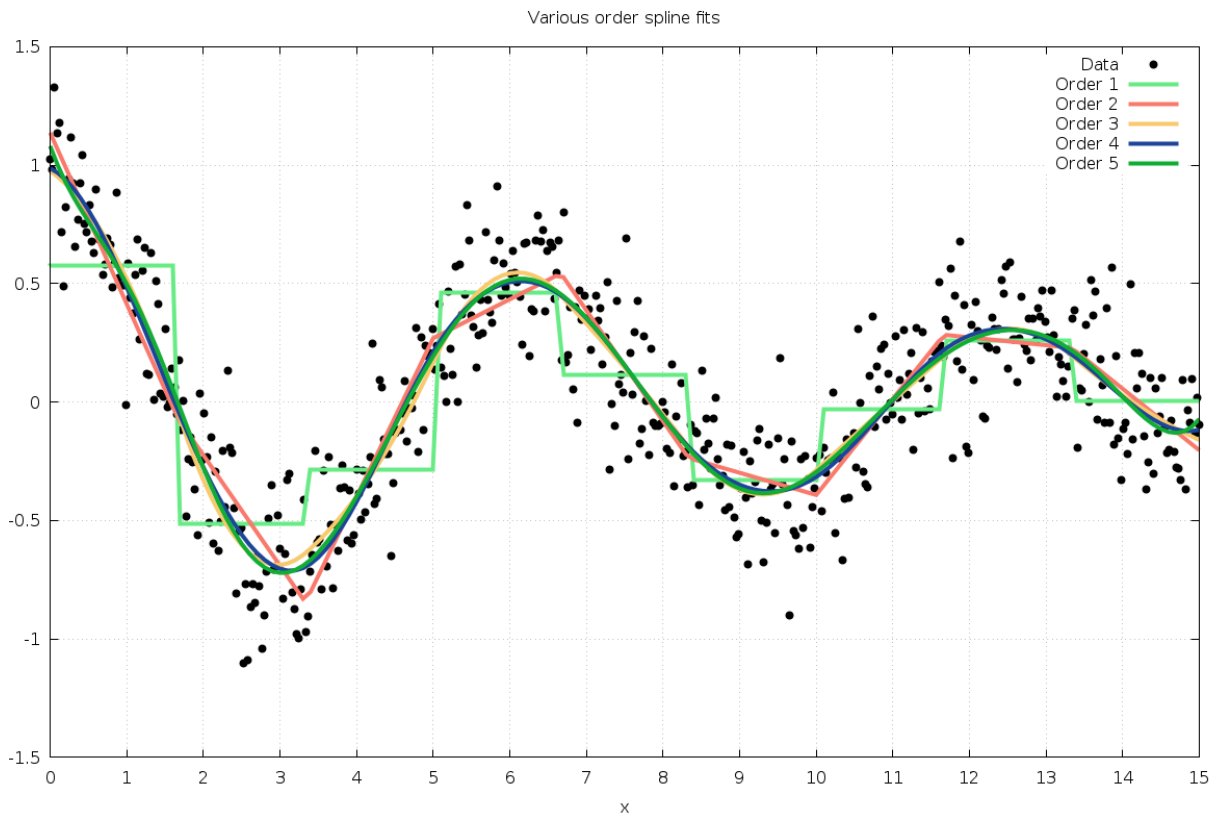


Fig. 42.4: B-spline least squares fit with varying spline orders

(continued from previous page)

```

const size_t n = 500;          /* number of data points to fit */
const double a = 0.0;        /* data interval [a,b] */
const double b = 15.0;
const size_t max_spline_order = 5; /* maximum spline order */
const size_t nbreak = 10;    /* number of breakpoints */
const double sigma = 0.2;    /* noise */
gsl_bspline_workspace **work = malloc(max_spline_order * sizeof(gsl_bspline_workspace_
↪*));
gsl_vector **c = malloc(max_spline_order * sizeof(gsl_vector *));
gsl_vector *x = gsl_vector_alloc(n);
gsl_vector *y = gsl_vector_alloc(n);
gsl_vector *w = gsl_vector_alloc(n);
size_t i;
gsl_rng *r;
double chisq;

gsl_rng_env_setup();
r = gsl_rng_alloc(gsl_rng_default);

/* this is the data to be fitted */
for (i = 0; i < n; ++i)
{
    double xi = (b - a) / (n - 1.0) * i + a;
    double yi = cos(xi) * exp(-0.1 * xi);
    double dyi = gsl_ran_gaussian(r, sigma);

    yi += dyi;

    gsl_vector_set(x, i, xi);
    gsl_vector_set(y, i, yi);
    gsl_vector_set(w, i, 1.0 / (sigma * sigma));

    printf("%f %f\n", xi, yi);
}

printf("\n\n");

for (i = 0; i < max_spline_order; ++i)
{
    /* allocate workspace for this spline order */
    work[i] = gsl_bspline_alloc(i + 1, nbreak);
    c[i] = gsl_vector_alloc(gsl_bspline_ncontrol(work[i]));

    /* use uniform breakpoints on [a, b] */
    gsl_bspline_init_uniform(a, b, work[i]);

    /* solve least squares problem */
    gsl_bspline_wlssolve(x, y, w, c[i], &chisq, work[i]);
}

/* output the spline curves */
{

```

(continues on next page)

(continued from previous page)

```

double xi;

for (xi = a; xi <= b; xi += 0.1)
{
    printf("%f ", xi);
    for (i = 0; i < max_spline_order; ++i)
    {
        double result;
        gsl_bspline_calc(xi, c[i], &result, work[i]);
        printf("%f ", result);
    }
    printf("\n");
}

for (i = 0; i < max_spline_order; ++i)
{
    gsl_vector_free(c[i]);
    gsl_bspline_free(work[i]);
}

gsl_rng_free(r);
gsl_vector_free(x);
gsl_vector_free(y);
gsl_vector_free(w);
free(work);
free(c);

return 0;
}

```

### 42.13.5 Example 5: Least squares, regularization and the Gram matrix

The following example program fits a cubic B-spline to data generated from the Gaussian

$$g(x) = e^{-x^2}$$

on the interval  $[-1.5, 1.5]$  with noise added. Two data gaps are deliberately introduced on  $[-1.1, -0.7]$  and  $[0.1, 0.55]$ , which makes the problem ill-posed. Fig. 42.5 shows that the ordinary least squares solution exhibits large errors in the data gap regions due to the ill-conditioned least squares matrix.

One way to correct the situation is to minimize the data misfit but also minimize the curvature of the spline averaged



over the full interval. We can define the average curvature using the second derivative of the spline as follows:

$$\begin{aligned}
 \text{curvature} &\approx \int \left| \frac{d^2}{dx^2} f(x) \right|^2 dx \\
 &= \int \left| \frac{d^2}{dx^2} \sum_{i=1}^n c_i B_i(x) \right|^2 dx \\
 &= \int \left| \sum_{i=1}^n c_i B_i''(x) \right|^2 dx \\
 &= \int |c^T B''(x)|^2 dx \\
 &= \int c^T B''(x) B''^T(x) c dx \\
 &= c^T \left( \int B''(x) B''^T(x) dx \right) c \\
 &= c^T G^{(2)} c
 \end{aligned}$$

where  $G^{(2)}$  is the *Gram matrix* of second derivatives of the B-splines. Therefore, our regularized least squares problem is

$$\min_c \|y - Xc\|_W^2 + \lambda^2 c^T G^{(2)} c$$

where the regularization parameter  $\lambda^2$  represents a tradeoff between minimizing the data misfit and minimizing the spline curvature. The solution of this least-squares problem is

$$c = \left( X^T W X + \lambda^2 G^{(2)} \right)^{-1} X^T W y$$

The example program below solves this problem without regularization ( $\lambda^2 = 0$ ) and with regularization ( $\lambda^2 = 0.1$ ) for comparison. We also plot the  $1\sigma$  confidence intervals for both solutions using the diagonal elements of the covariance matrix  $\text{Cov} = (X^T W X + \lambda^2 G^{(2)})^{-1}$ .

Fig. 42.5 shows the result of regularizing the spline over the full interval, using the command

```
./a.out > data.txt
```

Fig. 42.6 shows the result of regularizing the spline over the smaller interval  $[0, 0.6]$ , correcting only the second gap, using the command

```
./a.out 0 0.6 > data.txt
```

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_rng.h>

```

(continues on next page)

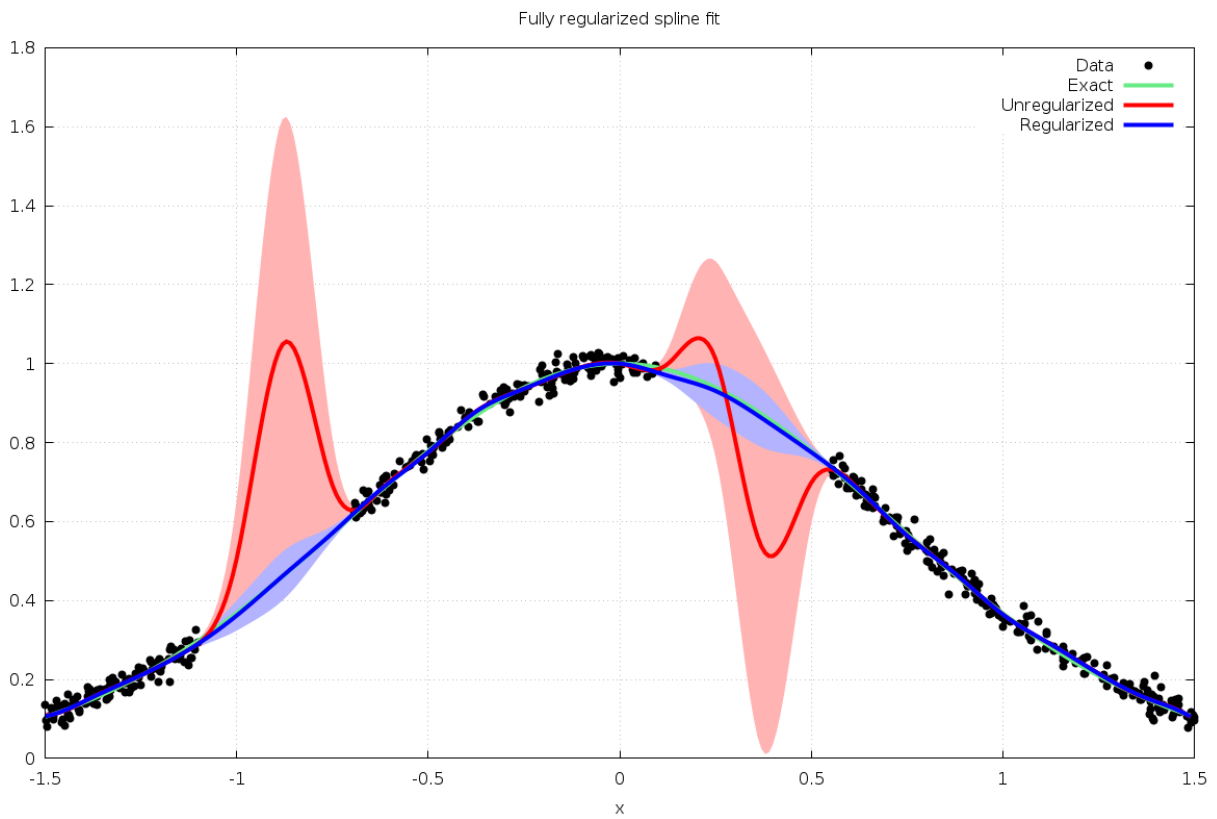


Fig. 42.5: B-spline least squares fit with exact (green), unregularized (red with  $1\sigma$  confidence intervals), and fully regularized (blue with  $1\sigma$  confidence intervals) solutions.

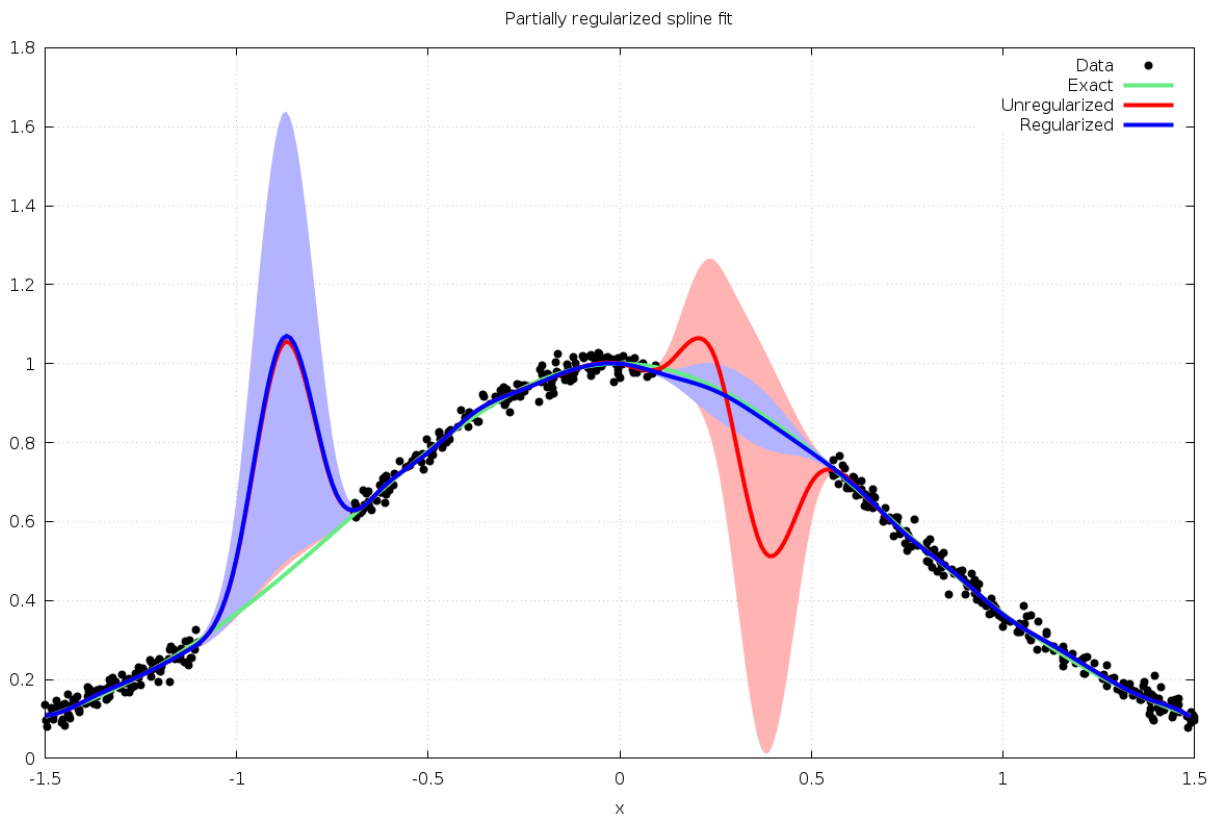


Fig. 42.6: B-spline least squares fit with exact (green), unregularized (red with  $1\sigma$  confidence intervals), and partially regularized (blue with  $1\sigma$  confidence intervals) solutions.

(continued from previous page)

```

#include <gsl/gsl_randist.h>

int
main (int argc, char * argv[])
{
    const size_t n = 500;                /* number of data points to
↳fit */
    const size_t k = 4;                  /* spline order */
    const double a = -1.5;               /* data interval [a,b] */
    const double b = 1.5;
    const double sigma = 0.02;           /* noise */
    const double lambda_sq = 0.1;        /* regularization parameter
↳*/
    gsl_bspline_workspace *work = gsl_bspline_alloc(k, 25); /* 25 breakpoints */
    const size_t p = gsl_bspline_ncontrol(work); /* number of control points
↳*/
    gsl_vector *c = gsl_vector_alloc(p); /* control points for
↳unregularized model */
    gsl_vector *creg = gsl_vector_alloc(p); /* control points for
↳regularized model */
    gsl_vector *x = gsl_vector_alloc(n); /* x data */
    gsl_vector *y = gsl_vector_alloc(n); /* y data */
    gsl_vector *w = gsl_vector_alloc(n); /* data weights */
    gsl_matrix *XTX = gsl_matrix_alloc(p, k); /* normal equations matrix */
    gsl_vector *XTy = gsl_vector_alloc(p); /* normal equations rhs */
    gsl_matrix *cov = gsl_matrix_alloc(p, p); /* covariance matrix */
    gsl_matrix *cov_reg = gsl_matrix_alloc(p, p); /* covariance matrix for
↳regularized model */
    gsl_matrix *G = gsl_matrix_alloc(p, k); /* regularization matrix */
    double reg_a = -1.0;                 /* regularization interval
↳[reg_a,reg_b] */
    double reg_b = -1.0;
    int reg_interval = 0;                 /* apply regularization on
↳smaller interval */
    size_t i;
    gsl_rng *r;

    if (argc > 2)
    {
        reg_a = atof(argv[1]);
        reg_b = atof(argv[2]);
        reg_interval = 1;
    }

    gsl_rng_env_setup();
    r = gsl_rng_alloc(gsl_rng_default);

    /* this is the data to be fitted */
    i = 0;
    while (i < n)
    {
        double ui = gsl_rng_uniform(r);

```

(continues on next page)

(continued from previous page)

```

double xi = (b - a) * ui + a;
double yi, dyi;

/* data gaps between [-1.1,-0.7] and [0.1,0.5] */
if ((xi >= -1.1 && xi <= -0.7) || (xi >= 0.1 && xi <= 0.55))
    continue;

dyi = gsl_ran_gaussian(r, sigma);
yi = exp(-xi*xi) + dyi;

gsl_vector_set(x, i, xi);
gsl_vector_set(y, i, yi);
gsl_vector_set(w, i, 1.0 / (sigma * sigma));

printf("%f %f\n", xi, yi);

++i;
}

printf("\n\n");

/* use uniform breakpoints on [a, b] */
gsl_bspline_init_uniform(a, b, work);

gsl_bspline_lsnormal(x, y, w, XTy, XTX, work); /* form normal equations matrix */
gsl_linalg_cholesky_band_decomp(XTX);          /* banded Cholesky decomposition */
gsl_linalg_cholesky_band_solve(XTX, XTy, c);   /* solve for unregularized solution */
gsl_bspline_covariance(XTX, cov, work);        /* compute covariance matrix */

/* compute regularization matrix */
if (reg_interval)
    gsl_bspline_gram_interval(reg_a, reg_b, 2, G, work);
else
    gsl_bspline_gram(2, G, work);

/* multiply by lambda^2 */
gsl_matrix_scale(G, lambda_sq);

gsl_bspline_lsnormal(x, y, w, XTy, XTX, work); /* form normal equations matrix */
gsl_matrix_add(XTX, G);                        /* add regularization term */
gsl_linalg_cholesky_band_decomp(XTX);          /* banded Cholesky decomposition */
gsl_linalg_cholesky_band_solve(XTX, XTy, creg); /* solve for regularized solution */
gsl_bspline_covariance(XTX, cov_reg, work);    /* compute covariance matrix */

/* output the spline curves */
{
    double xi;

    for (xi = a; xi <= b; xi += 0.01)
    {
        double result_unreg, result_reg;
        double err_unreg, err_reg;
    }
}

```

(continues on next page)

(continued from previous page)

```

    /* compute unregularized spline value and error at xi */
    gsl_bspline_calc(xi, c, &result_unreg, work);
    gsl_bspline_err(xi, 0, cov, &err_unreg, work);

    /* compute regularized spline value and error at xi */
    gsl_bspline_calc(xi, creg, &result_reg, work);
    gsl_bspline_err(xi, 0, cov_reg, &err_reg, work);

    printf("%f %e %e %e %e %e\n",
           xi,
           exp(-xi*xi),
           result_unreg,
           err_unreg,
           result_reg,
           err_reg);
}
}

gsl_rng_free(r);
gsl_vector_free(x);
gsl_vector_free(y);
gsl_vector_free(w);
gsl_vector_free(c);
gsl_vector_free(creg);
gsl_bspline_free(work);
gsl_matrix_free(XTX);
gsl_vector_free(XTy);
gsl_matrix_free(cov);
gsl_matrix_free(cov_reg);
gsl_matrix_free(G);

return 0;
}

```

### 42.13.6 Example 6: Least squares and endpoint regularization

Data fitting with splines can often produce undesired effects at the spline endpoints, where they are less constrained by data. As an example, we fit order  $k = 10$  splines to the Runge function,

$$g(x) = \frac{1}{1 + 25x^2}$$

with 20 uniform breakpoints on the interval  $[a, b] = [-1, 1]$ . Fig. 42.7 shows that the fitted spline (red) exhibits a sharp upward feature at the left endpoint. This is due to the lack of data constraining this part of the spline, and would likely result in errors if the user wanted to extrapolate the spline outside the fit interval. One method to correct this is to apply regularization to the spline at the endpoints by minimizing one or more of its derivatives. In this example, we minimize

the first derivative of the spline at both endpoints,  $|f'(a)|^2$  and  $|f'(b)|^2$ . Note that

$$\begin{aligned} \left| \frac{d}{dx} f(x) \right|^2 &= \left| \frac{d}{dx} \sum_{i=1}^n c_i B_i(x) \right|^2 \\ &= |c^T B'(x)|^2 \\ &= c^T B'(x) B'^T(x) c \\ &= c^T A^{(1)}(x) c \end{aligned}$$

where  $A^{(1)}$  is the *outer product matrix* of first derivatives. Therefore, our regularized least squares problem is

$$\min_c \|y - Xc\|_W^2 + \lambda^2 c^T A^{(1)}(a) c + \lambda^2 c^T A^{(1)}(b) c$$

where we have chosen to use the same regularization parameter  $\lambda^2$  to regularize both ends of the spline. In the example program below, we compute the unregularized solution ( $\lambda^2 = 0$ ) and regularized solution ( $\lambda^2 = 10$ ). The program outputs the first derivative of the spline at both endpoints:

```
unregularized endpoint deriv 1: [ -1.081170e+01, -2.963725e+00]
regularized endpoint deriv 1: [ -2.735857e-02, -5.371758e-03]
```

showing that the regularization has forced the first derivative smaller at both endpoints. The results are shown in Fig. 42.7.

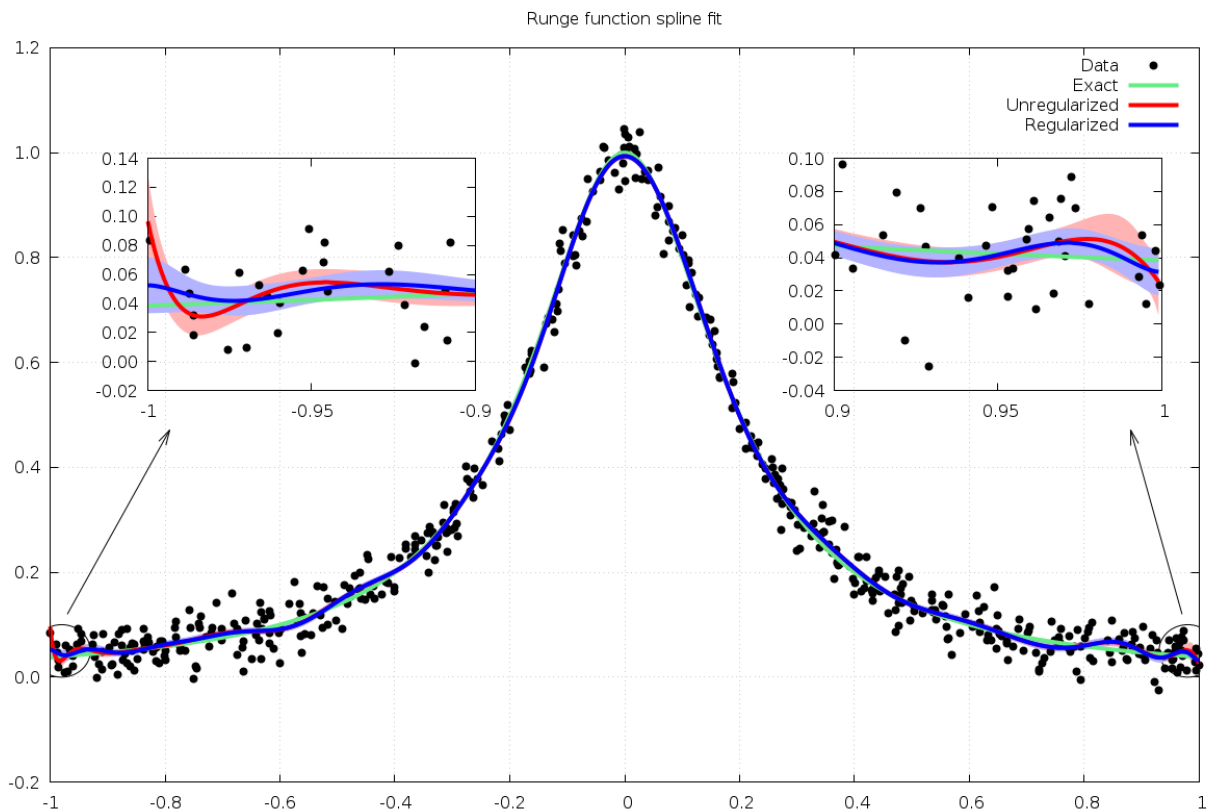


Fig. 42.7: Runge function B-spline least squares fit with exact (green), unregularized (red) and regularized (blue) solutions. The zoomed inset views show the spline endpoints with  $1\sigma$  confidence intervals.

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

double
f(double x)
{
    return (1.0 / (1.0 + 25.0*x*x));
}

int
main (void)
{
    const size_t n = 500;                /* number of data points to
    ↪fit */
    const size_t k = 10;                 /* spline order */
    const double a = -1.0;               /* data interval [a,b] */
    const double b = 1.0;
    const double sigma = 0.03;           /* noise */
    const double lambda_sq = 10.0;       /* regularization parameter
    ↪*/
    const size_t nderiv = 1;             /* derivative order to
    ↪regularize at endpoints */
    gsl_bspline_workspace *work = gsl_bspline_alloc(k, 20); /* 20 breakpoints */
    const size_t p = gsl_bspline_ncontrol(work); /* number of control points
    ↪*/
    gsl_vector *c = gsl_vector_alloc(p); /* control points for
    ↪unregularized model */
    gsl_vector *creg = gsl_vector_alloc(p); /* control points for
    ↪regularized model */
    gsl_vector *x = gsl_vector_alloc(n); /* x data */
    gsl_vector *y = gsl_vector_alloc(n); /* y data */
    gsl_vector *w = gsl_vector_alloc(n); /* data weights */
    gsl_matrix *XTX = gsl_matrix_alloc(p, k); /* normal equations matrix */
    gsl_vector *XTy = gsl_vector_alloc(p); /* normal equations rhs */
    gsl_matrix *cov = gsl_matrix_alloc(p, p); /* covariance matrix */
    gsl_matrix *cov_reg = gsl_matrix_alloc(p, p); /* covariance matrix for
    ↪regularized model */
    gsl_matrix *A1 = gsl_matrix_alloc(p, k); /* left regularization
    ↪matrix */
    gsl_matrix *A2 = gsl_matrix_alloc(p, k); /* right regularization
    ↪matrix */
    size_t i;
    gsl_rng *r;

```

(continues on next page)



(continued from previous page)

```

gsl_rng_env_setup();
r = gsl_rng_alloc(gsl_rng_default);

/* this is the data to be fitted */
i = 0;
while (i < n)
{
    double ui = gsl_rng_uniform(r);
    double xi = (b - a) * ui + a;
    double yi, dyi;

    dyi = gsl_ran_gaussian(r, sigma);
    yi = f(xi) + dyi;

    gsl_vector_set(x, i, xi);
    gsl_vector_set(y, i, yi);
    gsl_vector_set(w, i, 1.0 / (sigma * sigma));

    printf("%f %f\n", xi, yi);

    ++i;
}

printf("\n\n");

/* use uniform breakpoints on [a, b] */
gsl_bspline_init_uniform(a, b, work);

gsl_bspline_lsnormal(x, y, w, XTy, XTX, work); /* form normal equations matrix */
gsl_linalg_cholesky_band_decomp(XTX);          /* banded Cholesky decomposition */
gsl_linalg_cholesky_band_solve(XTX, XTy, c);   /* solve for unregularized solution */
gsl_bspline_covariance(XTX, cov, work);        /* compute covariance matrix */

/* compute regularization matrices */
gsl_bspline_oprod(nderiv, a, A1, work);
gsl_bspline_oprod(nderiv, b, A2, work);

/* multiply by lambda^2 */
gsl_matrix_scale(A1, lambda_sq);
gsl_matrix_scale(A2, lambda_sq);

gsl_bspline_lsnormal(x, y, w, XTy, XTX, work); /* form normal equations matrix */
gsl_matrix_add(XTX, A1);                       /* add regularization terms */
gsl_matrix_add(XTX, A2);
gsl_linalg_cholesky_band_decomp(XTX);          /* banded Cholesky decomposition */
gsl_linalg_cholesky_band_solve(XTX, XTy, creg); /* solve for regularized solution */
gsl_bspline_covariance(XTX, cov_reg, work);     /* compute covariance matrix */

/* output the spline curves */
{
    double xi;

```

(continues on next page)

(continued from previous page)

```

for (xi = a; xi <= b; xi += 0.001)
{
    double result_unreg, result_reg;
    double err_unreg, err_reg;

    /* compute unregularized spline value and error at xi */
    gsl_bspline_calc(xi, c, &result_unreg, work);
    gsl_bspline_err(xi, 0, cov, &err_unreg, work);

    /* compute regularized spline value and error at xi */
    gsl_bspline_calc(xi, creg, &result_reg, work);
    gsl_bspline_err(xi, 0, cov_reg, &err_reg, work);

    printf("%f %e %e %e %e %e\n",
           xi,
           f(xi),
           result_unreg,
           err_unreg,
           result_reg,
           err_reg);
}
}

{
    double result0, result1;

    gsl_bspline_calc_deriv(a, c, nderiv, &result0, work);
    gsl_bspline_calc_deriv(b, c, nderiv, &result1, work);
    fprintf(stderr, "unregularized endpoint deriv %zu: [%14.6e, %14.6e]\n", nderiv,
↳result0, result1);

    gsl_bspline_calc_deriv(a, creg, nderiv, &result0, work);
    gsl_bspline_calc_deriv(b, creg, nderiv, &result1, work);
    fprintf(stderr, " regularized endpoint deriv %zu: [%14.6e, %14.6e]\n", nderiv,
↳result0, result1);
}

gsl_rng_free(r);
gsl_vector_free(x);
gsl_vector_free(y);
gsl_vector_free(w);
gsl_vector_free(c);
gsl_vector_free(creg);
gsl_bspline_free(work);
gsl_matrix_free(XTX);
gsl_vector_free(XTy);
gsl_matrix_free(cov);
gsl_matrix_free(cov_reg);
gsl_matrix_free(A1);
gsl_matrix_free(A2);

return 0;

```

(continues on next page)

(continued from previous page)

}

### 42.13.7 Example 7: Least squares and periodic splines

The following example program fits B-spline curves to a dataset which is periodic. Both a non-periodic and periodic spline are fitted to the data for comparison. The data is derived from the function on  $[0, 2\pi]$ ,

$$f(x) = \sin x - \cos 2x$$

with Gaussian noise added. The program fits order  $k = 6$  splines to the dataset.

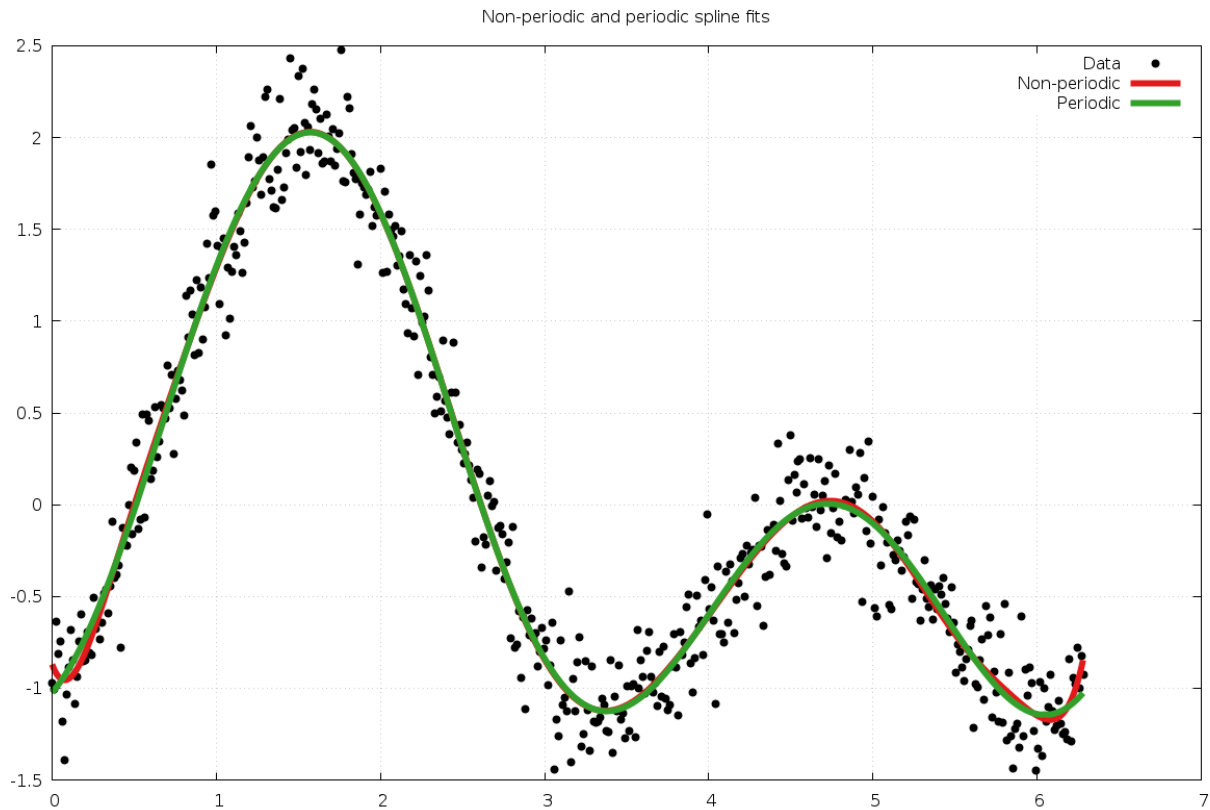


Fig. 42.8: Data (black) with non-periodic spline fit (red) and periodic spline fit (green).

The program additionally computes the derivatives up to order 5 at the endpoints  $x = 0$  and  $x = 2\pi$ :

```
=== Non-periodic spline endpoint derivatives ===
deriv 0: [ -8.697939e-01, -8.328553e-01]
deriv 1: [ -2.423132e+00,  4.277439e+00]
deriv 2: [  3.904362e+01,  3.710592e+01]
deriv 3: [ -2.096142e+02,  2.036125e+02]
deriv 4: [  7.240121e+02,  7.384888e+02]
deriv 5: [ -1.230036e+03,  1.294264e+03]
=== Periodic spline endpoint derivatives ===
deriv 0: [ -1.022798e+00, -1.022798e+00]
```

(continues on next page)

(continued from previous page)

```

deriv 1: [ 1.041910e+00, 1.041910e+00]
deriv 2: [ 4.466384e+00, 4.466384e+00]
deriv 3: [-1.356654e+00, -1.356654e+00]
deriv 4: [-2.751544e+01, -2.751544e+01]
deriv 5: [ 5.055419e+01, 2.674261e+01]

```

We can see that the periodic fit matches derivative values up to order 4, with the 5-th derivative being non-continuous.

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_statistics.h>

int
main (void)
{
    const size_t n = 500;          /* number of data points to fit */
    const double a = 0.0;         /* data interval [a,b] */
    const double b = 2.0 * M_PI;
    const size_t spline_order = 6; /* spline order */
    const size_t ncontrol = 15;   /* number of control points */
    const double sigma = 0.2;     /* noise */
    gsl_bspline_workspace *w = gsl_bspline_alloc_ncontrol(spline_order, ncontrol);
    gsl_bspline_workspace *wper = gsl_bspline_alloc_ncontrol(spline_order, ncontrol);
    gsl_vector *c = gsl_vector_alloc(ncontrol); /* non-periodic coefficients */
    gsl_vector *cper = gsl_vector_alloc(ncontrol); /* periodic coefficients */
    gsl_vector *x = gsl_vector_alloc(n);
    gsl_vector *y = gsl_vector_alloc(n);
    gsl_vector *wts = gsl_vector_alloc(n);
    size_t i;
    gsl_rng *r;
    double chisq;

    gsl_rng_env_setup();
    r = gsl_rng_alloc(gsl_rng_default);

    /* this is the data to be fitted */
    for (i = 0; i < n; ++i)
    {
        double xi = (b - a) / (n - 1.0) * i + a;
        double yi = sin(xi) - cos(2.0 * xi);
        double dyi = gsl_ran_gaussian(r, sigma);

        yi += dyi;

        gsl_vector_set(x, i, xi);

```

(continues on next page)

(continued from previous page)

```

    gsl_vector_set(y, i, yi);
    gsl_vector_set(wts, i, 1.0 / (sigma * sigma));

    printf("%f %f\n", xi, yi);
}

printf("\n\n");

/* use uniform non-periodic knots on [a, b] */
gsl_bspline_init_uniform(a, b, w);

/* solve least squares problem for non-periodic spline */
gsl_bspline_wlssolve(x, y, wts, c, &chisq, w);

/* use periodic knots on [a, b] */
gsl_bspline_init_periodic(a, b, wper);

/* solve least squares problem for periodic spline */
gsl_bspline_pwlssolve(x, y, wts, cper, &chisq, wper);

/* output the spline curves */
{
    double xi;

    for (xi = a; xi <= b; xi += 0.01)
    {
        double result, result_per;
        gsl_bspline_calc(xi, c, &result, w);
        gsl_bspline_calc(xi, cper, &result_per, wper);
        printf("%f %f %f\n", xi, result, result_per);
    }
}

fprintf(stderr, "=== Non-periodic spline endpoint derivatives ===\n");

for (i = 0; i < spline_order; ++i)
{
    double result0, result1;

    gsl_bspline_calc_deriv(a, c, i, &result0, w);
    gsl_bspline_calc_deriv(b, c, i, &result1, w);

    fprintf(stderr, "deriv %zu: [%14.6e, %14.6e]\n", i, result0, result1);
}

fprintf(stderr, "=== Periodic spline endpoint derivatives ===\n");

for (i = 0; i < spline_order; ++i)
{
    double result0, result1;

    gsl_bspline_calc_deriv(a, cper, i, &result0, wper);

```

(continues on next page)

(continued from previous page)

```

    gsl_bspline_calc_deriv(b, cper, i, &result1, wper);

    fprintf(stderr, "deriv %zu: [%14.6e, %14.6e]\n", i, result0, result1);
}

gsl_vector_free(c);
gsl_vector_free(cper);
gsl_bspline_free(w);
gsl_bspline_free(wper);
gsl_rng_free(r);
gsl_vector_free(x);
gsl_vector_free(y);
gsl_vector_free(wts);

return 0;
}

```

### 42.13.8 Example 8: Projecting onto the B-spline basis

The following example program projects the function

$$f(x) = 3x^3 - 2x^2 - 7x$$

onto a cubic B-spline basis with uniform knots on the interval  $[-2, 2]$ . Because the function is cubic, it can be exactly decomposed in the B-spline basis.

The program is given below.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>

/* function to be projected */
double
f(double x, void * params)
{
    (void) params;
    return (x * (-7.0 + x*(-2.0 + 3.0*x)));
}

int
main (void)
{
    const size_t k = 4;                /* spline order */
    const double a = -2.0;             /* spline interval [a,b] */
    const double b = 2.0;

```

(continues on next page)

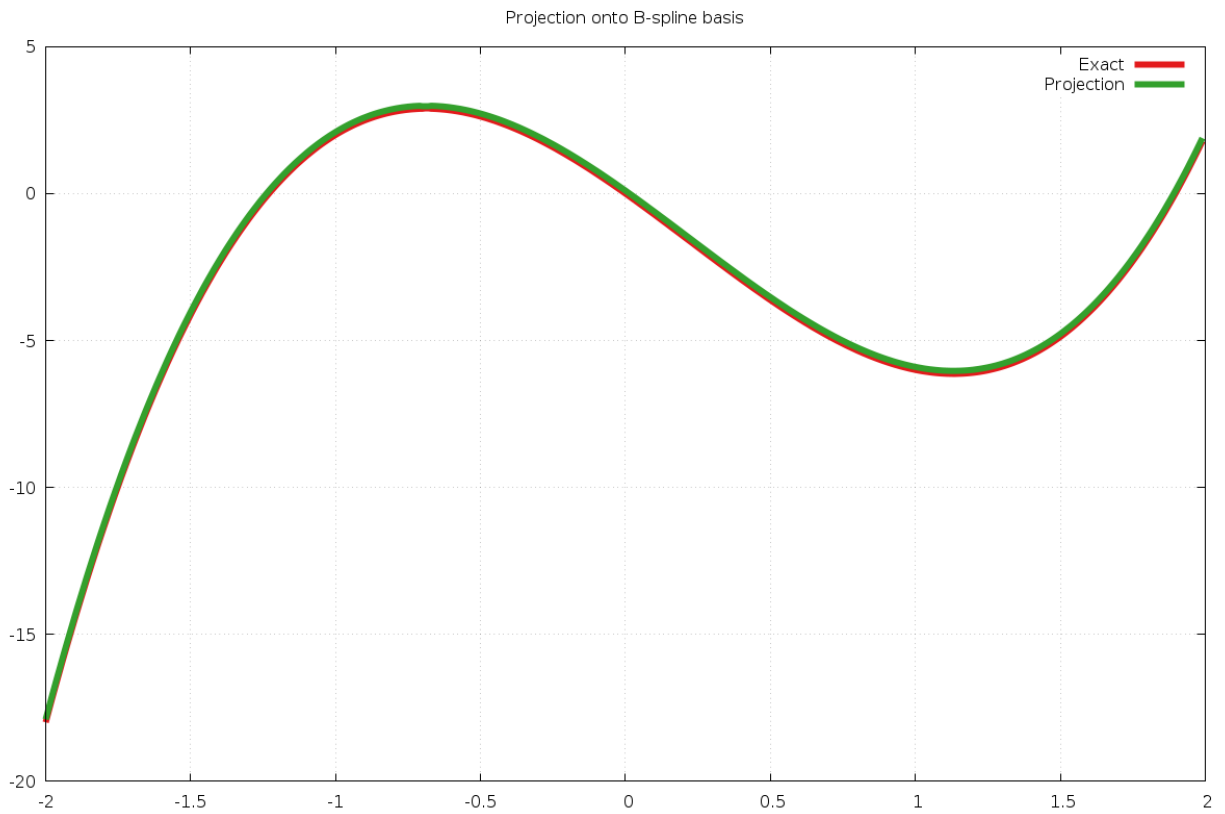


Fig. 42.9: Exact function (red) and spline projection (green). Small offset deliberately added to green curve.

(continued from previous page)

```
gsl_bspline_workspace *work = gsl_bspline_alloc(k, 10); /* 10 breakpoints */
const size_t n = gsl_bspline_ncontrol(work);          /* number of control points.
↪ */
gsl_vector *c = gsl_vector_alloc(n);                 /* control points for spline.
↪ */
gsl_vector *y = gsl_vector_alloc(n);                 /* rhs vector */
gsl_matrix *G = gsl_matrix_alloc(n, k);              /* Gram matrix */
gsl_function F;

F.function = f;
F.params = NULL;

/* use uniform breakpoints on [a, b] */
gsl_bspline_init_uniform(a, b, work);

/* compute Gram matrix */
gsl_bspline_gram(0, G, work);

/* construct rhs vector */
gsl_bspline_proj_rhs(&F, y, work);

/* solve system */
gsl_linalg_cholesky_band_decomp(G);
gsl_linalg_cholesky_band_solve(G, y, c);

/* output the result */
{
    double x;

    for (x = a; x <= b; x += 0.01)
    {
        double result;
        gsl_bspline_calc(x, c, &result, work);
        printf("%f %f %f\n", x, GSL_FN_EVAL(&F, x), result);
    }
}

gsl_vector_free(y);
gsl_vector_free(c);
gsl_matrix_free(G);
gsl_bspline_free(work);

return 0;
}
```



### 42.13.9 Example 9: Interpolation

The following example program computes an interpolating cubic B-spline for a set of data. It constructs the collocation matrix for the dataset and factors it with a banded LU decomposition. The results are shown in Fig. 42.10.

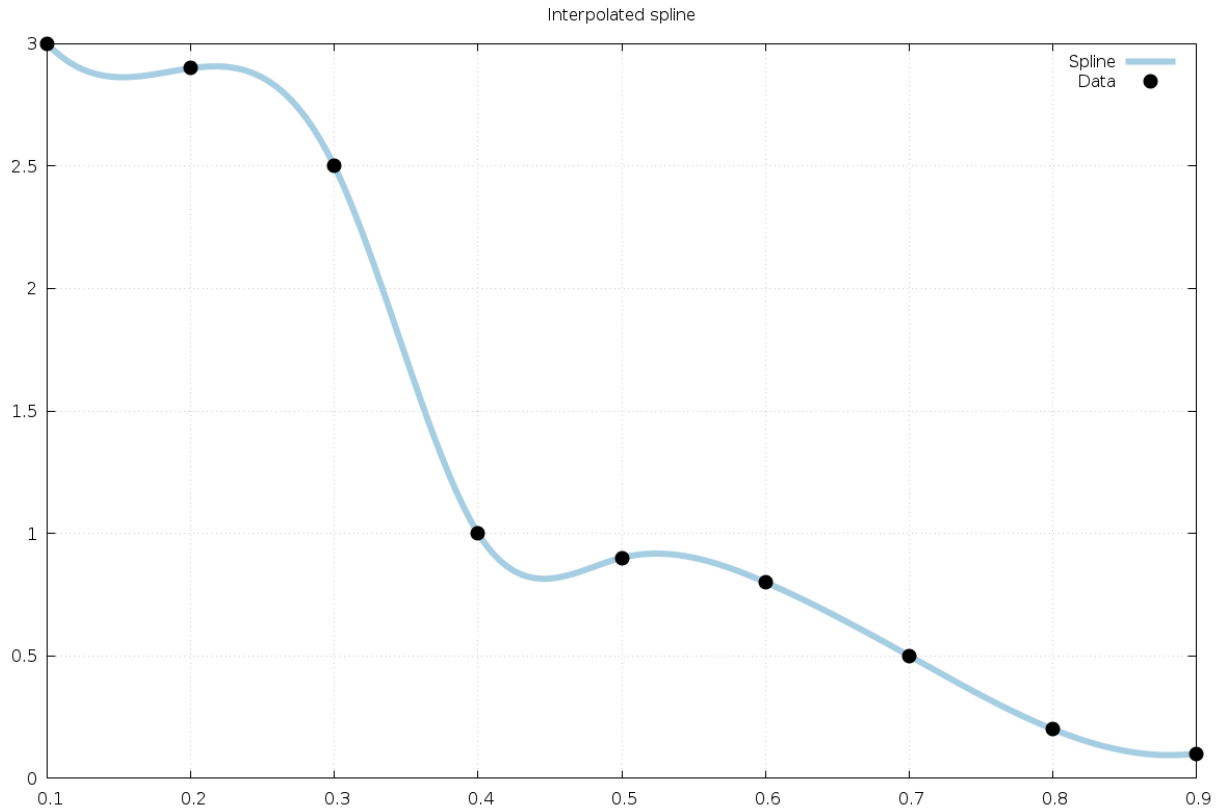


Fig. 42.10: Data points are shown in black and interpolated spline in light blue.

The program is given below.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_bspline.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>

int
main (void)
{
    const size_t n = 9;                /* number of data to interpolate */
    const double x_data[] = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 };
    const double y_data[] = { 3.0, 2.9, 2.5, 1.0, 0.9, 0.8, 0.5, 0.2, 0.1 };
    gsl_vector_const_view xv = gsl_vector_const_view_array(x_data, n);
```

(continues on next page)

(continued from previous page)

```
gsl_vector_const_view yv = gsl_vector_const_view_array(y_data, n);
const size_t k = 4; /* spline order */
gsl_vector *c = gsl_vector_alloc(n); /* control points for spline */
gsl_bspline_workspace *work = gsl_bspline_alloc_ncontrol(k, n);
gsl_matrix *XB = gsl_matrix_alloc(n, 3*(k-1) + 1); /* banded collocation matrix */
gsl_vector_uint *ipiv = gsl_vector_uint_alloc(n);
double t;
size_t i;

/* initialize knots for interpolation */
gsl_bspline_init_interp(&xv.vector, work);

/* compute collocation matrix for interpolation */
gsl_bspline_col_interp(&xv.vector, XB, work);

/* solve linear system with banded LU */
gsl_linalg_LU_band_decomp(n, k - 1, k - 1, XB, ipiv);
gsl_linalg_LU_band_solve(k - 1, k - 1, XB, ipiv, &yv.vector, c);

/* output the data */
for (i = 0; i < n; ++i)
{
    double xi = x_data[i];
    double yi = y_data[i];
    printf("%f %f\n", xi, yi);
}

printf("\n\n");

/* output the spline */
for (t = x_data[0]; t <= x_data[n-1]; t += 0.005)
{
    double result;
    gsl_bspline_calc(t, c, &result, work);
    printf("%f %f\n", t, result);
}

gsl_vector_free(c);
gsl_bspline_free(work);
gsl_vector_uint_free(ipiv);
gsl_matrix_free(XB);

return 0;
}
```

## 42.14 References and Further Reading

Further information on the algorithms described in this section can be found in the following references,

- C. de Boor, *A Practical Guide to Splines* (2001), Springer-Verlag, ISBN 0-387-95366-3.
- L. L. Schumaker, *Spline Functions: Basic Theory*, 3rd edition, Cambridge University Press, 2007.
- P. Dierckx, *Algorithms for smoothing data with periodic and parametric splines*, *Computer Graphics and Image Processing*, 20, 1982.
- P. Dierckx, *Curve and surface fitting with splines*, Oxford University Press, 1995.

Further information of Greville abscissae and B-spline collocation can be found in the following paper,

- Richard W. Johnson, Higher order B-spline collocation at the Greville abscissae. *Applied Numerical Mathematics*. vol.: 52, 2005, 63–75.

A large collection of B-spline routines is available in the PPPACK library available at <http://www.netlib.org/pppack>, which is also part of SLATEC.



## SPARSE MATRICES

This chapter describes functions for the construction and manipulation of sparse matrices, matrices which are populated primarily with zeros and contain only a few non-zero elements. Sparse matrices often appear in the solution of partial differential equations. It is beneficial to use specialized data structures and algorithms for storing and working with sparse matrices, since dense matrix algorithms and structures can be prohibitively slow and use huge amounts of memory when applied to sparse matrices.

The header file `gsl_spmatrix.h` contains the prototypes for the sparse matrix functions and related declarations.

### 43.1 Data types

All the functions are available for each of the standard data-types. The versions for double have the prefix `gsl_spmatrix`, Similarly the versions for single-precision float arrays have the prefix `gsl_spmatrix_float`. The full list of available types is given below,

Prefix	Type
<code>gsl_spmatrix</code>	double
<code>gsl_spmatrix_float</code>	float
<code>gsl_spmatrix_long_double</code>	long double
<code>gsl_spmatrix_int</code>	int
<code>gsl_spmatrix_uint</code>	unsigned int
<code>gsl_spmatrix_long</code>	long
<code>gsl_spmatrix_ulong</code>	unsigned long
<code>gsl_spmatrix_short</code>	short
<code>gsl_spmatrix_ushort</code>	unsigned short
<code>gsl_spmatrix_char</code>	char
<code>gsl_spmatrix_uchar</code>	unsigned char
<code>gsl_spmatrix_complex</code>	complex double
<code>gsl_spmatrix_complex_float</code>	complex float
<code>gsl_spmatrix_complex_long_double</code>	complex long double

## 43.2 Sparse Matrix Storage Formats

GSL currently supports three storage formats for sparse matrices: the coordinate (COO) representation, compressed sparse column (CSC) and compressed sparse row (CSR) formats. These are discussed in more detail below. In order to illustrate the different storage formats, the following sections will reference this  $M$ -by- $N$  sparse matrix, with  $M = 4$  and  $N = 5$ :

$$\begin{pmatrix} 9 & 0 & 0 & 0 & -3 \\ 4 & 7 & 0 & 0 & 0 \\ 0 & 8 & -1 & 8 & 0 \\ 4 & 0 & 5 & 6 & 0 \end{pmatrix}$$

The number of non-zero elements in the matrix, also abbreviated as `nnz` is equal to 10 in this case.

### 43.2.1 Coordinate Storage (COO)

The coordinate storage format, also known as *triplet format*, stores triplets  $(i, j, x)$  for each non-zero element of the matrix. This notation means that the  $(i, j)$  element of the matrix  $A$  is  $A_{ij} = x$ . The matrix is stored using three arrays of the same length, representing the row indices, column indices, and matrix data. For the reference matrix above, one possible storage format is:

data	9	7	4	8	-3	-1	8	5	6	4
row	0	1	1	2	0	2	2	3	3	3
col	0	1	0	1	4	2	3	2	3	0

Note that this representation is not unique - the coordinate triplets may appear in any ordering and would still represent the same sparse matrix. The length of the three arrays is equal to the number of non-zero elements in the matrix, `nnz`, which in this case is 10. The coordinate format is extremely convenient for sparse matrix *assembly*, the process of adding new elements, or changing existing elements, in a sparse matrix. However, it is generally not suitable for the efficient implementation of matrix-matrix products, or matrix factorization algorithms. For these applications it is better to use one of the compressed formats discussed below.

In order to facilitate efficient sparse matrix assembly, GSL stores the coordinate data in a balanced binary search tree, specifically an AVL tree, in addition to the three arrays discussed above. This allows GSL to efficiently determine whether an entry  $(i, j)$  already exists in the matrix, and to replace an existing matrix entry with a new value, without needing to search unsorted arrays.

### 43.2.2 Compressed Sparse Column (CSC)

Compressed sparse column storage stores each column of non-zero values in the sparse matrix in a continuous memory block, keeping pointers to the beginning of each column in that memory block, and storing the row indices of each non-zero element. For the reference matrix above, these arrays look like

data	9	4	4	7	8	-1	5	8	6	-3
row	0	1	3	1	2	2	3	2	3	0
col_ptr	0	3	5	7	9	10				

The `data` and `row` arrays are of length `nnz` and are the same as the COO storage format. The `col_ptr` array has length  $N + 1$ , and `col_ptr[j]` gives the index in `data` of the start of column  $j$ . Therefore, the  $j$ -th column of the matrix is stored in `data[col_ptr[j]]`, `data[col_ptr[j] + 1]`, ..., `data[col_ptr[j+1] - 1]`. The last element of `col_ptr` is `nnz`.

### 43.2.3 Compressed Sparse Row (CSR)

Compressed row storage stores each row of non-zero values in a continuous memory block, keeping pointers to the beginning of each row in the block and storing the column indices of each non-zero element. For the reference matrix above, these arrays look like

data	9	-3	4	7	8	-1	8	4	5	6
col	0	4	0	1	1	2	3	0	2	3
row_ptr	0	2	4	7	10					

The `data` and `col` arrays are of length `nnz` and are the same as the COO storage format. The `row_ptr` array has length  $M + 1$ , and `row_ptr[i]` gives the index in `data` of the start of row  $i$ . Therefore, the  $i$ -th row of the matrix is stored in `data[row_ptr[i]]`, `data[row_ptr[i] + 1]`, ..., `data[row_ptr[i+1] - 1]`. The last element of `row_ptr` is `nnz`.

## 43.3 Overview

These routines provide support for constructing and manipulating sparse matrices in GSL, using an API similar to `gsl_matrix`. The basic structure is called `gsl_spmatrix`.

type `gsl_spmatrix`

This structure is defined as:

```
typedef struct
{
    size_t size1;
    size_t size2;
    int *i;
    double *data;
    int *p;
    size_t nzmax;
    size_t nz;
    [ ... variables for binary tree and memory management ... ]
    size_t sptype;
} gsl_spmatrix;
```

This defines a `size1`-by-`size2` sparse matrix. The number of non-zero elements currently in the matrix is given by `nz`. For the triplet representation, `i`, `p`, and `data` are arrays of size `nz` which contain the row indices, column indices, and element value, respectively. So if  $data[k] = A(i, j)$ , then  $i = i[k]$  and  $j = p[k]$ .

For compressed column storage, `i` and `data` are arrays of size `nz` containing the row indices and element values, identical to the triplet case. `p` is an array of size `size2 + 1` where `p[j]` points to the index in `data` of the start of column  $j$ . Thus, if  $data[k] = A(i, j)$ , then  $i = i[k]$  and  $p[j] \leq k < p[j + 1]$ .

For compressed row storage, `i` and `data` are arrays of size `nz` containing the column indices and element values, identical to the triplet case. `p` is an array of size `size1 + 1` where `p[i]` points to the index in `data` of the start of row  $i$ . Thus, if  $data[k] = A(i, j)$ , then  $j = i[k]$  and  $p[i] \leq k < p[i + 1]$ .

There are additional variables in the `gsl_spmatrix` structure related to binary tree storage and memory management. The GSL implementation of sparse matrices uses balanced AVL trees to sort matrix elements in the triplet representation. This speeds up element searches and duplicate detection during the matrix assembly process. The `gsl_spmatrix` structure also contains additional workspace variables needed for various operations like converting from triplet to compressed storage. `sptype` indicates the type of storage format being used (COO, CSC or CSR).

The compressed storage format defined above makes it very simple to interface with sophisticated external linear solver libraries which accept compressed storage input. The user can simply pass the arrays `i`, `p`, and `data` as the inputs to external libraries.

## 43.4 Allocation

The functions for allocating memory for a sparse matrix follow the style of `malloc()` and `free()`. They also perform their own error checking. If there is insufficient memory available to allocate a matrix then the functions call the GSL error handler with an error code of `GSL_ENOMEM` in addition to returning a null pointer.

*gsl\_spmatrix* \***gsl\_spmatrix\_alloc**(const size\_t *n1*, const size\_t *n2*)

This function allocates a sparse matrix of size *n1*-by-*n2* and initializes it to all zeros. If the size of the matrix is not known at allocation time, both *n1* and *n2* may be set to 1, and they will automatically grow as elements are added to the matrix. This function sets the matrix to the triplet representation, which is the easiest for adding and accessing matrix elements. This function tries to make a reasonable guess for the number of non-zero elements (*nzmax*) which will be added to the matrix by assuming a sparse density of 10%. The function `gsl_spmatrix_alloc_nzmax()` can be used if this number is known more accurately. The workspace is of size  $O(nzmax)$ .

*gsl\_spmatrix* \***gsl\_spmatrix\_alloc\_nzmax**(const size\_t *n1*, const size\_t *n2*, const size\_t *nzmax*, const size\_t *sptype*)

This function allocates a sparse matrix of size *n1*-by-*n2* and initializes it to all zeros. If the size of the matrix is not known at allocation time, both *n1* and *n2* may be set to 1, and they will automatically grow as elements are added to the matrix. The parameter *nzmax* specifies the maximum number of non-zero elements which will be added to the matrix. It does not need to be precisely known in advance, since storage space will automatically grow using `gsl_spmatrix_realloc()` if *nzmax* is not large enough. Accurate knowledge of this parameter reduces the number of reallocation calls required. The parameter *sptype* specifies the storage format of the sparse matrix. Possible values are

**GSL\_SPMATRIX\_COO**

This flag specifies coordinate (triplet) storage.

**GSL\_SPMATRIX\_CSC**

This flag specifies compressed sparse column storage.

**GSL\_SPMATRIX\_CSR**

This flag specifies compressed sparse row storage.

The allocated *gsl\_spmatrix* structure is of size  $O(nzmax)$ .

int **gsl\_spmatrix\_realloc**(const size\_t *nzmax*, *gsl\_spmatrix* \**m*)

This function reallocates the storage space for *m* to accommodate *nzmax* non-zero elements. It is typically called internally by `gsl_spmatrix_set()` if the user wants to add more elements to the sparse matrix than the previously specified *nzmax*.

Input matrix formats supported: *COO*, *CSC*, *CSR*

void **gsl\_spmatrix\_free**(*gsl\_spmatrix* \**m*)

This function frees the memory associated with the sparse matrix *m*.

Input matrix formats supported: *COO*, *CSC*, *CSR*



## 43.5 Accessing Matrix Elements

double **gsl\_spmatrix\_get**(const *gsl\_spmatrix* \*m, const size\_t i, const size\_t j)

This function returns element  $(i, j)$  of the matrix  $m$ .

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_set**(*gsl\_spmatrix* \*m, const size\_t i, const size\_t j, const double x)

This function sets element  $(i, j)$  of the matrix  $m$  to the value  $x$ .

Input matrix formats supported: *COO*

double \***gsl\_spmatrix\_ptr**(*gsl\_spmatrix* \*m, const size\_t i, const size\_t j)

This function returns a pointer to the  $(i, j)$  element of the matrix  $m$ . If the  $(i, j)$  element is not explicitly stored in the matrix, a null pointer is returned.

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.6 Initializing Matrix Elements

Since the sparse matrix format only stores the non-zero elements, it is automatically initialized to zero upon allocation. The function *gsl\_spmatrix\_set\_zero()* may be used to re-initialize a matrix to zero after elements have been added to it.

int **gsl\_spmatrix\_set\_zero**(*gsl\_spmatrix* \*m)

This function sets (or resets) all the elements of the matrix  $m$  to zero. For *CSC* and *CSR* matrices, the cost of this operation is  $O(1)$ . For *COO* matrices, the binary tree structure must be dismantled, so the cost is  $O(nz)$ .

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.7 Reading and Writing Matrices

int **gsl\_spmatrix\_fwrite**(FILE \*stream, const *gsl\_spmatrix* \*m)

This function writes the elements of the matrix  $m$  to the stream *stream* in binary format. The return value is 0 for success and *GSL\_EFAILED* if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_fread**(FILE \*stream, *gsl\_spmatrix* \*m)

This function reads into the matrix  $m$  from the open stream *stream* in binary format. The matrix  $m$  must be preallocated with the correct storage format, dimensions and have a sufficiently large *nzmax* in order to read in all matrix elements, otherwise *GSL\_EBADLEN* is returned. The return value is 0 for success and *GSL\_EFAILED* if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_fprintf**(FILE \*stream, const *gsl\_spmatrix* \*m, const char \*format)

This function writes the elements of the matrix  $m$  line-by-line to the stream *stream* using the format specifier *format*, which should be one of the *%g*, *%e* or *%f* formats for floating point numbers. The function returns 0 for success and *GSL\_EFAILED* if there was a problem writing to the file. The input matrix  $m$  may be in any storage format, and the output file will be written in MatrixMarket format.

Input matrix formats supported: *COO*, *CSC*, *CSR*

*gsl\_spmatrix* \***gsl\_spmatrix\_fscanf**(FILE \*stream)

This function reads sparse matrix data in the MatrixMarket format from the stream *stream* and stores it in a newly allocated matrix which is returned in *COO* format. The function returns 0 for success and *GSL\_EFAILED* if there was a problem reading from the file. The user should free the returned matrix when it is no longer needed.

## 43.8 Copying Matrices

int **gsl\_spmatrix\_memcpy**(*gsl\_spmatrix* \*dest, const *gsl\_spmatrix* \*src)

This function copies the elements of the sparse matrix *src* into *dest*. The two matrices must have the same dimensions and be in the same storage format.

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.9 Exchanging Rows and Columns

int **gsl\_spmatrix\_transpose\_memcpy**(*gsl\_spmatrix* \*dest, const *gsl\_spmatrix* \*src)

This function copies the transpose of the sparse matrix *src* into *dest*. The dimensions of *dest* must match the transpose of the matrix *src*. Also, both matrices must use the same sparse storage format.

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_transpose**(*gsl\_spmatrix* \*m)

This function replaces the matrix *m* by its transpose, but changes the storage format for compressed matrix inputs. Since compressed column storage is the transpose of compressed row storage, this function simply converts a *CSC* matrix to *CSR* and vice versa. This is the most efficient way to transpose a compressed storage matrix, but the user should note that the storage format of their compressed matrix will change on output. For *COO* matrix inputs, the output matrix is also in *COO* storage.

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.10 Matrix Operations

int **gsl\_spmatrix\_scale**(*gsl\_spmatrix* \*m, const double x)

This function scales all elements of the matrix *m* by the constant factor *x*. The result  $m(i, j) \leftarrow xm(i, j)$  is stored in *m*.

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_scale\_columns**(*gsl\_spmatrix* \*A, const *gsl\_vector* \*x)

This function scales the columns of the *M*-by-*N* sparse matrix *A* by the elements of the vector *x*, of length *N*. The *j*-th column of *A* is multiplied by *x*[*j*]. This is equivalent to forming

$$A \rightarrow AX$$

where  $X = \text{diag}(x)$ .

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_scale\_rows**(*gsl\_spmatrix* \*A, const *gsl\_vector* \*x)

This function scales the rows of the *M*-by-*N* sparse matrix *A* by the elements of the vector *x*, of length *M*. The *i*-th row of *A* is multiplied by *x*[*i*]. This is equivalent to forming

$$A \rightarrow XA$$

where  $X = \text{diag}(x)$ .

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_add**(*gsl\_spmatrix* \*c, const *gsl\_spmatrix* \*a, const *gsl\_spmatrix* \*b)

This function computes the sum  $c = a + b$ . The three matrices must have the same dimensions.

Input matrix formats supported: *CSC*, *CSR*

int **gsl\_spmatrix\_dense\_add**(*gsl\_matrix* \*a, const *gsl\_spmatrix* \*b)

This function adds the elements of the sparse matrix  $b$  to the elements of the dense matrix  $a$ . The result  $a(i, j) \leftarrow a(i, j) + b(i, j)$  is stored in  $a$  and  $b$  remains unchanged. The two matrices must have the same dimensions.

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_dense\_sub**(*gsl\_matrix* \*a, const *gsl\_spmatrix* \*b)

This function subtracts the elements of the sparse matrix  $b$  from the elements of the dense matrix  $a$ . The result  $a(i, j) \leftarrow a(i, j) - b(i, j)$  is stored in  $a$  and  $b$  remains unchanged. The two matrices must have the same dimensions.

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.11 Matrix Properties

const char \***gsl\_spmatrix\_type**(const *gsl\_spmatrix* \*m)

This function returns a string describing the sparse storage format of the matrix  $m$ . For example:

```
printf ("matrix is '%s' format.\n", gsl_spmatrix_type (m));
```

would print something like:

```
matrix is 'CSR' format.
```

Input matrix formats supported: *COO*, *CSC*, *CSR*

size\_t **gsl\_spmatrix\_nnz**(const *gsl\_spmatrix* \*m)

This function returns the number of non-zero elements in  $m$ .

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_equal**(const *gsl\_spmatrix* \*a, const *gsl\_spmatrix* \*b)

This function returns 1 if the matrices  $a$  and  $b$  are equal (by comparison of element values) and 0 otherwise. The matrices  $a$  and  $b$  must be in the same sparse storage format for comparison.

Input matrix formats supported: *COO*, *CSC*, *CSR*

double **gsl\_spmatrix\_norm1**(const *gsl\_spmatrix* \*A)

This function returns the 1-norm of the  $m$ -by- $n$  matrix  $A$ , defined as the maximum column sum,

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |A_{ij}|$$

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.12 Finding Maximum and Minimum Elements

int **gsl\_spmatrix\_minmax**(const *gsl\_spmatrix* \*m, double \*min\_out, double \*max\_out)

This function returns the minimum and maximum elements of the matrix *m*, storing them in *min\_out* and *max\_out*, and searching only the non-zero values.

Input matrix formats supported: *COO*, *CSC*, *CSR*

int **gsl\_spmatrix\_min\_index**(const *gsl\_spmatrix* \*m, size\_t \*imin, size\_t \*jmin)

This function returns the indices of the minimum value in the matrix *m*, searching only the non-zero values, and storing them in *imin* and *jmin*. When there are several equal minimum elements then the first element found is returned.

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.13 Compressed Format

These routines calculate a compressed matrix from a coordinate representation.

int **gsl\_spmatrix\_csc**(*gsl\_spmatrix* \*dest, const *gsl\_spmatrix* \*src)

This function creates a sparse matrix in *compressed sparse column* format from the input sparse matrix *src* which must be in COO format. The compressed matrix is stored in *dest*.

Input matrix formats supported: *COO*

int **gsl\_spmatrix\_csr**(*gsl\_spmatrix* \*dest, const *gsl\_spmatrix* \*src)

This function creates a sparse matrix in *compressed sparse row* format from the input sparse matrix *src* which must be in COO format. The compressed matrix is stored in *dest*.

Input matrix formats supported: *COO*

*gsl\_spmatrix* \***gsl\_spmatrix\_compress**(const *gsl\_spmatrix* \*src, const int sptype)

This function allocates a new sparse matrix, and stores *src* into it using the format specified by *sptype*. The input *sptype* can be one of `GSL_SPMATRIX_COO`, `GSL_SPMATRIX_CSC`, or `GSL_SPMATRIX_CSR`. A pointer to the newly allocated matrix is returned, and must be freed by the caller when no longer needed.

## 43.14 Conversion Between Sparse and Dense Matrices

The *gsl\_spmatrix* structure can be converted into the dense *gsl\_matrix* format and vice versa with the following routines.

int **gsl\_spmatrix\_d2sp**(*gsl\_spmatrix* \*S, const *gsl\_matrix* \*A)

This function converts the dense matrix *A* into sparse COO format and stores the result in *S*.

Input matrix formats supported: *COO*

int **gsl\_spmatrix\_sp2d**(*gsl\_matrix* \*A, const *gsl\_spmatrix* \*S)

This function converts the sparse matrix *S* into a dense matrix and stores the result in *A*.

Input matrix formats supported: *COO*, *CSC*, *CSR*

## 43.15 Examples

The following example program builds a 5-by-4 sparse matrix and prints it in coordinate, compressed column, and compressed row format. The matrix which is constructed is

$$\begin{pmatrix} 0 & 0 & 3.1 & 4.6 \\ 1 & 0 & 7.2 & 0 \\ 0 & 0 & 0 & 0 \\ 2.1 & 2.9 & 0 & 8.5 \\ 4.1 & 0 & 0 & 0 \end{pmatrix}$$

The output of the program is:

```
printing all matrix elements:
A(0,0) = 0
A(0,1) = 0
A(0,2) = 3.1
A(0,3) = 4.6
A(1,0) = 1
.
.
.
A(4,0) = 4.1
A(4,1) = 0
A(4,2) = 0
A(4,3) = 0
matrix in triplet format (i,j,Aij):
(0, 2, 3.1)
(0, 3, 4.6)
(1, 0, 1.0)
(1, 2, 7.2)
(3, 0, 2.1)
(3, 1, 2.9)
(3, 3, 8.5)
(4, 0, 4.1)
matrix in compressed column format:
i = [ 1, 3, 4, 3, 0, 1, 0, 3, ]
p = [ 0, 3, 4, 6, 8, ]
d = [ 1, 2.1, 4.1, 2.9, 3.1, 7.2, 4.6, 8.5, ]
matrix in compressed row format:
i = [ 2, 3, 0, 2, 0, 1, 3, 0, ]
p = [ 0, 2, 4, 4, 7, 8, ]
d = [ 3.1, 4.6, 1, 7.2, 2.1, 2.9, 8.5, 4.1, ]
```

We see in the compressed column output, the data array stores each column contiguously, the array *i* stores the row index of the corresponding data element, and the array *p* stores the index of the start of each column in the data array. Similarly, for the compressed row output, the data array stores each row contiguously, the array *i* stores the column index of the corresponding data element, and the *p* array stores the index of the start of each row in the data array.

```
#include <stdio.h>
#include <stdlib.h>

#include <gsl/gsl_spmatrix.h>
```

(continues on next page)

```

int
main()
{
    gsl_spmatrix *A = gsl_spmatrix_alloc(5, 4); /* triplet format */
    gsl_spmatrix *B, *C;
    size_t i, j;

    /* build the sparse matrix */
    gsl_spmatrix_set(A, 0, 2, 3.1);
    gsl_spmatrix_set(A, 0, 3, 4.6);
    gsl_spmatrix_set(A, 1, 0, 1.0);
    gsl_spmatrix_set(A, 1, 2, 7.2);
    gsl_spmatrix_set(A, 3, 0, 2.1);
    gsl_spmatrix_set(A, 3, 1, 2.9);
    gsl_spmatrix_set(A, 3, 3, 8.5);
    gsl_spmatrix_set(A, 4, 0, 4.1);

    printf("printing all matrix elements:\n");
    for (i = 0; i < 5; ++i)
        for (j = 0; j < 4; ++j)
            printf("A(%zu,%zu) = %g\n", i, j,
                gsl_spmatrix_get(A, i, j));

    /* print out elements in triplet format */
    printf("matrix in triplet format (i,j,Aij):\n");
    gsl_spmatrix_fprintf(stdout, A, "%.1f");

    /* convert to compressed column format */
    B = gsl_spmatrix_ccs(A);

    printf("matrix in compressed column format:\n");
    printf("i = [ ");
    for (i = 0; i < B->nz; ++i)
        printf("%d, ", B->i[i]);
    printf("]\n");

    printf("p = [ ");
    for (i = 0; i < B->size2 + 1; ++i)
        printf("%d, ", B->p[i]);
    printf("]\n");

    printf("d = [ ");
    for (i = 0; i < B->nz; ++i)
        printf("%g, ", B->data[i]);
    printf("]\n");

    /* convert to compressed row format */
    C = gsl_spmatrix_crs(A);

    printf("matrix in compressed row format:\n");
    printf("i = [ ");
    for (i = 0; i < C->nz; ++i)

```

(continues on next page)

(continued from previous page)

```
    printf("%d, ", C->i[i]);
    printf("]\n");

    printf("p = [ ");
    for (i = 0; i < C->size1 + 1; ++i)
        printf("%d, ", C->p[i]);
    printf("]\n");

    printf("d = [ ");
    for (i = 0; i < C->nz; ++i)
        printf("%g, ", C->data[i]);
    printf("]\n");

    gsl_spmatrix_free(A);
    gsl_spmatrix_free(B);
    gsl_spmatrix_free(C);

    return 0;
}
```

## 43.16 References and Further Reading

The algorithms used by these functions are described in the following sources,

- Davis, T. A., *Direct Methods for Sparse Linear Systems*, SIAM, 2006.
- CSparse software library, <https://www.cise.ufl.edu/research/sparse/CSparse>





## SPARSE BLAS SUPPORT

The Sparse Basic Linear Algebra Subprograms (BLAS) define a set of fundamental operations on vectors and sparse matrices which can be used to create optimized higher-level linear algebra functionality. GSL supports a limited number of BLAS operations for sparse matrices.

The header file `gsl_sblas.h` contains the prototypes for the sparse BLAS functions and related declarations.

### 44.1 Sparse BLAS operations

int **gsl\_sblas\_dgemv**(const CBLAS\_TRANSPOSE\_t TransA, const double alpha, const *gsl\_spmatrix* \*A, const *gsl\_vector* \*x, const double beta, *gsl\_vector* \*y)

This function computes the matrix-vector product and sum  $y \leftarrow \alpha op(A)x + \beta y$ , where  $op(A) = A, A^T$  for  $TransA = CblasNoTrans, CblasTrans$ . In-place computations are not supported, so  $x$  and  $y$  must be distinct vectors. The matrix  $A$  may be in triplet or compressed format.

int **gsl\_sblas\_dgemm**(const double alpha, const *gsl\_spmatrix* \*A, const *gsl\_spmatrix* \*B, *gsl\_spmatrix* \*C)

This function computes the sparse matrix-matrix product  $C = \alpha AB$ . The matrices must be in compressed format.

### 44.2 References and Further Reading

The algorithms used by these functions are described in the following sources:

- Davis, T. A., *Direct Methods for Sparse Linear Systems*, SIAM, 2006.
- CSparse software library, <https://www.cise.ufl.edu/research/sparse/CSparse>



## SPARSE LINEAR ALGEBRA

This chapter describes functions for solving sparse linear systems of equations. The library provides linear algebra routines which operate directly on the *gsl\_spmatrix* and *gsl\_vector* objects.

The functions described in this chapter are declared in the header file `gsl_splinalg.h`.

### 45.1 Overview

This chapter is primarily concerned with the solution of the linear system

$$Ax = b$$

where  $A$  is a general square  $n$ -by- $n$  non-singular sparse matrix,  $x$  is an unknown  $n$ -by-1 vector, and  $b$  is a given  $n$ -by-1 right hand side vector. There exist many methods for solving such sparse linear systems, which broadly fall into either direct or iterative categories. Direct methods include LU and QR decompositions, while iterative methods start with an initial guess for the vector  $x$  and update the guess through iteration until convergence. GSL does not currently provide any direct sparse solvers.

### 45.2 Sparse Iterative Solvers

#### 45.2.1 Overview

Many practical iterative methods of solving large  $n$ -by- $n$  sparse linear systems involve projecting an approximate solution for  $\mathbf{x}$  onto a subspace of  $\mathbf{R}^n$ . If we define a  $m$ -dimensional subspace  $\mathcal{K}$  as the subspace of approximations to the solution  $\mathbf{x}$ , then  $m$  constraints must be imposed to determine the next approximation. These  $m$  constraints define another  $m$ -dimensional subspace denoted by  $\mathcal{L}$ . The subspace dimension  $m$  is typically chosen to be much smaller than  $n$  in order to reduce the computational effort needed to generate the next approximate solution vector. The many iterative algorithms which exist differ mainly in their choice of  $\mathcal{K}$  and  $\mathcal{L}$ .

#### 45.2.2 Types of Sparse Iterative Solvers

The sparse linear algebra library provides the following types of iterative solvers:

type `gsl_splinalg_itersolve_type`

*gsl\_splinalg\_itsolve\_type* \***gsl\_splinalg\_itsolve\_gmres**

This specifies the Generalized Minimum Residual Method (GMRES). This is a projection method using  $\mathcal{K} = \mathcal{K}_m$  and  $\mathcal{L} = A\mathcal{K}_m$  where  $\mathcal{K}_m$  is the  $m$ -th Krylov subspace

$$\mathcal{K}_m = \text{span} \{r_0, Ar_0, \dots, A^{m-1}r_0\}$$

and  $r_0 = b - Ax_0$  is the residual vector of the initial guess  $x_0$ . If  $m$  is set equal to  $n$ , then the Krylov subspace is  $\mathbf{R}^n$  and GMRES will provide the exact solution  $\mathbf{x}$ . However, the goal is for the method to arrive at a very good approximation to  $\mathbf{x}$  using a much smaller subspace  $\mathcal{K}_m$ . By default, the GMRES method selects  $m = \text{MIN}(n, 10)$  but the user may specify a different value for  $m$ . The GMRES storage requirements grow as  $O(n(m+1))$  and the number of flops grow as  $O(4m^2n - 4m^3/3)$ .

In the below function *gsl\_splinalg\_itsolve\_iterate()*, one GMRES iteration is defined as projecting the approximate solution vector onto each Krylov subspace  $\mathcal{K}_1, \dots, \mathcal{K}_m$ , and so  $m$  can be kept smaller by “restarting” the method and calling *gsl\_splinalg\_itsolve\_iterate()* multiple times, providing the updated approximation  $\mathbf{x}$  to each new call. If the method is not adequately converging, the user may try increasing the parameter  $m$ .

GMRES is considered a robust general purpose iterative solver, however there are cases where the method stagnates if the matrix is not positive-definite and fails to reduce the residual until the very last projection onto the subspace  $\mathcal{K}_n = \mathbf{R}^n$ . In these cases, preconditioning the linear system can help, but GSL does not currently provide any preconditioners.

### 45.2.3 Iterating the Sparse Linear System

The following functions are provided to allocate storage for the sparse linear solvers and iterate the system to a solution.

*gsl\_splinalg\_itsolve* \***gsl\_splinalg\_itsolve\_alloc**(const *gsl\_splinalg\_itsolve\_type* \**T*, const size\_t *n*,  
const size\_t *m*)

This function allocates a workspace for the iterative solution of  $n$ -by- $n$  sparse matrix systems. The iterative solver type is specified by *T*. The argument *m* specifies the size of the solution candidate subspace  $\mathcal{K}_m$ . The dimension *m* may be set to 0 in which case a reasonable default value is used.

void **gsl\_splinalg\_itsolve\_free**(*gsl\_splinalg\_itsolve* \**w*)

This function frees the memory associated with the workspace *w*.

const char \***gsl\_splinalg\_itsolve\_name**(const *gsl\_splinalg\_itsolve* \**w*)

This function returns a string pointer to the name of the solver.

int **gsl\_splinalg\_itsolve\_iterate**(const *gsl\_spmatrix* \**A*, const *gsl\_vector* \**b*, const double *tol*, *gsl\_vector*  
\**x*, *gsl\_splinalg\_itsolve* \**w*)

This function performs one iteration of the iterative method for the sparse linear system specified by the matrix *A*, right hand side vector *b* and solution vector *x*. On input, *x* must be set to an initial guess for the solution. On output, *x* is updated to give the current solution estimate. The parameter *tol* specifies the relative tolerance between the residual norm and norm of *b* in order to check for convergence. When the following condition is satisfied:

$$\|Ax - b\| \leq \text{tol} \times \|b\|$$

the method has converged, the function returns `GSL_SUCCESS` and the final solution is provided in *x*. Otherwise, the function returns `GSL_CONTINUE` to signal that more iterations are required. Here,  $\|\cdot\|$  represents the Euclidean norm. The input matrix *A* may be in triplet or compressed format.

double **gsl\_splinalg\_itsolve\_normr**(const *gsl\_splinalg\_itsolve* \**w*)

This function returns the current residual norm  $\|r\| = \|Ax - b\|$ , which is updated after each call to *gsl\_splinalg\_itsolve\_iterate()*.

## 45.3 Examples

This example program demonstrates the sparse linear algebra routines on the solution of a simple 1D Poisson equation on  $[0, 1]$ :

$$\frac{d^2 u(x)}{dx^2} = f(x) = -\pi^2 \sin(\pi x)$$

with boundary conditions  $u(0) = u(1) = 0$ . The analytic solution of this simple problem is  $u(x) = \sin \pi x$ . We will solve this problem by finite differencing the left hand side to give

$$\frac{1}{h^2} (u_{i+1} - 2u_i + u_{i-1}) = f_i$$

Defining a grid of  $N$  points,  $h = 1/(N - 1)$ . In the finite difference equation above,  $u_0 = u_{N-1} = 0$  are known from the boundary conditions, so we will only put the equations for  $i = 1, \dots, N - 2$  into the matrix system. The resulting  $(N - 2) \times (N - 2)$  matrix equation is

$$\frac{1}{h^2} \begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 1 & -2 & 1 \\ 0 & \dots & \dots & \dots & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-3} \\ u_{N-2} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-3} \\ f_{N-2} \end{pmatrix}$$

An example program which constructs and solves this system is given below. The system is solved using the iterative GMRES solver. Here is the output of the program:

```
iter 0 residual = 4.297275996844e-11
Converged
```

showing that the method converged in a single iteration. The calculated solution is shown in Fig. 45.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <gsl/gsl_math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_spmatrix.h>
#include <gsl/gsl_splinalg.h>

int
main()
{
    const size_t N = 100;           /* number of grid points */
    const size_t n = N - 2;       /* subtract 2 to exclude boundaries */
    const double h = 1.0 / (N - 1.0); /* grid spacing */
    gsl_spmatrix *A = gsl_spmatrix_alloc(n, n); /* triplet format */
    gsl_spmatrix *C;               /* compressed format */
    gsl_vector *f = gsl_vector_alloc(n); /* right hand side vector */
    gsl_vector *u = gsl_vector_alloc(n); /* solution vector */
    size_t i;
```

(continues on next page)



Fig. 45.1: Solution of PDE

(continued from previous page)

```

/* construct the sparse matrix for the finite difference equation */

/* construct first row */
gsl_spmatrix_set(A, 0, 0, -2.0);
gsl_spmatrix_set(A, 0, 1, 1.0);

/* construct rows [1:n-2] */
for (i = 1; i < n - 1; ++i)
{
    gsl_spmatrix_set(A, i, i + 1, 1.0);
    gsl_spmatrix_set(A, i, i, -2.0);
    gsl_spmatrix_set(A, i, i - 1, 1.0);
}

/* construct last row */
gsl_spmatrix_set(A, n - 1, n - 1, -2.0);
gsl_spmatrix_set(A, n - 1, n - 2, 1.0);

/* scale by h^2 */
gsl_spmatrix_scale(A, 1.0 / (h * h));

/* construct right hand side vector */
for (i = 0; i < n; ++i)
{
    double xi = (i + 1) * h;
    double fi = -M_PI * M_PI * sin(M_PI * xi);
    gsl_vector_set(f, i, fi);
}

/* convert to compressed column format */
C = gsl_spmatrix_ccs(A);

/* now solve the system with the GMRES iterative solver */
{
    const double tol = 1.0e-6; /* solution relative tolerance */
    const size_t max_iter = 10; /* maximum iterations */
    const gsl_splinalg_itorsolve_type *T = gsl_splinalg_itorsolve_gmres;
    gsl_splinalg_itorsolve *work =
        gsl_splinalg_itorsolve_alloc(T, n, 0);
    size_t iter = 0;
    double residual;
    int status;

    /* initial guess u = 0 */
    gsl_vector_set_zero(u);

    /* solve the system A u = f */
    do
    {
        status = gsl_splinalg_itorsolve_iterate(C, f, tol, u, work);

        /* print out residual norm ||A*u - f|| */
    }
}

```

(continues on next page)

```
    residual = gsl_splinalg_itersolve_normr(work);
    fprintf(stderr, "iter %zu residual = %.12e\n", iter, residual);

    if (status == GSL_SUCCESS)
        fprintf(stderr, "Converged\n");
}
while (status == GSL_CONTINUE && ++iter < max_iter);

/* output solution */
for (i = 0; i < n; ++i)
{
    double xi = (i + 1) * h;
    double u_exact = sin(M_PI * xi);
    double u_gsl = gsl_vector_get(u, i);

    printf("%f %.12e %.12e\n", xi, u_gsl, u_exact);
}

gsl_splinalg_itersolve_free(work);
}

gsl_spmatrix_free(A);
gsl_spmatrix_free(C);
gsl_vector_free(f);
gsl_vector_free(u);

return 0;
} /* main() */
```

## 45.4 References and Further Reading

The implementation of the GMRES iterative solver closely follows the publications

- H. F. Walker, Implementation of the GMRES method using Householder transformations, SIAM J. Sci. Stat. Comput. 9(1), 1988.
- Y. Saad, Iterative methods for sparse linear systems, 2nd edition, SIAM, 2003.



## PHYSICAL CONSTANTS

This chapter describes macros for the values of physical constants, such as the speed of light,  $c$ , and gravitational constant,  $G$ . The values are available in different unit systems, including the standard MKSA system (meters, kilograms, seconds, amperes) and the CGSM system (centimeters, grams, seconds, gauss), which is commonly used in Astronomy.

The definitions of constants in the MKSA system are available in the file `gsl_const_mkسا.h`. The constants in the CGSM system are defined in `gsl_const_cgsm.h`. Dimensionless constants, such as the fine structure constant, which are pure numbers are defined in `gsl_const_num.h`.

The full list of constants is described briefly below. Consult the header files themselves for the values of the constants used in the library.

### 46.1 Fundamental Constants

#### **GSL\_CONST\_MKSA\_SPEED\_OF\_LIGHT**

The speed of light in vacuum,  $c$ .

#### **GSL\_CONST\_MKSA\_VACUUM\_PERMEABILITY**

The permeability of free space,  $\mu_0$ . This constant is defined in the MKSA system only.

#### **GSL\_CONST\_MKSA\_VACUUM\_PERMITTIVITY**

The permittivity of free space,  $\epsilon_0$ . This constant is defined in the MKSA system only.

#### **GSL\_CONST\_MKSA\_PLANCKS\_CONSTANT\_H**

Planck's constant,  $h$ .

#### **GSL\_CONST\_MKSA\_PLANCKS\_CONSTANT\_HBAR**

Planck's constant divided by  $2\pi$ ,  $\hbar$ .

#### **GSL\_CONST\_NUM\_AVOGADRO**

Avogadro's number,  $N_a$ .

#### **GSL\_CONST\_MKSA\_FARADAY**

The molar charge of 1 Faraday.

#### **GSL\_CONST\_MKSA\_BOLTZMANN**

The Boltzmann constant,  $k$ .

#### **GSL\_CONST\_MKSA\_MOLAR\_GAS**

The molar gas constant,  $R_0$ .

#### **GSL\_CONST\_MKSA\_STANDARD\_GAS\_VOLUME**

The standard gas volume,  $V_0$ .

#### **GSL\_CONST\_MKSA\_STEFAN\_BOLTZMANN\_CONSTANT**

The Stefan-Boltzmann radiation constant,  $\sigma$ .

**GSL\_CONST\_MKSA\_GAUSS**

The magnetic field of 1 Gauss.

## 46.2 Astronomy and Astrophysics

**GSL\_CONST\_MKSA\_ASTRONOMICAL\_UNIT**

The length of 1 astronomical unit (mean earth-sun distance), *au*.

**GSL\_CONST\_MKSA\_GRAVITATIONAL\_CONSTANT**

The gravitational constant, *G*.

**GSL\_CONST\_MKSA\_LIGHT\_YEAR**

The distance of 1 light-year, *ly*.

**GSL\_CONST\_MKSA\_PARSEC**

The distance of 1 parsec, *pc*.

**GSL\_CONST\_MKSA\_GRAV\_ACCEL**

The standard gravitational acceleration on Earth, *g*.

**GSL\_CONST\_MKSA\_SOLAR\_MASS**

The mass of the Sun.

## 46.3 Atomic and Nuclear Physics

**GSL\_CONST\_MKSA\_ELECTRON\_CHARGE**

The charge of the electron, *e*.

**GSL\_CONST\_MKSA\_ELECTRON\_VOLT**

The energy of 1 electron volt, *eV*.

**GSL\_CONST\_MKSA\_UNIFIED\_ATOMIC\_MASS**

The unified atomic mass, *amu*.

**GSL\_CONST\_MKSA\_MASS\_ELECTRON**

The mass of the electron,  $m_e$ .

**GSL\_CONST\_MKSA\_MASS\_MUON**

The mass of the muon,  $m_\mu$ .

**GSL\_CONST\_MKSA\_MASS\_PROTON**

The mass of the proton,  $m_p$ .

**GSL\_CONST\_MKSA\_MASS\_NEUTRON**

The mass of the neutron,  $m_n$ .

**GSL\_CONST\_NUM\_FINE\_STRUCTURE**

The electromagnetic fine structure constant  $\alpha$ .

**GSL\_CONST\_MKSA\_RYDBERG**

The Rydberg constant, *Ry*, in units of energy. This is related to the Rydberg inverse wavelength  $R_\infty$  by  $Ry = hcR_\infty$ .

**GSL\_CONST\_MKSA\_BOHR\_RADIUS**

The Bohr radius,  $a_0$ .

**GSL\_CONST\_MKSA\_ANGSTROM**

The length of 1 angstrom.

**GSL\_CONST\_MKSA\_BARN**

The area of 1 barn.

**GSL\_CONST\_MKSA\_BOHR\_MAGNETON**

The Bohr Magneton,  $\mu_B$ .

**GSL\_CONST\_MKSA\_NUCLEAR\_MAGNETON**

The Nuclear Magnetron,  $\mu_N$ .

**GSL\_CONST\_MKSA\_ELECTRON\_MAGNETIC\_MOMENT**

The absolute value of the magnetic moment of the electron,  $\mu_e$ . The physical magnetic moment of the electron is negative.

**GSL\_CONST\_MKSA\_PROTON\_MAGNETIC\_MOMENT**

The magnetic moment of the proton,  $\mu_p$ .

**GSL\_CONST\_MKSA\_THOMSON\_CROSS\_SECTION**

The Thomson cross section,  $\sigma_T$ .

**GSL\_CONST\_MKSA\_DEBYE**

The electric dipole moment of 1 Debye,  $D$ .

## 46.4 Measurement of Time

**GSL\_CONST\_MKSA\_MINUTE**

The number of seconds in 1 minute.

**GSL\_CONST\_MKSA\_HOUR**

The number of seconds in 1 hour.

**GSL\_CONST\_MKSA\_DAY**

The number of seconds in 1 day.

**GSL\_CONST\_MKSA\_WEEK**

The number of seconds in 1 week.

## 46.5 Imperial Units

**GSL\_CONST\_MKSA\_INCH**

The length of 1 inch.

**GSL\_CONST\_MKSA\_FOOT**

The length of 1 foot.

**GSL\_CONST\_MKSA\_YARD**

The length of 1 yard.

**GSL\_CONST\_MKSA\_MILE**

The length of 1 mile.

**GSL\_CONST\_MKSA\_MIL**

The length of 1 mil (1/1000th of an inch).

## 46.6 Speed and Nautical Units

**GSL\_CONST\_MKSA\_KILOMETERS\_PER\_HOUR**

The speed of 1 kilometer per hour.

**GSL\_CONST\_MKSA\_MILES\_PER\_HOUR**

The speed of 1 mile per hour.

**GSL\_CONST\_MKSA\_NAUTICAL\_MILE**

The length of 1 nautical mile.

**GSL\_CONST\_MKSA\_FATHOM**

The length of 1 fathom.

**GSL\_CONST\_MKSA\_KNOT**

The speed of 1 knot.

## 46.7 Printers Units

**GSL\_CONST\_MKSA\_POINT**

The length of 1 printer's point (1/72 inch).

**GSL\_CONST\_MKSA\_TEXPOINT**

The length of 1 TeX point (1/72.27 inch).

## 46.8 Volume, Area and Length

**GSL\_CONST\_MKSA\_MICRON**

The length of 1 micron.

**GSL\_CONST\_MKSA\_HECTARE**

The area of 1 hectare.

**GSL\_CONST\_MKSA\_ACRE**

The area of 1 acre.

**GSL\_CONST\_MKSA\_LITER**

The volume of 1 liter.

**GSL\_CONST\_MKSA\_US\_GALLON**

The volume of 1 US gallon.

**GSL\_CONST\_MKSA\_CANADIAN\_GALLON**

The volume of 1 Canadian gallon.

**GSL\_CONST\_MKSA\_UK\_GALLON**

The volume of 1 UK gallon.

**GSL\_CONST\_MKSA\_QUART**

The volume of 1 quart.

**GSL\_CONST\_MKSA\_PINT**

The volume of 1 pint.

## 46.9 Mass and Weight

**GSL\_CONST\_MKSA\_POUND\_MASS**

The mass of 1 pound.

**GSL\_CONST\_MKSA\_OUNCE\_MASS**

The mass of 1 ounce.

**GSL\_CONST\_MKSA\_TON**

The mass of 1 ton.

**GSL\_CONST\_MKSA\_METRIC\_TON**

The mass of 1 metric ton (1000 kg).

**GSL\_CONST\_MKSA\_UK\_TON**

The mass of 1 UK ton.

**GSL\_CONST\_MKSA\_TROY\_OUNCE**

The mass of 1 troy ounce.

**GSL\_CONST\_MKSA\_CARAT**

The mass of 1 carat.

**GSL\_CONST\_MKSA\_GRAM\_FORCE**

The force of 1 gram weight.

**GSL\_CONST\_MKSA\_POUND\_FORCE**

The force of 1 pound weight.

**GSL\_CONST\_MKSA\_KILOPOUND\_FORCE**

The force of 1 kilopound weight.

**GSL\_CONST\_MKSA\_POUNDAL**

The force of 1 poundal.

## 46.10 Thermal Energy and Power

**GSL\_CONST\_MKSA\_CALORIE**

The energy of 1 calorie.

**GSL\_CONST\_MKSA\_BTU**

The energy of 1 British Thermal Unit, *btu*.

**GSL\_CONST\_MKSA\_THERM**

The energy of 1 Therm.

**GSL\_CONST\_MKSA\_HORSEPOWER**

The power of 1 horsepower.

## 46.11 Pressure

**GSL\_CONST\_MKSA\_BAR**

The pressure of 1 bar.

**GSL\_CONST\_MKSA\_STD\_ATMOSPHERE**

The pressure of 1 standard atmosphere.

**GSL\_CONST\_MKSA\_TORR**

The pressure of 1 torr.

**GSL\_CONST\_MKSA\_METER\_OF\_MERCURY**

The pressure of 1 meter of mercury.

**GSL\_CONST\_MKSA\_INCH\_OF\_MERCURY**

The pressure of 1 inch of mercury.

**GSL\_CONST\_MKSA\_INCH\_OF\_WATER**

The pressure of 1 inch of water.

**GSL\_CONST\_MKSA\_PSI**

The pressure of 1 pound per square inch.

## 46.12 Viscosity

**GSL\_CONST\_MKSA\_POISE**

The dynamic viscosity of 1 poise.

**GSL\_CONST\_MKSA\_STOKES**

The kinematic viscosity of 1 stokes.

## 46.13 Light and Illumination

**GSL\_CONST\_MKSA\_STILB**

The luminance of 1 stilb.

**GSL\_CONST\_MKSA\_LUMEN**

The luminous flux of 1 lumen.

**GSL\_CONST\_MKSA\_LUX**

The illuminance of 1 lux.

**GSL\_CONST\_MKSA\_PHOT**

The illuminance of 1 phot.

**GSL\_CONST\_MKSA\_FOOTCANDLE**

The illuminance of 1 footcandle.

**GSL\_CONST\_MKSA\_LAMBERT**

The luminance of 1 lambert.

**GSL\_CONST\_MKSA\_FOOTLAMBERT**

The luminance of 1 footlambert.

## 46.14 Radioactivity

### GSL\_CONST\_MKSA\_CURIE

The activity of 1 curie.

### GSL\_CONST\_MKSA\_ROENTGEN

The exposure of 1 roentgen.

### GSL\_CONST\_MKSA\_RAD

The absorbed dose of 1 rad.

## 46.15 Force and Energy

### GSL\_CONST\_MKSA\_NEWTON

The SI unit of force, 1 Newton.

### GSL\_CONST\_MKSA\_DYNE

The force of 1 Dyne =  $10^{-5}$  Newton.

### GSL\_CONST\_MKSA\_JOULE

The SI unit of energy, 1 Joule.

### GSL\_CONST\_MKSA\_ERG

The energy 1 erg =  $10^{-7}$  Joule.

## 46.16 Prefixes

These constants are dimensionless scaling factors.

### GSL\_CONST\_NUM\_YOTTA

$10^{24}$

### GSL\_CONST\_NUM\_ZETTA

$10^{21}$

### GSL\_CONST\_NUM\_EXA

$10^{18}$

### GSL\_CONST\_NUM\_PETA

$10^{15}$

### GSL\_CONST\_NUM\_TERA

$10^{12}$

### GSL\_CONST\_NUM\_GIGA

$10^9$

### GSL\_CONST\_NUM\_MEGA

$10^6$

### GSL\_CONST\_NUM\_KILO

$10^3$

### GSL\_CONST\_NUM\_MILLI

$10^{-3}$

**GSL\_CONST\_NUM\_MICRO**  
 $10^{-6}$

**GSL\_CONST\_NUM\_NANO**  
 $10^{-9}$

**GSL\_CONST\_NUM\_PICO**  
 $10^{-12}$

**GSL\_CONST\_NUM\_FEMTO**  
 $10^{-15}$

**GSL\_CONST\_NUM\_ATTO**  
 $10^{-18}$

**GSL\_CONST\_NUM\_ZEPTO**  
 $10^{-21}$

**GSL\_CONST\_NUM\_YOCTO**  
 $10^{-24}$

## 46.17 Examples

The following program demonstrates the use of the physical constants in a calculation. In this case, the goal is to calculate the range of light-travel times from Earth to Mars.

The required data is the average distance of each planet from the Sun in astronomical units (the eccentricities and inclinations of the orbits will be neglected for the purposes of this calculation). The average radius of the orbit of Mars is 1.52 astronomical units, and for the orbit of Earth it is 1 astronomical unit (by definition). These values are combined with the MKSA values of the constants for the speed of light and the length of an astronomical unit to produce a result for the shortest and longest light-travel times in seconds. The figures are converted into minutes before being displayed.

```
#include <stdio.h>
#include <gsl/gsl_const_mksa.h>

int
main (void)
{
    double c = GSL_CONST_MKSA_SPEED_OF_LIGHT;
    double au = GSL_CONST_MKSA_ASTRONOMICAL_UNIT;
    double minutes = GSL_CONST_MKSA_MINUTE;

    /* distance stored in meters */
    double r_earth = 1.00 * au;
    double r_mars = 1.52 * au;

    double t_min, t_max;

    t_min = (r_mars - r_earth) / c;
    t_max = (r_mars + r_earth) / c;

    printf ("light travel time from Earth to Mars:\n");
    printf ("minimum = %.1f minutes\n", t_min / minutes);
    printf ("maximum = %.1f minutes\n", t_max / minutes);
}
```

(continues on next page)



(continued from previous page)

```
return 0;  
}
```

Here is the output from the program,

```
light travel time from Earth to Mars:  
minimum = 4.3 minutes  
maximum = 21.0 minutes
```

## 46.18 References and Further Reading

The authoritative sources for physical constants are the 2006 CODATA recommended values, published in the article below. Further information on the values of physical constants is also available from the NIST website.

- P.J. Mohr, B.N. Taylor, D.B. Newell, “CODATA Recommended Values of the Fundamental Physical Constants: 2006”, *Reviews of Modern Physics*, 80(2), pp. 633–730 (2008).
- <http://www.physics.nist.gov/cuu/Constants/index.html>
- <http://physics.nist.gov/Pubs/SP811/appenB9.html>



## IEEE FLOATING-POINT ARITHMETIC

This chapter describes functions for examining the representation of floating point numbers and controlling the floating point environment of your program. The functions described in this chapter are declared in the header file `gsl_ieee_utils.h`.

### 47.1 Representation of floating point numbers

The IEEE Standard for Binary Floating-Point Arithmetic defines binary formats for single and double precision numbers. Each number is composed of three parts: a *sign bit* ( $s$ ), an *exponent* ( $E$ ) and a *fraction* ( $f$ ). The numerical value of the combination  $(s, E, f)$  is given by the following formula,

$$(-1)^s (1.ffff\dots)2^E$$

The sign bit is either zero or one. The exponent ranges from a minimum value  $E_{min}$  to a maximum value  $E_{max}$  depending on the precision. The exponent is converted to an unsigned number  $e$ , known as the *biased exponent*, for storage by adding a *bias* parameter,

$$e = E + bias$$

The sequence  $ffff\dots$  represents the digits of the binary fraction  $f$ . The binary digits are stored in *normalized form*, by adjusting the exponent to give a leading digit of 1. Since the leading digit is always 1 for normalized numbers it is assumed implicitly and does not have to be stored. Numbers smaller than  $2^{E_{min}}$  are stored in *denormalized form* with a leading zero,

$$(-1)^s (0.ffff\dots)2^{E_{min}}$$

This allows gradual underflow down to  $2^{E_{min}-p}$  for  $p$  bits of precision. A zero is encoded with the special exponent of  $2^{E_{min}-1}$  and infinities with the exponent of  $2^{E_{max}+1}$ .

The format for single precision numbers uses 32 bits divided in the following way:

```
seeeeeeeffffffffffffffffffffffffffff
```

```
s = sign bit, 1 bit  
e = exponent, 8 bits (E_min=-126, E_max=127, bias=127)  
f = fraction, 23 bits
```

The format for double precision numbers uses 64 bits divided in the following way:

```
seeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

```
s = sign bit, 1 bit
e = exponent, 11 bits (E_min=-1022, E_max=1023, bias=1023)
f = fraction, 52 bits
```

It is often useful to be able to investigate the behavior of a calculation at the bit-level and the library provides functions for printing the IEEE representations in a human-readable form.

```
void gsl_ieee_fprintf_float(FILE *stream, const float *x)
```

```
void gsl_ieee_fprintf_double(FILE *stream, const double *x)
```

These functions output a formatted version of the IEEE floating-point number pointed to by *x* to the stream *stream*. A pointer is used to pass the number indirectly, to avoid any undesired promotion from `float` to `double`. The output takes one of the following forms,

NaN

the Not-a-Number symbol

Inf, -Inf

positive or negative infinity

1.fffff...\*2<sup>E</sup>, -1.fffff...\*2<sup>E</sup>

a normalized floating point number

0.fffff...\*2<sup>E</sup>, -0.fffff...\*2<sup>E</sup>

a denormalized floating point number

0, -0

positive or negative zero

The output can be used directly in GNU Emacs Calc mode by preceding it with `2#` to indicate binary.

```
void gsl_ieee_printf_float(const float *x)
```

```
void gsl_ieee_printf_double(const double *x)
```

These functions output a formatted version of the IEEE floating-point number pointed to by *x* to the stream `stdout`.

The following program demonstrates the use of the functions by printing the single and double precision representations of the fraction 1/3. For comparison the representation of the value promoted from single to double precision is also printed.

```
#include <stdio.h>
#include <gsl/gsl_ieee_utils.h>

int
main (void)
{
    float f = 1.0/3.0;
    double d = 1.0/3.0;

    double fd = f; /* promote from float to double */
```

(continues on next page)

(continued from previous page)

```

printf (" f="); gsl_ieee_printf_float(&f);
printf ("\n");

printf ("fd="); gsl_ieee_printf_double(&fd);
printf ("\n");

printf (" d="); gsl_ieee_printf_double(&d);
printf ("\n");

return 0;
}

```

The binary representation of  $1/3$  is  $0.01010101\dots$ . The output below shows that the IEEE format normalizes this fraction to give a leading digit of 1:

```

f= 1.0101010101010101010101011*2^-2
fd= 1.0101010101010101010101011000000000000000000000000000000*2^-2
d= 1.01010101010101010101010101010101010101010101010101010101*2^-2

```

The output also shows that a single-precision number is promoted to double-precision by adding zeros in the binary representation.

## 47.2 Setting up your IEEE environment

The IEEE standard defines several *modes* for controlling the behavior of floating point operations. These modes specify the important properties of computer arithmetic: the direction used for rounding (e.g. whether numbers should be rounded up, down or to the nearest number), the rounding precision and how the program should handle arithmetic exceptions, such as division by zero.

Many of these features can now be controlled via standard functions such as `fpsetround()`, which should be used whenever they are available. Unfortunately in the past there has been no universal API for controlling their behavior—each system has had its own low-level way of accessing them. To help you write portable programs GSL allows you to specify modes in a platform-independent way using the environment variable `GSL_IEEE_MODE`. The library then takes care of all the necessary machine-specific initializations for you when you call the function `gsl_ieee_env_setup()`.

### `GSL_IEEE_MODE`

Environment variable which specifies IEEE mode.

### void `gsl_ieee_env_setup()`

This function reads the environment variable `GSL_IEEE_MODE` and attempts to set up the corresponding specified IEEE modes. The environment variable should be a list of keywords, separated by commas, like this:

```
GSL_IEEE_MODE = "keyword, keyword, ..."
```

where `keyword` is one of the following mode-names:

```

single-precision
double-precision
extended-precision
round-to-nearest
round-down

```

(continues on next page)

(continued from previous page)

```

round-up
round-to-zero
mask-all
mask-invalid
mask-denormalized
mask-division-by-zero
mask-overflow
mask-underflow
trap-inexact
trap-common

```

If `GSL_IEEE_MODE` is empty or undefined then the function returns immediately and no attempt is made to change the system's IEEE mode. When the modes from `GSL_IEEE_MODE` are turned on the function prints a short message showing the new settings to remind you that the results of the program will be affected.

If the requested modes are not supported by the platform being used then the function calls the error handler and returns an error code of `GSL_EUNSUP`.

When options are specified using this method, the resulting mode is based on a default setting of the highest available precision (double precision or extended precision, depending on the platform) in round-to-nearest mode, with all exceptions enabled apart from the INEXACT exception. The INEXACT exception is generated whenever rounding occurs, so it must generally be disabled in typical scientific calculations. All other floating-point exceptions are enabled by default, including underflows and the use of denormalized numbers, for safety. They can be disabled with the individual `mask-` settings or together using `mask-all`.

The following adjusted combination of modes is convenient for many purposes:

```

GSL_IEEE_MODE="double-precision,"\
              "mask-underflow,"\
              "mask-denormalized"

```

This choice ignores any errors relating to small numbers (either denormalized, or underflowing to zero) but traps overflows, division by zero and invalid operations.

Note that on the x86 series of processors this function sets both the original x87 mode and the newer MXCSR mode, which controls SSE floating-point operations. The SSE floating-point units do not have a precision-control bit, and always work in double-precision. The single-precision and extended-precision keywords have no effect in this case.

To demonstrate the effects of different rounding modes consider the following program which computes  $e$ , the base of natural logarithms, by summing a rapidly-decreasing series,

$$\begin{aligned}
 e &= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots \\
 &= 2.71828182846\dots
 \end{aligned}$$

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_ieee_utils.h>

int
main (void)
{
    double x = 1, oldsum = 0, sum = 0;

```

(continues on next page)

(continued from previous page)

```

int i = 0;

gsl_ieee_env_setup (); /* read GSL_IEEE_MODE */

do
{
    i++;

    oldsum = sum;
    sum += x;
    x = x / i;

    printf ("i=%2d sum=%.18f error=%g\n",
           i, sum, sum - M_E);

    if (i > 30)
        break;
}
while (sum != oldsum);

return 0;
}

```

Here are the results of running the program in `round-to-nearest` mode. This is the IEEE default so it isn't really necessary to specify it here:

```

$ GSL_IEEE_MODE="round-to-nearest" ./a.out
i= 1 sum=1.000000000000000000 error=-1.71828
i= 2 sum=2.000000000000000000 error=-0.718282
....
i=18 sum=2.718281828459045535 error=4.44089e-16
i=19 sum=2.718281828459045535 error=4.44089e-16

```

After nineteen terms the sum converges to within  $4 \times 10^{-16}$  of the correct value. If we now change the rounding mode to `round-down` the final result is less accurate:

```

$ GSL_IEEE_MODE="round-down" ./a.out
i= 1 sum=1.000000000000000000 error=-1.71828
....
i=19 sum=2.718281828459041094 error=-3.9968e-15

```

The result is about  $4 \times 10^{-15}$  below the correct value, an order of magnitude worse than the result obtained in the `round-to-nearest` mode.

If we change to rounding mode to `round-up` then the final result is higher than the correct value (when we add each term to the sum the final result is always rounded up, which increases the sum by at least one tick until the added term underflows to zero). To avoid this problem we would need to use a safer converge criterion, such as `while (fabs(sum - oldsum) > epsilon)`, with a suitably chosen value of `epsilon`.

Finally we can see the effect of computing the sum using single-precision rounding, in the default `round-to-nearest` mode. In this case the program thinks it is still using double precision numbers but the CPU rounds the result of each floating point operation to single-precision accuracy. This simulates the effect of writing the program using single-precision `float` variables instead of `double` variables. The iteration stops after about half the number of iterations and the final result is much less accurate:

```
$ GSL_IEEE_MODE="single-precision" ./a.out
....
i=12 sum=2.718281984329223633 error=1.5587e-07
```

with an error of  $O(10^{-7})$ , which corresponds to single precision accuracy (about 1 part in  $10^7$ ). Continuing the iterations further does not decrease the error because all the subsequent results are rounded to the same value.

## 47.3 References and Further Reading

The reference for the IEEE standard is,

- ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

A more pedagogical introduction to the standard can be found in the following paper,

- David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, Vol.: 23, No.: 1 (March 1991), pages 5–48.
- Corrigendum: *ACM Computing Surveys*, Vol.: 23, No.: 3 (September 1991), page 413. and see also the sections by B. A. Wichmann and Charles B. Dunham in Surveyor’s Forum: “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. *ACM Computing Surveys*, Vol.: 24, No.: 3 (September 1992), page 319.

A detailed textbook on IEEE arithmetic and its practical use is available from SIAM Press,

- Michael L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM Press, ISBN 0898715717.



## DEBUGGING NUMERICAL PROGRAMS

This chapter describes some tips and tricks for debugging numerical programs which use GSL.

### 48.1 Using gdb

Any errors reported by the library are passed to the function `gsl_error()`. By running your programs under `gdb` and setting a breakpoint in this function you can automatically catch any library errors. You can add a breakpoint for every session by putting:

```
break gsl_error
```

into your `.gdbinit` file in the directory where your program is started.

If the breakpoint catches an error then you can use a backtrace (`bt`) to see the call-tree, and the arguments which possibly caused the error. By moving up into the calling function you can investigate the values of variables at that point. Here is an example from the program `fft/test_trap`, which contains the following line:

```
status = gsl_fft_complex_wavetable_alloc (0, &complex_wavetable);
```

The function `gsl_fft_complex_wavetable_alloc()` takes the length of an FFT as its first argument. When this line is executed an error will be generated because the length of an FFT is not allowed to be zero.

To debug this problem we start `gdb`, using the file `.gdbinit` to define a breakpoint in `gsl_error()`:

```
$ gdb test_trap
```

```
GDB is free software and you are welcome to distribute copies
of it under certain conditions; type "show copying" to see
the conditions.  There is absolutely no warranty for GDB;
type "show warranty" for details.  GDB 4.16 (i586-debian-linux),
Copyright 1996 Free Software Foundation, Inc.
```

```
Breakpoint 1 at 0x8050b1e: file error.c, line 14.
```

When we run the program this breakpoint catches the error and shows the reason for it:

```
(gdb) run
Starting program: test_trap

Breakpoint 1, gsl_error (reason=0x8052b0d
    "length n must be positive integer",
```

(continues on next page)

(continued from previous page)

```

file=0x8052b04 "c_init.c", line=108, gsl_errno=1)
at error.c:14
14      if (gsl_error_handler)

```

The first argument of `gsl_error()` is always a string describing the error. Now we can look at the backtrace to see what caused the problem:

```

(gdb) bt
#0  gsl_error (reason=0x8052b0d
"length n must be positive integer",
file=0x8052b04 "c_init.c", line=108, gsl_errno=1)
at error.c:14
#1  0x8049376 in gsl_fft_complex_wavetable_alloc (n=0,
wavetable=0xbffff778) at c_init.c:108
#2  0x8048a00 in main (argc=1, argv=0xbffff9bc)
at test_trap.c:94
#3  0x80488be in ___crt_dummy__ ()

```

We can see that the error was generated in the function `gsl_fft_complex_wavetable_alloc()` when it was called with an argument of `n = 0`. The original call came from line 94 in the file `test_trap.c`.

By moving up to the level of the original call we can find the line that caused the error:

```

(gdb) up
#1  0x8049376 in gsl_fft_complex_wavetable_alloc (n=0,
wavetable=0xbffff778) at c_init.c:108
108  GSL_ERROR ("length n must be positive integer", GSL_EDOM);
(gdb) up
#2  0x8048a00 in main (argc=1, argv=0xbffff9bc)
at test_trap.c:94
94   status = gsl_fft_complex_wavetable_alloc (0,
&complex_wavetable);

```

Thus we have found the line that caused the problem. From this point we could also print out the values of other variables such as `complex_wavetable`.

## 48.2 Examining floating point registers

The contents of floating point registers can be examined using the command `info float` (on supported platforms):

```

(gdb) info float
st0: 0xc4018b895aa17a945000 Valid Normal -7.838871e+308
st1: 0x3ff9ea3f50e4d7275000 Valid Normal 0.0285946
st2: 0x3fe790c64ce27dad4800 Valid Normal 6.7415931e-08
st3: 0x3ffaa3ef0df6607d7800 Spec Normal 0.0400229
st4: 0x3c028000000000000000 Valid Normal 4.4501477e-308
st5: 0x3ffef5412c22219d9000 Zero Normal 0.9580257
st6: 0x3fff8000000000000000 Valid Normal 1
st7: 0xc4028b65a1f6d243c800 Valid Normal -1.566206e+309
fctrl: 0x0272 53 bit; NEAR; mask DENOR UNDER LOS;
fstat: 0xb9ba flags 0001; top 7; excep DENOR OVERF UNDER LOS

```

(continues on next page)

(continued from previous page)

```
ftag: 0x3fff
fip: 0x08048b5c
fcs: 0x051a0023
fopoff: 0x08086820
fopsel: 0x002b
```

Individual registers can be examined using the variables \$reg, where reg is the register name:

```
(gdb) p $st1
$1 = 0.02859464454261210347719
```

## 48.3 Handling floating point exceptions

It is possible to stop the program whenever a SIGFPE floating point exception occurs. This can be useful for finding the cause of an unexpected infinity or NaN. The current handler settings can be shown with the command `info signal SIGFPE`:

```
(gdb) info signal SIGFPE
Signal Stop Print Pass to program Description
SIGFPE Yes Yes Yes Arithmetic exception
```

Unless the program uses a signal handler the default setting should be changed so that SIGFPE is not passed to the program, as this would cause it to exit. The command `handle SIGFPE stop nopass` prevents this:

```
(gdb) handle SIGFPE stop nopass
Signal Stop Print Pass to program Description
SIGFPE Yes Yes No Arithmetic exception
```

Depending on the platform it may be necessary to instruct the kernel to generate signals for floating point exceptions. For programs using GSL this can be achieved using the `GSL_IEEE_MODE` environment variable in conjunction with the function `gsl_ieee_env_setup()` as described in *IEEE floating-point arithmetic*:

```
(gdb) set env GSL_IEEE_MODE=double-precision
```

## 48.4 GCC warning options for numerical programs

Writing reliable numerical programs in C requires great care. The following GCC warning options are recommended when compiling numerical programs:

```
gcc -ansi -pedantic -Werror -Wall -W
-Wmissing-prototypes -Wstrict-prototypes
-Wconversion -Wshadow -Wpointer-arith
-Wcast-qual -Wcast-align
-Wwrite-strings -Wnested-externs
-fshort-enums -fno-common -Dinline= -g -O2
```

For details of each option consult the manual *Using and Porting GCC*. The following table gives a brief explanation of what types of errors these options catch.

```
-ansi -pedantic
```

Use ANSI C, and reject any non-ANSI extensions. These flags help in writing portable programs that will compile on other systems.

**-Werror**

Consider warnings to be errors, so that compilation stops. This prevents warnings from scrolling off the top of the screen and being lost. You won't be able to compile the program until it is completely warning-free.

**-Wall**

This turns on a set of warnings for common programming problems. You need `-Wall`, but it is not enough on its own.

**-O2**

Turn on optimization. The warnings for uninitialized variables in `-Wall` rely on the optimizer to analyze the code. If there is no optimization then these warnings aren't generated.

**-W**

This turns on some extra warnings not included in `-Wall`, such as missing return values and comparisons between signed and unsigned integers.

**-Wmissing-prototypes -Wstrict-prototypes**

Warn if there are any missing or inconsistent prototypes. Without prototypes it is harder to detect problems with incorrect arguments.

**-Wconversion**

The main use of this option is to warn about conversions from signed to unsigned integers. For example, `unsigned int x = -1`. If you need to perform such a conversion you can use an explicit cast.

**-Wshadow**

This warns whenever a local variable shadows another local variable. If two variables have the same name then it is a potential source of confusion.

**-Wpointer-arith -Wcast-qual -Wcast-align**

These options warn if you try to do pointer arithmetic for types which don't have a size, such as `void`, if you remove a `const` cast from a pointer, or if you cast a pointer to a type which has a different size, causing an invalid alignment.

**-Wwrite-strings**

This option gives string constants a `const` qualifier so that it will be a compile-time error to attempt to overwrite them.

**-fshort-enums**

This option makes the type of `enum` as short as possible. Normally this makes an `enum` different from an `int`. Consequently any attempts to assign a pointer-to-`int` to a pointer-to-`enum` will generate a cast-alignment warning.

**-fno-common**

This option prevents global variables being simultaneously defined in different object files (you get an error at link time). Such a variable should be defined in one file and referred to in other files with an `extern` declaration.

**-Wnested-externs**

This warns if an `extern` declaration is encountered within a function.

**-Dinline=**

The `inline` keyword is not part of ANSI C. Thus if you want to use `-ansi` with a program which uses inline functions you can use this preprocessor definition to remove the `inline` keywords.

-g

It always makes sense to put debugging symbols in the executable so that you can debug it using `gdb`. The only effect of debugging symbols is to increase the size of the file, and you can use the `strip` command to remove them later if necessary.

## 48.5 References and Further Reading

The following books are essential reading for anyone writing and debugging numerical programs with `gcc` and `gdb`.

- R.M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, ISBN 1882114388
- R.M. Stallman, R.H. Pesch, *Debugging with GDB: The GNU Source-Level Debugger*, Free Software Foundation, ISBN 1882114779

For a tutorial introduction to the GNU C Compiler and related programs, see

- B.J. Gough, <http://www.network-theory.co.uk/gcc/intro/>, 'An Introduction to GCC', Network Theory Ltd, ISBN 0954161793



## CONTRIBUTORS TO GSL

(See the AUTHORS file in the distribution for up-to-date information.)

Mark Galassi

Conceived GSL (with James Theiler) and wrote the design document. Wrote the simulated annealing package and the relevant chapter in the manual.

James Theiler

Conceived GSL (with Mark Galassi). Wrote the random number generators and the relevant chapter in this manual.

Jim Davies

Wrote the statistical routines and the relevant chapter in this manual.

Brian Gough

FFTs, numerical integration, random number generators and distributions, root finding, minimization and fitting, polynomial solvers, complex numbers, physical constants, permutations, vector and matrix functions, histograms, statistics, ieee-utils, revised CBLAS Level 2 & 3, matrix decompositions, eigensystems, cumulative distribution functions, testing, documentation and releases.

Reid Priedhorsky

Wrote and documented the initial version of the root finding routines while at Los Alamos National Laboratory, Mathematical Modeling and Analysis Group.

Gerard Jungman

Special Functions, Series acceleration, ODEs, BLAS, Linear Algebra, Eigensystems, Hankel Transforms.

Patrick Alken

Implementation of nonsymmetric and generalized eigensystems, B-splines, linear and nonlinear least squares, matrix decompositions, associated Legendre functions, running statistics, sparse matrices, and sparse linear algebra.

Mike Booth

Wrote the Monte Carlo library.

Jorma Olavi Tähtinen

Wrote the initial complex arithmetic functions.

Thomas Walter

Wrote the initial heapsort routines and Cholesky decomposition.

Fabrice Rossi

Multidimensional minimization.

Carlo Perassi

Implementation of the random number generators in Knuth's *Seminumerical Algorithms*, 3rd Ed.

Szymon Jaroszewicz

Wrote the routines for generating combinations.

Nicolas Darnis

Wrote the cyclic functions and the initial functions for canonical permutations.

Jason H. Stover

Wrote the major cumulative distribution functions.

Ivo Alxneit

Wrote the routines for wavelet transforms.

Tuomo Keskitalo

Improved the implementation of the ODE solvers and wrote the ode-initval2 routines.

Lowell Johnson

Implementation of the Mathieu functions.

Rhys Ulerich

Wrote the multiset routines.

Pavel Holoborodko

Wrote the fixed order Gauss-Legendre quadrature routines.

Pedro Gonnet

Wrote the CQUAD integration routines.

Thanks to Nigel Lowry for help in proofreading the manual.

The non-symmetric eigensystems routines contain code based on the LAPACK linear algebra library. LAPACK is distributed under the following license:

Copyright (c) 1992-2006 The University of Tennessee. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

(continues on next page)



(continued from previous page)

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR  
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT  
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,  
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT  
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,  
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY  
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT  
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE  
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



## AUTOCONF MACROS

For applications using `autoconf` the standard macro `AC_CHECK_LIB` can be used to link with GSL automatically from a `configure` script. The library itself depends on the presence of a CBLAS and math library as well, so these must also be located before linking with the main `libgsl` file. The following commands should be placed in the `configure.ac` file to perform these tests:

```
AC_CHECK_LIB([m],[cos])
AC_CHECK_LIB([gslcblas],[cblas_dgemm])
AC_CHECK_LIB([gsl],[gsl_blas_dgemm])
```

It is important to check for `libm` and `libgslcblas` before `libgsl`, otherwise the tests will fail. Assuming the libraries are found the output during the configure stage looks like this:

```
checking for cos in -lm... yes
checking for cblas_dgemm in -lgslcblas... yes
checking for gsl_blas_dgemm in -lgsl... yes
```

If the library is found then the tests will define the macros `HAVE_LIBGSL`, `HAVE_LIBGSLCBLAS`, `HAVE_LIBM` and add the options `-lgsl -lgslcblas -lm` to the variable `LIBS`.

The tests above will find any version of the library. They are suitable for general use, where the versions of the functions are not important. An alternative macro is available in the file `gsl.m4` to test for a specific version of the library. To use this macro simply add the following line to your `configure.in` file instead of the tests above:

```
AX_PATH_GSL(GSL_VERSION,
            [action-if-found],
            [action-if-not-found])
```

The argument `GSL_VERSION` should be the two or three digit `major.minor` or `major.minor.micro` version number of the release you require. A suitable choice for `action-if-not-found` is:

```
AC_MSG_ERROR(could not find required version of GSL)
```

Then you can add the variables `GSL_LIBS` and `GSL_CFLAGS` to your `Makefile.am` files to obtain the correct compiler flags. `GSL_LIBS` is equal to the output of the `gsl-config --libs` command and `GSL_CFLAGS` is equal to `gsl-config --cflags` command. For example:

```
libfoo_la_LDFLAGS = -lfoo $(GSL_LIBS) -lgslcblas
```

Note that the macro `AX_PATH_GSL` needs to use the C compiler so it should appear in the `configure.in` file before the macro `AC_LANG_CPLUSPLUS` for programs that use C++.

To test for `inline` the following test should be placed in your `configure.in` file:

```
AC_C_INLINE

if test "$ac_cv_c_inline" != no ; then
  AC_DEFINE(HAVE_INLINE,1)
  AC_SUBST(HAVE_INLINE)
fi
```

and the macro will then be defined in the compilation flags or by including the file `config.h` before any library headers. The following autoconf test will check for `extern inline`:

```
dnl Check for "extern inline", using a modified version
dnl of the test for AC_C_INLINE from acspecific.mt
dnl
AC_CACHE_CHECK([for extern inline], ac_cv_c_extern_inline,
[ac_cv_c_extern_inline=no
AC_TRY_COMPILE([extern $ac_cv_c_inline double foo(double x);
extern $ac_cv_c_inline double foo(double x) { return x+1.0; };
double foo (double x) { return x + 1.0; };],
[ foo(1.0) ],
[ac_cv_c_extern_inline="yes"])
])

if test "$ac_cv_c_extern_inline" != no ; then
  AC_DEFINE(HAVE_INLINE,1)
  AC_SUBST(HAVE_INLINE)
fi
```

The substitution of portability functions can be made automatically if you use autoconf. For example, to test whether the BSD function `hypot()` is available you can include the following line in the configure file `configure.in` for your application:

```
AC_CHECK_FUNCS(hypot)
```

and place the following macro definitions in the file `config.h.in`:

```
/* Substitute gsl_hypot for missing system hypot */

#ifdef HAVE_HYPOT
#define hypot gsl_hypot
#endif
```

The application source files can then use the include command `#include <config.h>` to substitute `gsl_hypot()` for each occurrence of `hypot()` when `hypot()` is not available.

## GSL CBLAS LIBRARY

The prototypes for the low-level CBLAS functions are declared in the file `gsl_cblas.h`. For the definition of the functions consult the documentation available from Netlib (*see BLAS References and Further Reading*).

### 51.1 Level 1

float **cblas\_sdsdot**(const int N, const float alpha, const float \*x, const int incx, const float \*y, const int incy)

double **cblas\_dsdot**(const int N, const float \*x, const int incx, const float \*y, const int incy)

float **cblas\_sdot**(const int N, const float \*x, const int incx, const float \*y, const int incy)

double **cblas\_ddot**(const int N, const double \*x, const int incx, const double \*y, const int incy)

void **cblas\_cdotu\_sub**(const int N, const void \*x, const int incx, const void \*y, const int incy, void \*dotu)

void **cblas\_cdotc\_sub**(const int N, const void \*x, const int incx, const void \*y, const int incy, void \*dotc)

void **cblas\_zdotu\_sub**(const int N, const void \*x, const int incx, const void \*y, const int incy, void \*dotu)

void **cblas\_zdotc\_sub**(const int N, const void \*x, const int incx, const void \*y, const int incy, void \*dotc)

float **cblas\_snrm2**(const int N, const float \*x, const int incx)

float **cblas\_sasum**(const int N, const float \*x, const int incx)

double **cblas\_dnrm2**(const int N, const double \*x, const int incx)

double **cblas\_dasum**(const int N, const double \*x, const int incx)

float **cblas\_scnrm2**(const int N, const void \*x, const int incx)

float **cblas\_scasum**(const int N, const void \*x, const int incx)

double **cblas\_dznrm2**(const int N, const void \*x, const int incx)

double **cblas\_dzasum**(const int N, const void \*x, const int incx)

CBLAS\_INDEX **cblas\_isamax**(const int N, const float \*x, const int incx)

CBLAS\_INDEX **cblas\_idamax**(const int N, const double \*x, const int incx)

CBLAS\_INDEX **cblas\_icamax**(const int N, const void \*x, const int incx)

CBLAS\_INDEX **cblas\_izamax**(const int N, const void \*x, const int incx)

void **cblas\_sswap**(const int N, float \*x, const int incx, float \*y, const int incy)

void **cblas\_scopy**(const int N, const float \*x, const int incx, float \*y, const int incy)

void **cblas\_saxpy**(const int N, const float alpha, const float \*x, const int incx, float \*y, const int incy)

void **cblas\_dswap**(const int N, double \*x, const int incx, double \*y, const int incy)

void **cblas\_dcopy**(const int N, const double \*x, const int incx, double \*y, const int incy)

void **cblas\_daxpy**(const int N, const double alpha, const double \*x, const int incx, double \*y, const int incy)

void **cblas\_cswap**(const int N, void \*x, const int incx, void \*y, const int incy)

void **cblas\_ccopy**(const int N, const void \*x, const int incx, void \*y, const int incy)

void **cblas\_caxpy**(const int N, const void \*alpha, const void \*x, const int incx, void \*y, const int incy)

void **cblas\_zswap**(const int N, void \*x, const int incx, void \*y, const int incy)

void **cblas\_zcopy**(const int N, const void \*x, const int incx, void \*y, const int incy)

void **cblas\_zaxpy**(const int N, const void \*alpha, const void \*x, const int incx, void \*y, const int incy)

void **cblas\_srotg**(float \*a, float \*b, float \*c, float \*s)

void **cblas\_srotmg**(float \*d1, float \*d2, float \*b1, const float b2, float \*P)

void **cblas\_srot**(const int N, float \*x, const int incx, float \*y, const int incy, const float c, const float s)

void **cblas\_srotm**(const int N, float \*x, const int incx, float \*y, const int incy, const float \*P)

void **cblas\_drotg**(double \*a, double \*b, double \*c, double \*s)

void **cblas\_drotmg**(double \*d1, double \*d2, double \*b1, const double b2, double \*P)

void **cblas\_drot**(const int N, double \*x, const int incx, double \*y, const int incy, const double c, const double s)

void **cblas\_drotm**(const int N, double \*x, const int incx, double \*y, const int incy, const double \*P)

void **cblas\_sscal**(const int N, const float alpha, float \*x, const int incx)

void **cblas\_dscal**(const int N, const double alpha, double \*x, const int incx)

void **cblas\_cscal**(const int N, const void \*alpha, void \*x, const int incx)

void **cblas\_zscal**(const int N, const void \*alpha, void \*x, const int incx)

void **cblas\_csscal**(const int N, const float alpha, void \*x, const int incx)

void **cblas\_zdscal**(const int N, const double alpha, void \*x, const int incx)

## 51.2 Level 2

void **cblas\_sgemv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const float alpha, const float \*A, const int lda, const float \*x, const int incx, const float beta, float \*y, const int incy)

void **cblas\_sgbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const float alpha, const float \*A, const int lda, const float \*x, const int incx, const float beta, float \*y, const int incy)

void **cblas\_strmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const float \*A, const int lda, float \*x, const int incx)

void **cblas\_stbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const float \*A, const int lda, float \*x, const int incx)

void **cblas\_stpmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const float \*Ap, float \*x, const int incx)

void **cblas\_strsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const float \*A, const int lda, float \*x, const int incx)

void **cblas\_stbsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const float \*A, const int lda, float \*x, const int incx)

void **cblas\_stpsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const float \*Ap, float \*x, const int incx)

void **cblas\_dgemv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const double alpha, const double \*A, const int lda, const double \*x, const int incx, const double beta, double \*y, const int incy)

void **cblas\_dgbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const double alpha, const double \*A, const int lda, const double \*x, const int incx, const double beta, double \*y, const int incy)

void **cblas\_dtrmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const double \*A, const int lda, double \*x, const int incx)

void **cblas\_dtbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const double \*A, const int lda, double \*x, const int incx)

void **cblas\_dtpmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const double \*Ap, double \*x, const int incx)

void **cblas\_dtrsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const double \*A, const int lda, double \*x, const int incx)

void **cblas\_dtbsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const double \*A, const int lda, double \*x, const int incx)

void **cblas\_dtpsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const double \*Ap, double \*x, const int incx)

void **cblas\_cgemv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)



void **cblas\_cgbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_ctrmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ctbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ctpmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*Ap, void \*x, const int incx)

void **cblas\_ctrsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ctbsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ctpsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*Ap, void \*x, const int incx)

void **cblas\_zgemv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_zgbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_TRANSPOSE TransA, const int M, const int N, const int KL, const int KU, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_ztrmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ztbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ztpmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*Ap, void \*x, const int incx)

void **cblas\_ztrsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ztbsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const int K, const void \*A, const int lda, void \*x, const int incx)

void **cblas\_ztpsv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int N, const void \*Ap, void \*x, const int incx)

void **cblas\_ssymv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const float \*A, const int lda, const float \*x, const int incx, const float beta, float \*y, const int incy)

void **cblas\_ssbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const int K, const float alpha, const float \*A, const int lda, const float \*x, const int incx, const float beta, float \*y, const int incy)

void **cblas\_sspmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const float \*Ap, const float \*x, const int incx, const float beta, float \*y, const int incy)

void **cblas\_sger**(const enum CBLAS\_ORDER order, const int M, const int N, const float alpha, const float \*x, const int incx, const float \*y, const int incy, float \*A, const int lda)

void **cblas\_ssytr**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const float \*x, const int incx, float \*A, const int lda)

void **cblas\_sspr**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const float \*x, const int incx, float \*Ap)

void **cblas\_ssytr2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const float \*x, const int incx, const float \*y, const int incy, float \*A, const int lda)

void **cblas\_sspr2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const float \*x, const int incx, const float \*y, const int incy, float \*A)

void **cblas\_dsymv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const double \*A, const int lda, const double \*x, const int incx, const double beta, double \*y, const int incy)

void **cblas\_dsmbv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const int K, const double alpha, const double \*A, const int lda, const double \*x, const int incx, const double beta, double \*y, const int incy)

---

void **cblas\_dspmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const double \*Ap, const double \*x, const int incx, const double beta, double \*y, const int incy)

void **cblas\_dger**(const enum CBLAS\_ORDER order, const int M, const int N, const double alpha, const double \*x, const int incx, const double \*y, const int incy, double \*A, const int lda)

void **cblas\_dsyr**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const double \*x, const int incx, double \*A, const int lda)

void **cblas\_dspr**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const double \*x, const int incx, double \*Ap)

void **cblas\_dsyr2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const double \*x, const int incx, const double \*y, const int incy, double \*A, const int lda)

void **cblas\_dspr2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const double \*x, const int incx, const double \*y, const int incy, double \*A)

void **cblas\_chemv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_chbmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_chpmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*Ap, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_cgeru**(const enum CBLAS\_ORDER order, const int M, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*A, const int lda)

void **cblas\_cgerc**(const enum CBLAS\_ORDER order, const int M, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*A, const int lda)

void **cblas\_cher**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const void \*x, const int incx, void \*A, const int lda)

void **cblas\_chpr**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const float alpha, const void \*x, const int incx, void \*A)

void **cblas\_cher2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*A, const int lda)

void **cblas\_chpr2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*Ap)

void **cblas\_zhemv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_zhbmw**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_zhpmv**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*Ap, const void \*x, const int incx, const void \*beta, void \*y, const int incy)

void **cblas\_zgeru**(const enum CBLAS\_ORDER order, const int M, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*A, const int lda)

void **cblas\_zgerc**(const enum CBLAS\_ORDER order, const int M, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*A, const int lda)

void **cblas\_zher**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const void \*x, const int incx, void \*A, const int lda)

void **cblas\_zhpr**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const double alpha, const void \*x, const int incx, void \*A)

void **cblas\_zher2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*A, const int lda)

void **cblas\_zhpr2**(const enum CBLAS\_ORDER order, const enum CBLAS\_UPLO Uplo, const int N, const void \*alpha, const void \*x, const int incx, const void \*y, const int incy, void \*Ap)

## 51.3 Level 3

void **cblas\_sgemm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_TRANSPOSE TransB, const int M, const int N, const int K, const float alpha, const float \*A, const int lda, const float \*B, const int ldb, const float beta, float \*C, const int ldc)

void **cblas\_ssymm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const int M, const int N, const float alpha, const float \*A, const int lda, const float \*B, const int ldb, const float beta, float \*C, const int ldc)

void **cblas\_ssyprk**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const float alpha, const float \*A, const int lda, const float beta, float \*C, const int ldc)

void **cblas\_ssyrr2k**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const float alpha, const float \*A, const int lda, const float \*B, const int ldb, const float beta, float \*C, const int ldc)

void **cblas\_strmm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const float alpha, const float \*A, const int lda, float \*B, const int ldb)

void **cblas\_strsm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const float alpha, const float \*A, const int lda, float \*B, const int ldb)

void **cblas\_dgemm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_TRANSPOSE TransB, const int M, const int N, const int K, const double alpha, const double \*A, const int lda, const double \*B, const int ldb, const double beta, double \*C, const int ldc)

void **cblas\_dsymm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const int M, const int N, const double alpha, const double \*A, const int lda, const double \*B, const int ldb, const double beta, double \*C, const int ldc)

void **cblas\_dsyrk**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const double alpha, const double \*A, const int lda, const double beta, double \*C, const int ldc)

void **cblas\_dsyr2k**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const double alpha, const double \*A, const int lda, const double \*B, const int ldb, const double beta, double \*C, const int ldc)

void **cblas\_dtrmm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const double alpha, const double \*A, const int lda, double \*B, const int ldb)

void **cblas\_dtrsm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const double alpha, const double \*A, const int lda, double \*B, const int ldb)

void **cblas\_cgemm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_TRANSPOSE TransB, const int M, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_csymm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const int M, const int N, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_csyrrk**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*beta, void \*C, const int ldc)

void **cblas\_csyrr2k**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_ctrmm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const void \*alpha, const void \*A, const int lda, void \*B, const int ldb)

void **cblas\_ctrsm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const void \*alpha, const void \*A, const int lda, void \*B, const int ldb)

void **cblas\_zgemm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_TRANSPOSE TransB, const int M, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_zsymm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const int M, const int N, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_zsyrrk**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*beta, void \*C, const int ldc)

void **cblas\_zsyrr2k**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_ztrmm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const void \*alpha, const void \*A, const int lda, void \*B, const int ldb)

void **cblas\_ztrsm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE TransA, const enum CBLAS\_DIAG Diag, const int M, const int N, const void \*alpha, const void \*A, const int lda, void \*B, const int ldb)

void **cblas\_chemm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const int M, const int N, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_cherk**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const float alpha, const void \*A, const int lda, const float beta, void \*C, const int ldc)

void **cblas\_cher2k**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const float beta, void \*C, const int ldc)

void **cblas\_zhemm**(const enum CBLAS\_ORDER Order, const enum CBLAS\_SIDE Side, const enum CBLAS\_UPLO Uplo, const int M, const int N, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const void \*beta, void \*C, const int ldc)

void **cblas\_zherk**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const double alpha, const void \*A, const int lda, const double beta, void \*C, const int ldc)

void **cblas\_zher2k**(const enum CBLAS\_ORDER Order, const enum CBLAS\_UPLO Uplo, const enum CBLAS\_TRANSPOSE Trans, const int N, const int K, const void \*alpha, const void \*A, const int lda, const void \*B, const int ldb, const double beta, void \*C, const int ldc)

void **cblas\_xerbla**(int p, const char \*rout, const char \*form, ...)

## 51.4 Examples

The following program computes the product of two matrices using the Level-3 BLAS function SGEMM,

$$\begin{pmatrix} 0.11 & 0.12 & 0.13 \\ 0.21 & 0.22 & 0.23 \end{pmatrix} \begin{pmatrix} 1011 & 1012 \\ 1021 & 1022 \\ 1031 & 1032 \end{pmatrix} = \begin{pmatrix} 367.76 & 368.12 \\ 674.06 & 674.72 \end{pmatrix}$$

The matrices are stored in row major order but could be stored in column major order if the first argument of the call to `cblas_sgemm()` was changed to `CblasColMajor`.

```
#include <stdio.h>
#include <gsl/gsl_cblas.h>

int
main (void)
{
    int lda = 3;

    float A[] = { 0.11, 0.12, 0.13,
                  0.21, 0.22, 0.23 };

    int ldb = 2;

    float B[] = { 1011, 1012,
                  1021, 1022,
                  1031, 1032 };
}
```

(continues on next page)

(continued from previous page)

```
int ldc = 2;

float C[] = { 0.00, 0.00,
             0.00, 0.00 };

/* Compute C = A B */

cblas_sgemm (CblasRowMajor,
            CblasNoTrans, CblasNoTrans, 2, 2, 3,
            1.0, A, lda, B, ldb, 0.0, C, ldc);

printf ("[ %g, %g\n", C[0], C[1]);
printf (" %g, %g ]\n", C[2], C[3]);

return 0;
}
```

To compile the program use the following command line:

```
$ gcc -Wall demo.c -lgslcblas
```

There is no need to link with the main library `-lgsl` in this case as the CBLAS library is an independent unit. Here is the output from the program,

```
[ 367.76, 368.12
 674.06, 674.72 ]
```



## GNU GENERAL PUBLIC LICENSE

GNU GENERAL PUBLIC LICENSE  
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <<http://fsf.org/>>  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

### Preamble

The GNU General Public License is a free, copyleft license for  
software and other kinds of works.

The licenses for most software and other practical works are designed  
to take away your freedom to share and change the works. By contrast,  
the GNU General Public License is intended to guarantee your freedom to  
share and change all versions of a program--to make sure it remains free  
software for all its users. We, the Free Software Foundation, use the  
GNU General Public License for most of our software; it applies also to  
any other work released this way by its authors. You can apply it to  
your programs, too.

When we speak of free software, we are referring to freedom, not  
price. Our General Public Licenses are designed to make sure that you  
have the freedom to distribute copies of free software (and charge for  
them if you wish), that you receive source code or can get it if you  
want it, that you can change the software or use pieces of it in new  
free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you  
these rights or asking you to surrender the rights. Therefore, you have  
certain responsibilities if you distribute copies of the software, or if  
you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether  
gratis or for a fee, you must pass on to the recipients the same  
freedoms that you received. You must make sure that they, too, receive  
or can get the source code. And you must show them these terms so they  
know their rights.

Developers that use the GNU GPL protect your rights with two steps:

(continues on next page)

(continued from previous page)

(1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

#### TERMS AND CONDITIONS

##### 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without

(continues on next page)

(continued from previous page)

permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

#### 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those

(continues on next page)

subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

## 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

## 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

## 4. Conveying Verbatim Copies.

(continued from previous page)

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

#### 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

#### 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the

(continues on next page)

(continued from previous page)

machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product,

(continues on next page)

(continued from previous page)

doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

#### 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

(continues on next page)

(continued from previous page)

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions;

(continues on next page)



(continued from previous page)

the above requirements apply either way.

#### 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

#### 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

#### 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that

(continues on next page)

(continued from previous page)

transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

#### 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the

(continues on next page)

(continued from previous page)

covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

#### 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

#### 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the

(continues on next page)

combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

(continued from previous page)

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

#### END OF TERMS AND CONDITIONS

#### How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>  
Copyright (C) <year> <name of author>
```

```
This program is free software: you can redistribute it and/or modify  
it under the terms of the GNU General Public License as published by  
the Free Software Foundation, either version 3 of the License, or  
(at your option) any later version.
```

```
This program is distributed in the hope that it will be useful,  
but WITHOUT ANY WARRANTY; without even the implied warranty of  
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License  
along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>  
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.  
This is free software, and you are welcome to redistribute it  
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school,

(continues on next page)

(continued from previous page)

if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

## GNU FREE DOCUMENTATION LICENSE

GNU Free Documentation License  
Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.  
<<http://fsf.org/>>

Everyone **is** permitted to copy **and** distribute verbatim copies  
of this license document, but changing it **is not** allowed.

### 0. PREAMBLE

The purpose of this License **is** to make a manual, textbook, **or** other  
functional **and** useful document "**free**" **in** the sense of freedom: to  
assure everyone the effective freedom to copy **and** redistribute it,  
**with or** without modifying it, either commercially **or** noncommercially.  
Secondarily, this License preserves **for** the author **and** publisher a way  
to get credit **for** their work, **while not** being considered responsible  
**for** modifications made by others.

This License **is** a kind of "**copyleft**", which means that derivative  
works of the document must themselves be free **in** the same sense. It  
complements the GNU General Public License, which **is** a copyleft  
license designed **for** free software.

We have designed this License **in** order to use it **for** manuals **for** free  
software, because free software needs free documentation: a free  
program should come **with** manuals providing the same freedoms that the  
software does. But this License **is not** limited to software manuals;  
it can be used **for any** textual work, regardless of subject matter **or**  
whether it **is** published **as** a printed book. We recommend this License  
principally **for** works whose purpose **is** instruction **or** reference.

### 1. APPLICABILITY AND DEFINITIONS

This License applies to **any** manual **or** other work, **in any** medium, that  
contains a notice placed by the copyright holder saying it can be  
distributed under the terms of this License. Such a notice grants a  
world-wide, royalty-free license, unlimited **in** duration, to use that

(continues on next page)

work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats



(continued from previous page)

include proprietary formats that can be read **and** edited only by proprietary word processors, SGML **or** XML **for** which the DTD **and/or** processing tools are **not** generally available, **and** the machine-generated HTML, PostScript **or** PDF produced by some word processors **for** output purposes only.

The "Title Page" means, **for** a printed book, the title page itself, plus such following pages **as** are needed to hold, legibly, the material this License requires to appear **in** the title page. For works **in** formats which do **not** have any title page **as** such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person **or** entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either **is** precisely XYZ **or** contains XYZ **in** parentheses following text that translates XYZ **in** another language. (Here XYZ stands **for** a specific section name mentioned below, such **as** "Acknowledgements", "Dedications", "Endorsements", **or** "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers **next** to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference **in** this License, but only **as** regards disclaiming warranties: any other implication that these Warranty Disclaimers may have **is** void **and** has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy **and** distribute the Document **in** any medium, either commercially **or** noncommercially, provided that this License, the copyright notices, **and** the license notice saying this License applies to the Document are reproduced **in** all copies, **and** that you add no other conditions whatsoever to those of this License. You may **not** use technical measures to obstruct **or** control the reading **or** further copying of the copies you make **or** distribute. However, you may accept compensation **in** exchange **for** copies. If you distribute a large enough number of copies you must also follow the conditions **in** section 3.

You may also lend copies, under the same conditions stated above, **and** you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (**or** copies **in** media that commonly have printed covers) of the Document, numbering more than 100, **and** the Document's license notice requires Cover Texts, you must enclose the

(continues on next page)

(continued from previous page)

copies **in** covers that carry, clearly **and** legibly, **all** these Cover Texts: Front-Cover Texts on the front cover, **and** Back-Cover Texts on the back cover. Both covers must also clearly **and** legibly identify you **as** the publisher of these copies. The front cover must present the full title **with all** words of the title equally prominent **and** visible. You may add other material on the covers **in** addition. Copying **with** changes limited to the covers, **as long as** they preserve the title of the Document **and** satisfy these conditions, can be treated **as** verbatim copying **in** other respects.

If the required texts **for** either cover are too voluminous to fit legibly, you should put the first ones listed (**as many as** fit reasonably) on the actual cover, **and continue** the rest onto adjacent pages.

If you publish **or** distribute Opaque copies of the Document numbering more than **100**, you must either include a machine-readable Transparent copy along **with** each Opaque copy, **or** state **in or with** each Opaque copy a computer-network location **from which** the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies **in** quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly **or** through your agents **or** retailers) of that edition to the public.

It **is** requested, but **not** required, that you contact the authors of the Document well before redistributing **any** large number of copies, to give them a chance to provide you **with** an updated version of the Document.

#### 4. MODIFICATIONS

You may copy **and** distribute a Modified Version of the Document under the conditions of sections **2 and 3** above, provided that you release the Modified Version under precisely this License, **with** the Modified Version filling the role of the Document, thus licensing distribution **and** modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things **in** the Modified Version:

- A. Use **in** the Title Page (**and** on the covers, **if any**) a title distinct **from that** of the Document, **and from those** of previous versions (which should, **if there were any**, be listed **in** the History section of the Document). You may use the same title **as** a previous version **if** the original publisher of that version gives permission.
- B. List on the Title Page, **as** authors, one **or** more persons **or** entities responsible **for** authorship of the modifications **in** the Modified Version, together **with** at least five of the principal authors of the Document (**all** of its principal authors, **if** it has fewer than five),

(continues on next page)

(continued from previous page)

- unless they release you **from this** requirement.
- C. State on the Title page the name of the publisher of the Modified Version, **as** the publisher.
  - D. Preserve **all** the copyright notices of the Document.
  - E. Add an appropriate copyright notice **for** your modifications adjacent to the other copyright notices.
  - F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, **in** the form shown **in** the Addendum below.
  - G. Preserve **in** that license notice the full lists of Invariant Sections **and** required Cover Texts given **in** the Document's **license notice**.
  - H. Include an unaltered copy of this License.
  - I. Preserve the section Entitled "History", Preserve its Title, **and** add to it an item stating at least the title, year, new authors, **and** publisher of the Modified Version **as** given on the Title Page. If there **is** no section Entitled "History" **in** the Document, create one stating the title, year, authors, **and** publisher of the Document **as** given on its Title Page, then add an item describing the Modified Version **as** stated **in** the previous sentence.
  - J. Preserve the network location, **if any**, given **in** the Document **for** public access to a Transparent copy of the Document, **and** likewise the network locations given **in** the Document **for** previous versions it was based on. These may be placed **in** the "History" section. You may omit a network location **for** a work that was published at least four years before the Document itself, **or if** the original publisher of the version it refers to gives permission.
  - K. For **any** section Entitled "Acknowledgements" **or** "Dedications", Preserve the Title of the section, **and** preserve **in** the section **all** the substance **and** tone of each of the contributor acknowledgements **and/or** dedications given therein.
  - L. Preserve **all** the Invariant Sections of the Document, unaltered **in** their text **and in** their titles. Section numbers **or** the equivalent are **not** considered part of the section titles.
  - M. Delete **any** section Entitled "Endorsements". Such a section may **not** be included **in** the Modified Version.
  - N. Do **not** retitle **any** existing section to be Entitled "Endorsements" **or** to conflict **in** title **with any** Invariant Section.
  - O. Preserve **any** Warranty Disclaimers.

If the Modified Version includes new front-matter sections **or** appendices that qualify **as** Secondary Sections **and** contain no material copied **from the** Document, you may at your option designate some **or all** of these sections **as** invariant. To do this, add their titles to the **list** of Invariant Sections **in** the Modified Version's **license notice**. These titles must be distinct **from any** other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--**for** example, statements of peer review **or** that the text has been approved by an organization **as** the authoritative definition of a standard.

(continues on next page)

(continued from previous page)

You may add a passage of up to five words **as** a Front-Cover Text, **and** a passage of up to 25 words **as** a Back-Cover Text, to the end of the **list** of Cover Texts **in** the Modified Version. Only one passage of Front-Cover Text **and** one of Back-Cover Text may be added by (**or** through arrangements made by) **any** one entity. If the Document already includes a cover text **for** the same cover, previously added by you **or** by arrangement made by the same entity you are acting on behalf of, you may **not** add another; but you may replace the old one, on explicit permission **from the** previous publisher that added the old one.

The author(s) **and** publisher(s) of the Document do **not** by this License give permission to use their names **for** publicity **for or** to **assert or** imply endorsement of **any** Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document **with** other documents released under this License, under the terms defined **in** section 4 above **for** modified versions, provided that you include **in** the combination **all** of the Invariant Sections of **all** of the original documents, unmodified, **and list** them **all as** Invariant Sections of your combined work **in** its license notice, **and** that you preserve **all** their Warranty Disclaimers.

The combined work need only contain one copy of this License, **and** multiple identical Invariant Sections may be replaced **with** a single copy. If there are multiple Invariant Sections **with** the same name but different contents, make the title of each such section unique by adding at the end of it, **in** parentheses, the name of the original author **or** publisher of that section **if** known, **or else** a unique number. Make the same adjustment to the section titles **in** the **list** of Invariant Sections **in** the license notice of the combined work.

In the combination, you must combine **any** sections Entitled "History" **in** the various original documents, forming one section Entitled "History"; likewise combine **any** sections Entitled "Acknowledgements", **and any** sections Entitled "Dedications". You must delete **all** sections Entitled "Endorsements".

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document **and** other documents released under this License, **and** replace the individual copies of this License **in** the various documents **with** a single copy that **is** included **in** the collection, provided that you follow the rules of this License **for** verbatim copying of each of the documents **in all** other respects.

You may extract a single document **from such** a collection, **and** distribute it individually under this License, provided you insert a copy of this License into the extracted document, **and** follow this

(continues on next page)

(continued from previous page)

License **in all** other respects regarding verbatim copying of that document.

#### 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document **or** its derivatives **with** other separate **and** independent documents **or** works, **in or** on a volume of a storage **or** distribution medium, **is** called an "aggregate" **if** the copyright resulting **from the** compilation **is not** used to limit the legal rights of the compilation's users **beyond what the individual works permit**. When the Document **is** included **in** an aggregate, this License does **not** apply to the other works **in** the aggregate which are **not** themselves derivative works of the Document.

If the Cover Text requirement of section 3 **is** applicable to these copies of the Document, then **if** the Document **is** less than one half of the entire aggregate, the Document's Cover Texts **may be placed on** covers that bracket the Document within the aggregate, **or** the electronic equivalent of covers **if** the Document **is in** electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

#### 8. TRANSLATION

Translation **is** considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections **with** translations requires special permission **from their** copyright holders, but you may include translations of some **or all** Invariant Sections **in** addition to the original versions of these Invariant Sections. You may include a translation of this License, **and all** the license notices **in** the Document, **and any** Warranty Disclaimers, provided that you also include the original English version of this License **and** the original versions of those notices **and** disclaimers. In case of a disagreement between the translation **and** the original version of this License **or** a notice **or** disclaimer, the original version will prevail.

If a section **in** the Document **is** Entitled "Acknowledgements", "Dedications", **or** "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

#### 9. TERMINATION

You may **not** copy, modify, sublicense, **or** distribute the Document **except as** expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, **or** distribute it **is** void, **and** will automatically terminate your rights under this License.

(continues on next page)

(continued from previous page)

However, **if** you cease **all** violation of this License, then your license **from a** particular copyright holder **is** reinstated (a) provisionally, unless **and** until the copyright holder explicitly **and finally** terminates your license, **and** (b) permanently, **if** the copyright holder fails to notify you of the violation by some reasonable means prior to **60** days after the cessation.

Moreover, your license **from a** particular copyright holder **is** reinstated permanently **if** the copyright holder notifies you of the violation by some reasonable means, this **is** the first time you have received notice of violation of this License (**for any work**) **from that** copyright holder, **and** you cure the violation prior to **30** days after your receipt of the notice.

Termination of your rights under this section does **not** terminate the licenses of parties who have received copies **or** rights **from you** under this License. If your rights have been terminated **and not** permanently reinstated, receipt of a copy of some **or all** of the same material does **not** give you **any** rights to use it.

#### 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License **from time** to time. Such new versions will be similar **in** spirit to the present version, but may differ **in** detail to address new problems **or** concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License **is** given a distinguishing version number. If the Document specifies that a particular numbered version of this License "**or any later version**" applies to it, you have the option of following the terms **and** conditions either of that specified version **or** of **any** later version that has been published (**not as** a draft) by the Free Software Foundation. If the Document does **not** specify a version number of this License, you may choose **any** version ever published (**not as** a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's **public statement of acceptance of a** version permanently authorizes you to choose that version **for** the Document.

#### 11. RELICENSING

"Massive Multiauthor Collaboration Site" (**or "MMC Site"**) means **any** World Wide Web server that publishes copyrightable works **and** also provides prominent facilities **for** anybody to edit those works. A public wiki that anybody can edit **is** an example of such a server. A "Massive Multiauthor Collaboration" (**or "MMC"**) contained **in** the site means **any set** of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0

(continues on next page)

(continued from previous page)

license published by Creative Commons Corporation, a **not-for-profit** corporation **with** a principal place of business **in** San Francisco, California, **as well as** future copyleft versions of that license published by that same organization.

"**Incorporate**" means to publish **or** republish a Document, **in whole or in part, as** part of another Document.

An MMC **is "eligible for relicensing" if it is** licensed under this License, **and if all** works that were first published under this License somewhere other than this MMC, **and** subsequently incorporated **in whole or in part** into the MMC, (1) had no cover texts **or** invariant sections, **and** (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained **in** the site under CC-BY-SA on the same site at **any** time before August 1, 2009, provided the MMC **is** eligible **for** relicensing.

ADDENDUM: How to use this License **for** your documents

To use this License **in** a document you have written, include a copy of the License **in** the document **and** put the following copyright **and** license notices just after the title page:

```
Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts **and** Back-Cover Texts, replace the "**with...Texts.**" line **with** this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, **or** some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples **in** parallel under your choice of free software license, such **as** the GNU General Public License, to permit their use **in** free software.

- genindex





## Symbols

\$, shell prompt, 3  
 2D histograms, 341  
 2D random direction vector, 265  
 3D random direction vector, 265  
 3-j symbols, 47  
 6-j symbols, 47  
 9-j symbols, 47

## A

acceleration of series, 418  
 acosh, 18  
 Adams method, 385  
 Adaptive step-size control, differential equations, 386  
 Ai(x), 37  
 Airy functions, 37  
 Akima splines, 396  
 aliasing of arrays, 11  
 alternative optimized functions, 9  
 AMAX, Level-1 BLAS, 131  
 Angular Mathieu Functions, 75  
 angular reduction, 79  
 ANSI C, use of, 4  
 Apell symbol, see Pochhammer symbol, 60  
 approximate comparison of floating point numbers, 21  
 arctangent integral, 56  
 argument of complex number, 25  
 arithmetic exceptions, 639  
 asinh, 18  
 astronomical constants, 628  
 ASUM, Level-1 BLAS, 131  
 atanh, 18  
 atomic physics, constants, 628  
 autoconf, using with GSL, 651  
 AXPY, Level-1 BLAS, 132

## B

B-spline wavelets, 424  
 Bader and Deuflhard, Bulirsch-Stoer method., 385

balancing matrices, 167  
 banded Cholesky Decomposition, 166  
 banded general matrices, 164  
 banded LDLT decomposition, 167  
 banded LU Decomposition, 165  
 banded matrices, 163  
 banded symmetric matrices, 164  
 Basic Linear Algebra Subroutines (BLAS), 127, 654  
 basis splines, allocating, 568  
 basis splines, B-splines, 565  
 basis splines, breakpoints, 568  
 basis splines, derivatives, 570  
 basis splines, evaluation, 570  
 basis splines, examples, 576  
 basis splines, Gram matrix, 574  
 basis splines, Greville abscissae, 576  
 basis splines, integration, 571  
 basis splines, interpolation, 575  
 basis splines, knots, 568  
 basis splines, least squares, 571  
 basis splines, Marsden-Schoenberg points, 576  
 basis splines, overview, 567  
 basis splines, properties, 569  
 BDF method, 386  
 Bernoulli trial, random variates, 272  
 Bessel functions, 38  
 Bessel Functions, Fractional Order, 43  
 best-fit parameters, covariance, 535  
 Beta distribution, 262  
 Beta function, 61  
 Beta function, incomplete normalized, 61  
 BFGS algorithm, minimization, 474  
 Bi(x), 37  
 bias, IEEE format, 637  
 bicubic interpolation, 404  
 bidiagonalization of real matrices, 160  
 bilinear interpolation, 404  
 binning data, 333  
 Binomial random variates, 273  
 biorthogonal wavelets, 424  
 bisection algorithm for finding roots, 438

Bivariate Gaussian distribution, 241, 242  
BLAS, 127  
BLAS, Low-level C interface, 654  
BLAS, sparse, 617  
blocks, 83  
bounds checking, extension to GCC, 86  
breakpoints, 643  
Brent's method for finding minima, 450  
Brent's method for finding roots, 439  
Broyden algorithm for multidimensional roots, 462  
BSD random number generator, 223  
bug-gsl, 2  
bug-gsl mailing list, 2  
bugs, 2  
Bulirsch-Stoer method, 385

## C

C extensions, compatible use of, 4  
C++, compatibility, 11  
C99, inline keyword, 7  
Carlson forms of Elliptic integrals, 50  
Cash-Karp, Runge-Kutta method, 385  
Cauchy distribution, 246  
Cauchy principal value, by numerical quadrature, 203  
CBLAS, 127  
CBLAS, Low-level interface, 654  
cblas\_caxpy (C function), 656  
cblas\_ccopy (C function), 656  
cblas\_cdotc\_sub (C function), 655  
cblas\_cdotu\_sub (C function), 655  
cblas\_cgbmv (C function), 658  
cblas\_cgemm (C function), 663  
cblas\_cgemv (C function), 658  
cblas\_cgerc (C function), 661  
cblas\_cgeru (C function), 661  
cblas\_chbmv (C function), 661  
cblas\_chemm (C function), 664  
cblas\_chemv (C function), 661  
cblas\_cher (C function), 661  
cblas\_cher2 (C function), 661  
cblas\_cher2k (C function), 665  
cblas\_cherk (C function), 664  
cblas\_chpmv (C function), 661  
cblas\_chpr (C function), 661  
cblas\_chpr2 (C function), 661  
cblas\_cscal (C function), 657  
cblas\_csscal (C function), 657  
cblas\_cswap (C function), 656  
cblas\_csymm (C function), 663  
cblas\_csy2k (C function), 664  
cblas\_csyk (C function), 663  
cblas\_ctbmv (C function), 659

cblas\_ctbsv (C function), 659  
cblas\_ctpmv (C function), 659  
cblas\_ctorsv (C function), 659  
cblas\_ctrmm (C function), 664  
cblas\_ctrmv (C function), 659  
cblas\_ctrsm (C function), 664  
cblas\_ctrsv (C function), 659  
cblas\_dasum (C function), 655  
cblas\_daxpy (C function), 656  
cblas\_dcopy (C function), 656  
cblas\_ddot (C function), 655  
cblas\_dgbmv (C function), 658  
cblas\_dgemm (C function), 663  
cblas\_dgemv (C function), 658  
cblas\_dger (C function), 661  
cblas\_dnorm2 (C function), 655  
cblas\_drot (C function), 657  
cblas\_drotg (C function), 657  
cblas\_drotm (C function), 657  
cblas\_drotmg (C function), 657  
cblas\_dsbmv (C function), 660  
cblas\_dscal (C function), 657  
cblas\_dsdot (C function), 655  
cblas\_dspmv (C function), 660  
cblas\_dspr (C function), 661  
cblas\_dspr2 (C function), 661  
cblas\_dswap (C function), 656  
cblas\_dsymm (C function), 663  
cblas\_dsymv (C function), 660  
cblas\_dsyr (C function), 661  
cblas\_dsyr2 (C function), 661  
cblas\_dsyr2k (C function), 663  
cblas\_dsyk (C function), 663  
cblas\_dtbmv (C function), 658  
cblas\_dtbsv (C function), 658  
cblas\_dtpmv (C function), 658  
cblas\_dtpsv (C function), 658  
cblas\_dtrmm (C function), 663  
cblas\_dtrmv (C function), 658  
cblas\_dtrsm (C function), 663  
cblas\_dtrsv (C function), 658  
cblas\_dzasum (C function), 656  
cblas\_dznrm2 (C function), 655  
cblas\_icamax (C function), 656  
cblas\_idamax (C function), 656  
cblas\_isamax (C function), 656  
cblas\_izamax (C function), 656  
cblas\_sasum (C function), 655  
cblas\_saxpy (C function), 656  
cblas\_scasum (C function), 655  
cblas\_scnrm2 (C function), 655  
cblas\_scopy (C function), 656  
cblas\_sdot (C function), 655  
cblas\_sdsdot (C function), 655

- cblas\_sgbmv (*C function*), 657
- cblas\_sgemm (*C function*), 662
- cblas\_sgemv (*C function*), 657
- cblas\_sger (*C function*), 660
- cblas\_snrm2 (*C function*), 655
- cblas\_srot (*C function*), 656
- cblas\_srotg (*C function*), 656
- cblas\_srotm (*C function*), 656
- cblas\_srotmg (*C function*), 656
- cblas\_ssbmv (*C function*), 660
- cblas\_sscal (*C function*), 657
- cblas\_sspmv (*C function*), 660
- cblas\_sspr (*C function*), 660
- cblas\_sspr2 (*C function*), 660
- cblas\_sswap (*C function*), 656
- cblas\_ssymm (*C function*), 662
- cblas\_ssymv (*C function*), 660
- cblas\_ssyr (*C function*), 660
- cblas\_ssyr2 (*C function*), 660
- cblas\_ssyr2k (*C function*), 662
- cblas\_ssyrk (*C function*), 662
- cblas\_stbmv (*C function*), 657
- cblas\_stbsv (*C function*), 658
- cblas\_stpmv (*C function*), 657
- cblas\_stpsv (*C function*), 658
- cblas\_strmm (*C function*), 663
- cblas\_strmv (*C function*), 657
- cblas\_strsm (*C function*), 663
- cblas\_strsv (*C function*), 657
- cblas\_xerbla (*C function*), 665
- cblas\_zaxpy (*C function*), 656
- cblas\_zcopy (*C function*), 656
- cblas\_zdotc\_sub (*C function*), 655
- cblas\_zdotu\_sub (*C function*), 655
- cblas\_zdscal (*C function*), 657
- cblas\_zgbmv (*C function*), 659
- cblas\_zgemm (*C function*), 664
- cblas\_zgemv (*C function*), 659
- cblas\_zgerc (*C function*), 662
- cblas\_zgeru (*C function*), 662
- cblas\_zhbmvm (*C function*), 662
- cblas\_zhemm (*C function*), 665
- cblas\_zhemv (*C function*), 662
- cblas\_zher (*C function*), 662
- cblas\_zher2 (*C function*), 662
- cblas\_zher2k (*C function*), 665
- cblas\_zherk (*C function*), 665
- cblas\_zhpmv (*C function*), 662
- cblas\_zhpr (*C function*), 662
- cblas\_zhpr2 (*C function*), 662
- cblas\_zscal (*C function*), 657
- cblas\_zswap (*C function*), 656
- cblas\_zsymm (*C function*), 664
- cblas\_zsyr2k (*C function*), 664
- cblas\_zsyrk (*C function*), 664
- cblas\_ztbmv (*C function*), 659
- cblas\_ztbsv (*C function*), 660
- cblas\_ztpmv (*C function*), 659
- cblas\_ztpsv (*C function*), 660
- cblas\_ztrmm (*C function*), 664
- cblas\_ztrmv (*C function*), 659
- cblas\_ztrsm (*C function*), 664
- cblas\_ztrsv (*C function*), 659
- CDFs, cumulative distribution functions, 233
- ce(q,x), Mathieu function, 75
- Chebyshev series, 413
- checking combination for validity, 114
- checking multiset for validity, 118
- checking permutation for validity, 108
- Chi(x), 55
- Chi-squared distribution, 256
- Cholesky decomposition, 154
- Cholesky decomposition, banded, 166
- Cholesky decomposition, modified, 157
- Cholesky decomposition, pivoted, 156
- Cholesky decomposition, square root free, 157
- Ci(x), 56
- Clausen functions, 45
- Clenshaw-Curtis quadrature, 200
- CMRG, combined multiple recursive random number generator, 221
- code reuse in applications, 11
- combinations, 112
- combinatorial factor C(m, 59
- combinatorial optimization, 366
- comparison functions, definition, 123
- compatibility, 4
- compiling programs, include paths, 5
- compiling programs, library paths, 6
- complementary incomplete Gamma function, 60
- complete Fermi-Dirac integrals, 57
- complete orthogonal decomposition, 151
- complex arithmetic, 25
- complex cosine function, special functions, 78
- Complex Gamma function, 59
- complex hermitian matrix, eigensystem, 172
- complex log sine function, special functions, 78
- complex numbers, 21
- complex sinc function, special functions, 78
- complex sine function, special functions, 78
- confluent hypergeometric function, 67
- confluent hypergeometric functions, 65
- conical functions, 68
- Conjugate gradient algorithm, minimization, 474
- conjugate of complex number, 25

constant matrix, 95  
constants, fundamental, 627  
constants, mathematical (*defined as macros*), 17  
constants, physical, 626  
constants, prefixes, 633  
contacting the GSL developers, 3  
conventions, used in manual, 3  
convergence, accelerating a series, 418  
conversion of units, 626  
cooling schedule, 367  
COPY, Level-1 BLAS, 131  
correlation, of two datasets, 290  
cosine function, special functions, 78  
cosine of complex number, 26  
cost function, 366  
Coulomb wave functions, 45  
coupling coefficients, 47  
covariance matrix, from linear regression, 482  
covariance matrix, linear fits, 481  
covariance matrix, nonlinear fits, 535  
covariance, of two datasets, 289  
cquad, doubly-adaptive integration, 206  
CRAY random number generator, RANF, 224  
cubic equation, solving, 31  
cubic splines, 396  
cumulative distribution functions (*CDFs*), 233  
Cylindrical Bessel Functions, 38

## D

Daubechies wavelets, 423  
Dawson function, 48  
DAXPY, Level-1 BLAS, 132  
debugging numerical programs, 643  
Debye functions, 48  
denormalized form, IEEE format, 637  
deprecated functions, 11  
derivatives, calculating numerically, 409  
determinant of a matrix, by LU decomposition, 142  
Deuflhard and Bader, Bulirsch-Stoer method., 385  
DFTs, see FFT, 182  
diagonal, of a matrix, 99  
differential equations, initial value problems, 381  
differentiation of functions, numeric, 409  
digamma function, 76  
dilogarithm, 49  
direction vector, random 2D, 265  
direction vector, random 3D, 265  
direction vector, random N-dimensional, 265  
Dirichlet distribution, 269  
discontinuities, in ODE systems, 388

Discrete Fourier Transforms, see FFT, 182  
discrete Hankel transforms, 429  
Discrete Newton algorithm for multidimensional roots, 462  
Discrete random numbers, 270  
Discrete random numbers, preprocessing, 270  
divided differences, polynomials, 29  
division by zero, IEEE exceptions, 639  
Dogleg algorithm, 523  
Dogleg algorithm, double, 524  
dollar sign \$, shell prompt, 3  
DOT, Level-1 BLAS, 130  
double Dogleg algorithm, 524  
double factorial, 59  
double precision, IEEE format, 637  
downloading GSL, 2  
DWT initialization, 423  
DWT, mathematical definition, 423  
DWT, one dimensional, 424  
DWT, see wavelet transforms, 422  
DWT, two dimensional, 425

## E

e, defined as a macro, 17  
E1(x), 55  
E2(x), 55  
Ei(x), 55  
eigenvalues and eigenvectors, 170  
elementary functions, 16  
elementary operations, 49  
elliptic functions (*Jacobi*), 52  
elliptic integrals, 50  
energy function, 366  
energy, units of, 631  
erf(x), 52  
erfc(x), 52  
Erlang distribution, 252  
error codes, 14  
error codes, reserved, 13  
error function, 52  
error handlers, 14  
error handling, 12  
error handling macros, 15  
estimated standard deviation, 285  
estimated variance, 285  
estimation, location, 293  
estimation, scale, 294  
Eta Function, 81  
euclidean distance function, hypot, 18  
euclidean distance function, hypot3, 18  
Euler's constant, defined as a macro, 17  
evaluation of polynomials, 29  
evaluation of polynomials, in divided difference form, 29

examples, conventions used in, 3  
 exceptions, C++, 11  
 exceptions, floating point, 645  
 exceptions, IEEE arithmetic, 639  
 exchanging permutation elements, 108  
 exp, 53  
 expm1, 18  
 exponent, IEEE format, 637  
 Exponential distribution, 243  
 exponential function, 53  
 exponential integrals, 55  
 Exponential power distribution, 245  
 exponential, difference from 1 computed  
     accurately, 18  
 exponentiation of complex number, 26  
 extern inline, 7

## F

F-distribution, 258  
 factorial, 59  
 factorization of matrices, 139  
 false position algorithm for finding roots,  
     439  
 Fast Fourier Transforms, see FFT, 182  
 Fehlberg method, differential equations, 385  
 Fermi-Dirac function, 57  
 FFT, 182  
 FFT mathematical definition, 183  
 FFT of complex data, mixed-radix algorithm,  
     186  
 FFT of complex data, radix-2 algorithm, 185  
 FFT of real data, 191  
 FFT of real data, mixed-radix algorithm, 193  
 FFT of real data, radix-2 algorithm, 191  
 FFT, complex data, 184  
 finding minima, 445  
 finding roots, 432  
 finding zeros, 432  
 fits, multi-parameter linear, 483  
 fitting, 480  
 fitting, using Chebyshev polynomials, 413  
 Fj(x), Fermi-Dirac integral, 57  
 Fj(x,b), incomplete Fermi-Dirac integral, 58  
 flat distribution, 254  
 Fletcher-Reeves conjugate gradient  
     algorithm, minimization, 474  
 floating point exceptions, 645  
 floating point numbers, approximate  
     comparison, 21  
 floating point registers, 644  
 force and energy, 633  
 Fortran range checking, equivalent in gcc, 86  
 Four-tap Generalized Feedback Shift  
     Register, 222

Fourier integrals, numerical, 205  
 Fourier Transforms, see FFT, 182  
 Fractional Order Bessel Functions, 43  
 free software, explanation of, 1  
 frexp, 19  
 functions, numerical differentiation, 409  
 fundamental constants, 627

## G

Gamma distribution, 252  
 gamma functions, 58  
 Gastwirth estimator, 294  
 Gauss-Kronrod quadrature, 200  
 Gaussian distribution, 237  
 Gaussian distribution, bivariate, 241, 242  
 Gaussian Tail distribution, 239  
 gcc extensions, range-checking, 86  
 gcc warning options, 645  
 gdb, 643  
 Gegenbauer functions, 61  
 GEMM, Level-3 BLAS, 135  
 GEMV, Level-2 BLAS, 133  
 general polynomial equations, solving, 31  
 generalized eigensystems, 176  
 generalized hermitian definite eigensystems,  
     175  
 generalized symmetric eigensystems, 174  
 Geometric random variates, 277, 278  
 GER, Level-2 BLAS, 134  
 GERC, Level-2 BLAS, 134  
 GERU, Level-2 BLAS, 134  
 Givens rotation, 160  
 Givens Rotation, BLAS, 132  
 Givens Rotation, Modified, BLAS, 132  
 gmres, 621  
 GNU General Public License, 1  
 golden section algorithm for finding minima,  
     450  
 gsl\_acosh (C function), 18  
 gsl\_asinh (C function), 18  
 gsl\_atanh (C function), 18  
 gsl\_blas\_caxpy (C function), 132  
 gsl\_blas\_ccopy (C function), 131  
 gsl\_blas\_cdotc (C function), 130  
 gsl\_blas\_cdotu (C function), 130  
 gsl\_blas\_cgemm (C function), 135  
 gsl\_blas\_cgemv (C function), 133  
 gsl\_blas\_cgerc (C function), 134  
 gsl\_blas\_cgeru (C function), 134  
 gsl\_blas\_chemm (C function), 136  
 gsl\_blas\_chemv (C function), 134  
 gsl\_blas\_cher (C function), 134  
 gsl\_blas\_cher2 (C function), 135  
 gsl\_blas\_cher2k (C function), 137



`gsl_blas_cherk` (*C function*), 137  
`gsl_blas_cscal` (*C function*), 132  
`gsl_blas_csscal` (*C function*), 132  
`gsl_blas_cswap` (*C function*), 131  
`gsl_blas_csymm` (*C function*), 135  
`gsl_blas_csy2k` (*C function*), 137  
`gsl_blas_csyrk` (*C function*), 137  
`gsl_blas_ctrmm` (*C function*), 136  
`gsl_blas_ctrmv` (*C function*), 133  
`gsl_blas_ctrsm` (*C function*), 136  
`gsl_blas_ctrsv` (*C function*), 133  
`gsl_blas_dasum` (*C function*), 131  
`gsl_blas_daxpy` (*C function*), 132  
`gsl_blas_dcopy` (*C function*), 131  
`gsl_blas_ddot` (*C function*), 130  
`gsl_blas_dgemm` (*C function*), 135  
`gsl_blas_dgemv` (*C function*), 133  
`gsl_blas_dger` (*C function*), 134  
`gsl_blas_dnorm2` (*C function*), 131  
`gsl_blas_drot` (*C function*), 132  
`gsl_blas_drotg` (*C function*), 132  
`gsl_blas_drotm` (*C function*), 132  
`gsl_blas_drotmg` (*C function*), 132  
`gsl_blas_dscal` (*C function*), 132  
`gsl_blas_dsdot` (*C function*), 130  
`gsl_blas_dswap` (*C function*), 131  
`gsl_blas_dsymm` (*C function*), 135  
`gsl_blas_dsymv` (*C function*), 133  
`gsl_blas_dsyr` (*C function*), 134  
`gsl_blas_dsyr2` (*C function*), 134  
`gsl_blas_dsyr2k` (*C function*), 137  
`gsl_blas_dsyrc` (*C function*), 137  
`gsl_blas_dtrmm` (*C function*), 136  
`gsl_blas_dtrmv` (*C function*), 133  
`gsl_blas_dtrsm` (*C function*), 136  
`gsl_blas_dtrsv` (*C function*), 133  
`gsl_blas_dzasum` (*C function*), 131  
`gsl_blas_dznrm2` (*C function*), 131  
`gsl_blas_icamax` (*C function*), 131  
`gsl_blas_idamax` (*C function*), 131  
`gsl_blas_isamax` (*C function*), 131  
`gsl_blas_izamax` (*C function*), 131  
`gsl_blas_sasum` (*C function*), 131  
`gsl_blas_saxpy` (*C function*), 132  
`gsl_blas_scasum` (*C function*), 131  
`gsl_blas_scnrm2` (*C function*), 131  
`gsl_blas_scopy` (*C function*), 131  
`gsl_blas_sdot` (*C function*), 130  
`gsl_blas_sdsdot` (*C function*), 130  
`gsl_blas_sgemm` (*C function*), 135  
`gsl_blas_sgemv` (*C function*), 133  
`gsl_blas_sger` (*C function*), 134  
`gsl_blas_snorm2` (*C function*), 131  
`gsl_blas_srot` (*C function*), 132  
`gsl_blas_srotg` (*C function*), 132  
`gsl_blas_srotm` (*C function*), 132  
`gsl_blas_srotmg` (*C function*), 132  
`gsl_blas_sscal` (*C function*), 132  
`gsl_blas_sswap` (*C function*), 131  
`gsl_blas_ssymm` (*C function*), 135  
`gsl_blas_ssymv` (*C function*), 133  
`gsl_blas_ssyr` (*C function*), 134  
`gsl_blas_ssyr2` (*C function*), 134  
`gsl_blas_ssyr2k` (*C function*), 137  
`gsl_blas_ssyrc` (*C function*), 137  
`gsl_blas_strmm` (*C function*), 136  
`gsl_blas_strmv` (*C function*), 133  
`gsl_blas_strsm` (*C function*), 136  
`gsl_blas_strsv` (*C function*), 133  
`gsl_blas_zaxpy` (*C function*), 132  
`gsl_blas_zcopy` (*C function*), 131  
`gsl_blas_zdotc` (*C function*), 130  
`gsl_blas_zdotu` (*C function*), 130  
`gsl_blas_zdscal` (*C function*), 132  
`gsl_blas_zgemm` (*C function*), 135  
`gsl_blas_zgemv` (*C function*), 133  
`gsl_blas_zgerc` (*C function*), 134  
`gsl_blas_zgeru` (*C function*), 134  
`gsl_blas_zhemm` (*C function*), 136  
`gsl_blas_zhemv` (*C function*), 134  
`gsl_blas_zher` (*C function*), 134  
`gsl_blas_zher2` (*C function*), 135  
`gsl_blas_zher2k` (*C function*), 137  
`gsl_blas_zherk` (*C function*), 137  
`gsl_blas_zscal` (*C function*), 132  
`gsl_blas_zswap` (*C function*), 131  
`gsl_blas_zsymm` (*C function*), 135  
`gsl_blas_zsyr2k` (*C function*), 137  
`gsl_blas_zsyrc` (*C function*), 137  
`gsl_blas_ztrmm` (*C function*), 136  
`gsl_blas_ztrmv` (*C function*), 133  
`gsl_blas_ztrsm` (*C function*), 136  
`gsl_blas_ztrsv` (*C function*), 133  
`gsl_block` (*C type*), 84  
`gsl_block_alloc` (*C function*), 84  
`gsl_block_calloc` (*C function*), 84  
`gsl_block_fprintf` (*C function*), 84  
`gsl_block_fread` (*C function*), 84  
`gsl_block_free` (*C function*), 84  
`gsl_block_fscanf` (*C function*), 85  
`gsl_block_fwrite` (*C function*), 84  
`gsl_bspline_alloc` (*C function*), 568  
`gsl_bspline_alloc_ncontrol` (*C function*), 568  
`gsl_bspline_basis` (*C function*), 570  
`gsl_bspline_basis_deriv` (*C function*), 571  
`gsl_bspline_basis_integ` (*C function*), 571  
`gsl_bspline_calc` (*C function*), 570  
`gsl_bspline_calc_deriv` (*C function*), 570

- gsl\_bspline\_calc\_integ (C function), 571  
 gsl\_bspline\_col\_interp (C function), 575  
 gsl\_bspline\_covariance (C function), 573  
 gsl\_bspline\_err (C function), 573  
 gsl\_bspline\_free (C function), 568  
 gsl\_bspline\_gram (C function), 575  
 gsl\_bspline\_gram\_interval (C function), 575  
 gsl\_bspline\_greville\_abscissa (C function), 576  
 gsl\_bspline\_init (C function), 569  
 gsl\_bspline\_init\_augment (C function), 569  
 gsl\_bspline\_init\_interp (C function), 569  
 gsl\_bspline\_init\_periodic (C function), 569  
 gsl\_bspline\_init\_uniform (C function), 569  
 gsl\_bspline\_lsnormal (C function), 572  
 gsl\_bspline\_lsnormalm (C function), 572  
 gsl\_bspline\_lssolve (C function), 572  
 gsl\_bspline\_nbreak (C function), 570  
 gsl\_bspline\_ncontrol (C function), 570  
 gsl\_bspline\_oprod (C function), 574  
 gsl\_bspline\_order (C function), 570  
 gsl\_bspline\_plsqr (C function), 574  
 gsl\_bspline\_plssolve (C function), 574  
 gsl\_bspline\_proj\_rhs (C function), 576  
 gsl\_bspline\_pwlssolve (C function), 574  
 gsl\_bspline\_rcond (C function), 573  
 gsl\_bspline\_residuals (C function), 573  
 gsl\_bspline\_vector\_calc (C function), 570  
 gsl\_bspline\_vector\_calc\_deriv (C function), 571  
 gsl\_bspline\_wlssolve (C function), 572  
 gsl\_bspline\_workspace (C type), 568  
 GSL\_C99\_INLINE, 7  
 GSL\_C99\_INLINE (C macro), 86  
 gsl\_cdf\_beta\_P (C function), 262  
 gsl\_cdf\_beta\_Pinv (C function), 262  
 gsl\_cdf\_beta\_Q (C function), 262  
 gsl\_cdf\_beta\_Qinv (C function), 262  
 gsl\_cdf\_binomial\_P (C function), 273  
 gsl\_cdf\_binomial\_Q (C function), 273  
 gsl\_cdf\_cauchy\_P (C function), 246  
 gsl\_cdf\_cauchy\_Pinv (C function), 246  
 gsl\_cdf\_cauchy\_Q (C function), 246  
 gsl\_cdf\_cauchy\_Qinv (C function), 246  
 gsl\_cdf\_chisq\_P (C function), 256  
 gsl\_cdf\_chisq\_Pinv (C function), 256  
 gsl\_cdf\_chisq\_Q (C function), 256  
 gsl\_cdf\_chisq\_Qinv (C function), 256  
 gsl\_cdf\_exponential\_P (C function), 243  
 gsl\_cdf\_exponential\_Pinv (C function), 243  
 gsl\_cdf\_exponential\_Q (C function), 243  
 gsl\_cdf\_exponential\_Qinv (C function), 243  
 gsl\_cdf\_exppow\_P (C function), 245  
 gsl\_cdf\_exppow\_Q (C function), 245  
 gsl\_cdf\_fdist\_P (C function), 258  
 gsl\_cdf\_fdist\_Pinv (C function), 258  
 gsl\_cdf\_fdist\_Q (C function), 258  
 gsl\_cdf\_fdist\_Qinv (C function), 258  
 gsl\_cdf\_flat\_P (C function), 254  
 gsl\_cdf\_flat\_Pinv (C function), 254  
 gsl\_cdf\_flat\_Q (C function), 254  
 gsl\_cdf\_flat\_Qinv (C function), 254  
 gsl\_cdf\_gamma\_P (C function), 252  
 gsl\_cdf\_gamma\_Pinv (C function), 252  
 gsl\_cdf\_gamma\_Q (C function), 252  
 gsl\_cdf\_gamma\_Qinv (C function), 252  
 gsl\_cdf\_gaussian\_P (C function), 238  
 gsl\_cdf\_gaussian\_Pinv (C function), 238  
 gsl\_cdf\_gaussian\_Q (C function), 238  
 gsl\_cdf\_gaussian\_Qinv (C function), 238  
 gsl\_cdf\_geometric\_P (C function), 277  
 gsl\_cdf\_geometric\_Q (C function), 277  
 gsl\_cdf\_gumbel1\_P (C function), 267  
 gsl\_cdf\_gumbel1\_Pinv (C function), 267  
 gsl\_cdf\_gumbel1\_Q (C function), 267  
 gsl\_cdf\_gumbel1\_Qinv (C function), 267  
 gsl\_cdf\_gumbel2\_P (C function), 268  
 gsl\_cdf\_gumbel2\_Pinv (C function), 268  
 gsl\_cdf\_gumbel2\_Q (C function), 268  
 gsl\_cdf\_gumbel2\_Qinv (C function), 268  
 gsl\_cdf\_hypergeometric\_P (C function), 278  
 gsl\_cdf\_hypergeometric\_Q (C function), 278  
 gsl\_cdf\_laplace\_P (C function), 244  
 gsl\_cdf\_laplace\_Pinv (C function), 244  
 gsl\_cdf\_laplace\_Q (C function), 244  
 gsl\_cdf\_laplace\_Qinv (C function), 244  
 gsl\_cdf\_logistic\_P (C function), 263  
 gsl\_cdf\_logistic\_Pinv (C function), 263  
 gsl\_cdf\_logistic\_Q (C function), 263  
 gsl\_cdf\_logistic\_Qinv (C function), 263  
 gsl\_cdf\_lognormal\_P (C function), 255  
 gsl\_cdf\_lognormal\_Pinv (C function), 255  
 gsl\_cdf\_lognormal\_Q (C function), 255  
 gsl\_cdf\_lognormal\_Qinv (C function), 255  
 gsl\_cdf\_negative\_binomial\_P (C function), 275  
 gsl\_cdf\_negative\_binomial\_Q (C function), 275  
 gsl\_cdf\_pareto\_P (C function), 264  
 gsl\_cdf\_pareto\_Pinv (C function), 264  
 gsl\_cdf\_pareto\_Q (C function), 264  
 gsl\_cdf\_pareto\_Qinv (C function), 264  
 gsl\_cdf\_pascal\_P (C function), 276  
 gsl\_cdf\_pascal\_Q (C function), 276  
 gsl\_cdf\_poisson\_P (C function), 271  
 gsl\_cdf\_poisson\_Q (C function), 271  
 gsl\_cdf\_rayleigh\_P (C function), 247  
 gsl\_cdf\_rayleigh\_Pinv (C function), 247  
 gsl\_cdf\_rayleigh\_Q (C function), 247  
 gsl\_cdf\_rayleigh\_Qinv (C function), 247  
 gsl\_cdf\_tdist\_P (C function), 260  
 gsl\_cdf\_tdist\_Pinv (C function), 260

- gsl\_cdf\_tdist\_Q (*C function*), 260  
 gsl\_cdf\_tdist\_Qinv (*C function*), 260  
 gsl\_cdf\_ugaussian\_P (*C function*), 238  
 gsl\_cdf\_ugaussian\_Pinv (*C function*), 238  
 gsl\_cdf\_ugaussian\_Q (*C function*), 238  
 gsl\_cdf\_ugaussian\_Qinv (*C function*), 238  
 gsl\_cdf\_weibull\_P (*C function*), 266  
 gsl\_cdf\_weibull\_Pinv (*C function*), 266  
 gsl\_cdf\_weibull\_Q (*C function*), 266  
 gsl\_cdf\_weibull\_Qinv (*C function*), 266  
 gsl\_cheb\_alloc (*C function*), 415  
 gsl\_cheb\_calc\_deriv (*C function*), 416  
 gsl\_cheb\_calc\_integ (*C function*), 416  
 gsl\_cheb\_coeffs (*C function*), 416  
 gsl\_cheb\_eval (*C function*), 416  
 gsl\_cheb\_eval\_err (*C function*), 416  
 gsl\_cheb\_eval\_n (*C function*), 416  
 gsl\_cheb\_eval\_n\_err (*C function*), 416  
 gsl\_cheb\_free (*C function*), 415  
 gsl\_cheb\_init (*C function*), 415  
 gsl\_cheb\_order (*C function*), 416  
 gsl\_cheb\_series (*C type*), 415  
 gsl\_cheb\_size (*C function*), 416  
 gsl\_check\_range (*C var*), 87  
 gsl\_combination (*C type*), 113  
 gsl\_combination\_alloc (*C function*), 113  
 gsl\_combination\_calloc (*C function*), 113  
 gsl\_combination\_data (*C function*), 114  
 gsl\_combination\_fprintf (*C function*), 115  
 gsl\_combination\_fread (*C function*), 115  
 gsl\_combination\_free (*C function*), 113  
 gsl\_combination\_fscanf (*C function*), 115  
 gsl\_combination\_fwrite (*C function*), 115  
 gsl\_combination\_get (*C function*), 114  
 gsl\_combination\_init\_first (*C function*), 113  
 gsl\_combination\_init\_last (*C function*), 113  
 gsl\_combination\_k (*C function*), 114  
 gsl\_combination\_memcpy (*C function*), 114  
 gsl\_combination\_n (*C function*), 114  
 gsl\_combination\_next (*C function*), 114  
 gsl\_combination\_prev (*C function*), 114  
 gsl\_combination\_valid (*C function*), 114  
 gsl\_complex, 23  
 gsl\_complex\_abs (*C function*), 25  
 gsl\_complex\_abs2 (*C function*), 25  
 gsl\_complex\_add (*C function*), 25  
 gsl\_complex\_add\_imag (*C function*), 25  
 gsl\_complex\_add\_real (*C function*), 25  
 gsl\_complex\_arccos (*C function*), 27  
 gsl\_complex\_arccos\_real (*C function*), 27  
 gsl\_complex\_arccosh (*C function*), 28  
 gsl\_complex\_arccosh\_real (*C function*), 28  
 gsl\_complex\_arccot (*C function*), 27  
 gsl\_complex\_arccoth (*C function*), 28  
 gsl\_complex\_arccsc (*C function*), 27  
 gsl\_complex\_arccsc\_real (*C function*), 27  
 gsl\_complex\_arccsch (*C function*), 28  
 gsl\_complex\_arcsec (*C function*), 27  
 gsl\_complex\_arcsec\_real (*C function*), 27  
 gsl\_complex\_arcsech (*C function*), 28  
 gsl\_complex\_arcsin (*C function*), 27  
 gsl\_complex\_arcsin\_real (*C function*), 27  
 gsl\_complex\_arcsinh (*C function*), 28  
 gsl\_complex\_arctan (*C function*), 27  
 gsl\_complex\_arctanh (*C function*), 28  
 gsl\_complex\_arctanh\_real (*C function*), 28  
 gsl\_complex\_arg (*C function*), 25  
 gsl\_complex\_conjugate (*C function*), 25  
 gsl\_complex\_cos (*C function*), 26  
 gsl\_complex\_cosh (*C function*), 27  
 gsl\_complex\_cot (*C function*), 26  
 gsl\_complex\_coth (*C function*), 27  
 gsl\_complex\_csc (*C function*), 26  
 gsl\_complex\_csch (*C function*), 27  
 gsl\_complex\_div (*C function*), 25  
 gsl\_complex\_div\_imag (*C function*), 25  
 gsl\_complex\_div\_real (*C function*), 25  
 gsl\_complex\_exp (*C function*), 26  
 gsl\_complex\_inverse (*C function*), 25  
 gsl\_complex\_log (*C function*), 26  
 gsl\_complex\_log10 (*C function*), 26  
 gsl\_complex\_log\_b (*C function*), 26  
 gsl\_complex\_logabs (*C function*), 25  
 gsl\_complex\_mul (*C function*), 25  
 gsl\_complex\_mul\_imag (*C function*), 25  
 gsl\_complex\_mul\_real (*C function*), 25  
 gsl\_complex\_negative (*C function*), 25  
 gsl\_complex\_polar (*C function*), 24  
 gsl\_complex\_poly\_complex\_eval (*C function*), 29  
 gsl\_complex\_pow (*C function*), 26  
 gsl\_complex\_pow\_real (*C function*), 26  
 gsl\_complex\_rect (*C function*), 24  
 gsl\_complex\_sec (*C function*), 26  
 gsl\_complex\_sech (*C function*), 27  
 gsl\_complex\_sin (*C function*), 26  
 gsl\_complex\_sinh (*C function*), 27  
 gsl\_complex\_sqrt (*C function*), 26  
 gsl\_complex\_sqrt\_real (*C function*), 26  
 gsl\_complex\_sub (*C function*), 25  
 gsl\_complex\_sub\_imag (*C function*), 25  
 gsl\_complex\_sub\_real (*C function*), 25  
 gsl\_complex\_tan (*C function*), 26  
 gsl\_complex\_tanh (*C function*), 27  
 GSL\_CONST\_MKSA\_ACRE (*C macro*), 630  
 GSL\_CONST\_MKSA\_ANGSTROM (*C macro*), 628  
 GSL\_CONST\_MKSA\_ASTRONOMICAL\_UNIT (*C macro*), 628  
 GSL\_CONST\_MKSA\_BAR (*C macro*), 632  
 GSL\_CONST\_MKSA\_BARN (*C macro*), 628



- GSL\_CONST\_MKSA\_BOHR\_MAGNETON (*C macro*), 629  
 GSL\_CONST\_MKSA\_BOHR\_RADIUS (*C macro*), 628  
 GSL\_CONST\_MKSA\_BOLTZMANN (*C macro*), 627  
 GSL\_CONST\_MKSA\_BTU (*C macro*), 631  
 GSL\_CONST\_MKSA\_CALORIE (*C macro*), 631  
 GSL\_CONST\_MKSA\_CANADIAN\_GALLON (*C macro*), 630  
 GSL\_CONST\_MKSA\_CARAT (*C macro*), 631  
 GSL\_CONST\_MKSA\_CURIE (*C macro*), 633  
 GSL\_CONST\_MKSA\_DAY (*C macro*), 629  
 GSL\_CONST\_MKSA\_DEBYE (*C macro*), 629  
 GSL\_CONST\_MKSA\_DYNE (*C macro*), 633  
 GSL\_CONST\_MKSA\_ELECTRON\_CHARGE (*C macro*), 628  
 GSL\_CONST\_MKSA\_ELECTRON\_MAGNETIC\_MOMENT (*C macro*), 629  
 GSL\_CONST\_MKSA\_ELECTRON\_VOLT (*C macro*), 628  
 GSL\_CONST\_MKSA\_ERG (*C macro*), 633  
 GSL\_CONST\_MKSA\_FARADAY (*C macro*), 627  
 GSL\_CONST\_MKSA\_FATHOM (*C macro*), 630  
 GSL\_CONST\_MKSA\_FOOT (*C macro*), 629  
 GSL\_CONST\_MKSA\_FOOTCANDLE (*C macro*), 632  
 GSL\_CONST\_MKSA\_FOOTLAMBERT (*C macro*), 632  
 GSL\_CONST\_MKSA\_GAUSS (*C macro*), 627  
 GSL\_CONST\_MKSA\_GRAM\_FORCE (*C macro*), 631  
 GSL\_CONST\_MKSA\_GRAV\_ACCEL (*C macro*), 628  
 GSL\_CONST\_MKSA\_GRAVITATIONAL\_CONSTANT (*C macro*), 628  
 GSL\_CONST\_MKSA\_HECTARE (*C macro*), 630  
 GSL\_CONST\_MKSA\_HORSEPOWER (*C macro*), 631  
 GSL\_CONST\_MKSA\_HOUR (*C macro*), 629  
 GSL\_CONST\_MKSA\_INCH (*C macro*), 629  
 GSL\_CONST\_MKSA\_INCH\_OF\_MERCURY (*C macro*), 632  
 GSL\_CONST\_MKSA\_INCH\_OF\_WATER (*C macro*), 632  
 GSL\_CONST\_MKSA\_JOULE (*C macro*), 633  
 GSL\_CONST\_MKSA\_KILOMETERS\_PER\_HOUR (*C macro*), 630  
 GSL\_CONST\_MKSA\_KILOPOUND\_FORCE (*C macro*), 631  
 GSL\_CONST\_MKSA\_KNOT (*C macro*), 630  
 GSL\_CONST\_MKSA\_LAMBERT (*C macro*), 632  
 GSL\_CONST\_MKSA\_LIGHT\_YEAR (*C macro*), 628  
 GSL\_CONST\_MKSA\_LITER (*C macro*), 630  
 GSL\_CONST\_MKSA\_LUMEN (*C macro*), 632  
 GSL\_CONST\_MKSA\_LUX (*C macro*), 632  
 GSL\_CONST\_MKSA\_MASS\_ELECTRON (*C macro*), 628  
 GSL\_CONST\_MKSA\_MASS\_MUON (*C macro*), 628  
 GSL\_CONST\_MKSA\_MASS\_NEUTRON (*C macro*), 628  
 GSL\_CONST\_MKSA\_MASS\_PROTON (*C macro*), 628  
 GSL\_CONST\_MKSA\_METER\_OF\_MERCURY (*C macro*), 632  
 GSL\_CONST\_MKSA\_METRIC\_TON (*C macro*), 631  
 GSL\_CONST\_MKSA\_MICRON (*C macro*), 630  
 GSL\_CONST\_MKSA\_MIL (*C macro*), 629  
 GSL\_CONST\_MKSA\_MILE (*C macro*), 629  
 GSL\_CONST\_MKSA\_MILES\_PER\_HOUR (*C macro*), 630  
 GSL\_CONST\_MKSA\_MINUTE (*C macro*), 629  
 GSL\_CONST\_MKSA\_MOLAR\_GAS (*C macro*), 627  
 GSL\_CONST\_MKSA\_NAUTICAL\_MILE (*C macro*), 630  
 GSL\_CONST\_MKSA\_NEWTON (*C macro*), 633  
 GSL\_CONST\_MKSA\_NUCLEAR\_MAGNETON (*C macro*), 629  
 GSL\_CONST\_MKSA\_OUNCE\_MASS (*C macro*), 631  
 GSL\_CONST\_MKSA\_PARSEC (*C macro*), 628  
 GSL\_CONST\_MKSA\_PHOT (*C macro*), 632  
 GSL\_CONST\_MKSA\_PINT (*C macro*), 630  
 GSL\_CONST\_MKSA\_PLANCKS\_CONSTANT\_H (*C macro*), 627  
 GSL\_CONST\_MKSA\_PLANCKS\_CONSTANT\_HBAR (*C macro*), 627  
 GSL\_CONST\_MKSA\_POINT (*C macro*), 630  
 GSL\_CONST\_MKSA\_POISE (*C macro*), 632  
 GSL\_CONST\_MKSA\_POUND\_FORCE (*C macro*), 631  
 GSL\_CONST\_MKSA\_POUND\_MASS (*C macro*), 631  
 GSL\_CONST\_MKSA\_POUNDAL (*C macro*), 631  
 GSL\_CONST\_MKSA\_PROTON\_MAGNETIC\_MOMENT (*C macro*), 629  
 GSL\_CONST\_MKSA\_PSI (*C macro*), 632  
 GSL\_CONST\_MKSA\_QUART (*C macro*), 630  
 GSL\_CONST\_MKSA\_RAD (*C macro*), 633  
 GSL\_CONST\_MKSA\_ROENTGEN (*C macro*), 633  
 GSL\_CONST\_MKSA\_RYDBERG (*C macro*), 628  
 GSL\_CONST\_MKSA\_SOLAR\_MASS (*C macro*), 628  
 GSL\_CONST\_MKSA\_SPEED\_OF\_LIGHT (*C macro*), 627  
 GSL\_CONST\_MKSA\_STANDARD\_GAS\_VOLUME (*C macro*), 627  
 GSL\_CONST\_MKSA\_STD\_ATMOSPHERE (*C macro*), 632  
 GSL\_CONST\_MKSA\_STEFAN\_BOLTZMANN\_CONSTANT (*C macro*), 627  
 GSL\_CONST\_MKSA\_STILB (*C macro*), 632  
 GSL\_CONST\_MKSA\_STOKES (*C macro*), 632  
 GSL\_CONST\_MKSA\_TEXPOINT (*C macro*), 630  
 GSL\_CONST\_MKSA\_THERM (*C macro*), 631  
 GSL\_CONST\_MKSA\_THOMSON\_CROSS\_SECTION (*C macro*), 629  
 GSL\_CONST\_MKSA\_TON (*C macro*), 631  
 GSL\_CONST\_MKSA\_TORR (*C macro*), 632  
 GSL\_CONST\_MKSA\_TROY\_OUNCE (*C macro*), 631  
 GSL\_CONST\_MKSA\_UK\_GALLON (*C macro*), 630  
 GSL\_CONST\_MKSA\_UK\_TON (*C macro*), 631  
 GSL\_CONST\_MKSA\_UNIFIED\_ATOMIC\_MASS (*C macro*), 628  
 GSL\_CONST\_MKSA\_US\_GALLON (*C macro*), 630  
 GSL\_CONST\_MKSA\_VACUUM\_PERMEABILITY (*C macro*), 627  
 GSL\_CONST\_MKSA\_VACUUM\_PERMITTIVITY (*C macro*), 627  
 GSL\_CONST\_MKSA\_WEEK (*C macro*), 629  
 GSL\_CONST\_MKSA\_YARD (*C macro*), 629  
 GSL\_CONST\_NUM\_ATTO (*C macro*), 634  
 GSL\_CONST\_NUM\_AVOGADRO (*C macro*), 627  
 GSL\_CONST\_NUM\_EXA (*C macro*), 633  
 GSL\_CONST\_NUM\_FEMTO (*C macro*), 634

- GSL\_CONST\_NUM\_FINE\_STRUCTURE (*C macro*), 628  
 GSL\_CONST\_NUM\_GIGA (*C macro*), 633  
 GSL\_CONST\_NUM\_KILO (*C macro*), 633  
 GSL\_CONST\_NUM\_MEGA (*C macro*), 633  
 GSL\_CONST\_NUM\_MICRO (*C macro*), 633  
 GSL\_CONST\_NUM\_MILLI (*C macro*), 633  
 GSL\_CONST\_NUM\_NANO (*C macro*), 634  
 GSL\_CONST\_NUM\_PETA (*C macro*), 633  
 GSL\_CONST\_NUM\_PICO (*C macro*), 634  
 GSL\_CONST\_NUM\_TERA (*C macro*), 633  
 GSL\_CONST\_NUM\_YOCTO (*C macro*), 634  
 GSL\_CONST\_NUM\_YOTTA (*C macro*), 633  
 GSL\_CONST\_NUM\_ZEPTO (*C macro*), 634  
 GSL\_CONST\_NUM\_ZETTA (*C macro*), 633  
 gsl\_deriv\_backward (*C function*), 411  
 gsl\_deriv\_central (*C function*), 411  
 gsl\_deriv\_forward (*C function*), 411  
 gsl\_dht (*C type*), 432  
 gsl\_dht\_alloc (*C function*), 432  
 gsl\_dht\_apply (*C function*), 432  
 gsl\_dht\_free (*C function*), 432  
 gsl\_dht\_init (*C function*), 432  
 gsl\_dht\_k\_sample (*C function*), 432  
 gsl\_dht\_new (*C function*), 432  
 gsl\_dht\_x\_sample (*C function*), 432  
 GSL\_EDOM (*C var*), 14  
 gsl\_eigen\_gen (*C function*), 177  
 gsl\_eigen\_gen\_alloc (*C function*), 176  
 gsl\_eigen\_gen\_free (*C function*), 176  
 gsl\_eigen\_gen\_params (*C function*), 176  
 gsl\_eigen\_gen\_QZ (*C function*), 177  
 gsl\_eigen\_gen\_workspace (*C type*), 176  
 gsl\_eigen\_genherm (*C function*), 175  
 gsl\_eigen\_genherm\_alloc (*C function*), 175  
 gsl\_eigen\_genherm\_free (*C function*), 175  
 gsl\_eigen\_genherm\_workspace (*C type*), 175  
 gsl\_eigen\_genhermv (*C function*), 175  
 gsl\_eigen\_genhermv\_alloc (*C function*), 175  
 gsl\_eigen\_genhermv\_free (*C function*), 175  
 gsl\_eigen\_genhermv\_sort (*C function*), 178  
 gsl\_eigen\_genhermv\_workspace (*C type*), 175  
 gsl\_eigen\_gensymm (*C function*), 174  
 gsl\_eigen\_gensymm\_alloc (*C function*), 174  
 gsl\_eigen\_gensymm\_free (*C function*), 174  
 gsl\_eigen\_gensymm\_workspace (*C type*), 174  
 gsl\_eigen\_gensymmv (*C function*), 174  
 gsl\_eigen\_gensymmv\_alloc (*C function*), 174  
 gsl\_eigen\_gensymmv\_free (*C function*), 174  
 gsl\_eigen\_gensymmv\_sort (*C function*), 178  
 gsl\_eigen\_gensymmv\_workspace (*C type*), 174  
 gsl\_eigen\_genv (*C function*), 177  
 gsl\_eigen\_genv\_alloc (*C function*), 177  
 gsl\_eigen\_genv\_free (*C function*), 177  
 gsl\_eigen\_genv\_QZ (*C function*), 177  
 gsl\_eigen\_genv\_sort (*C function*), 178  
 gsl\_eigen\_genv\_workspace (*C type*), 177  
 gsl\_eigen\_herm (*C function*), 172  
 gsl\_eigen\_herm\_alloc (*C function*), 172  
 gsl\_eigen\_herm\_free (*C function*), 172  
 gsl\_eigen\_herm\_workspace (*C type*), 172  
 gsl\_eigen\_hermv (*C function*), 172  
 gsl\_eigen\_hermv\_alloc (*C function*), 172  
 gsl\_eigen\_hermv\_free (*C function*), 172  
 gsl\_eigen\_hermv\_sort (*C function*), 178  
 gsl\_eigen\_hermv\_workspace (*C type*), 172  
 gsl\_eigen\_nonsymm (*C function*), 173  
 gsl\_eigen\_nonsymm\_alloc (*C function*), 173  
 gsl\_eigen\_nonsymm\_free (*C function*), 173  
 gsl\_eigen\_nonsymm\_params (*C function*), 173  
 gsl\_eigen\_nonsymm\_workspace (*C type*), 172  
 gsl\_eigen\_nonsymm\_Z (*C function*), 173  
 gsl\_eigen\_nonsymmv (*C function*), 174  
 gsl\_eigen\_nonsymmv\_alloc (*C function*), 173  
 gsl\_eigen\_nonsymmv\_free (*C function*), 173  
 gsl\_eigen\_nonsymmv\_params (*C function*), 173  
 gsl\_eigen\_nonsymmv\_sort (*C function*), 178  
 gsl\_eigen\_nonsymmv\_workspace (*C type*), 173  
 gsl\_eigen\_nonsymmv\_Z (*C function*), 174  
 gsl\_eigen\_symm (*C function*), 171  
 gsl\_eigen\_symm\_alloc (*C function*), 171  
 gsl\_eigen\_symm\_free (*C function*), 171  
 gsl\_eigen\_symm\_workspace (*C type*), 171  
 gsl\_eigen\_symmv (*C function*), 171  
 gsl\_eigen\_symmv\_alloc (*C function*), 171  
 gsl\_eigen\_symmv\_free (*C function*), 171  
 gsl\_eigen\_symmv\_sort (*C function*), 177  
 gsl\_eigen\_symmv\_sort.gsl\_eigen\_sort\_t (*C type*), 177  
 gsl\_eigen\_symmv\_workspace (*C type*), 171  
 GSL\_EINVAL (*C var*), 14  
 GSL\_ENOMEM (*C var*), 14  
 GSL\_ERANGE (*C var*), 14  
 GSL\_ERROR (*C macro*), 15  
 gsl\_error\_handler\_t (*C type*), 14  
 GSL\_ERROR\_VAL (*C macro*), 16  
 gsl\_expm1 (*C function*), 18  
 gsl\_fcmp (*C function*), 21  
 gsl\_fft\_complex\_backward (*C function*), 189  
 gsl\_fft\_complex\_forward (*C function*), 189  
 gsl\_fft\_complex\_inverse (*C function*), 189  
 gsl\_fft\_complex\_radix2\_backward (*C function*), 185  
 gsl\_fft\_complex\_radix2\_dif\_backward (*C function*), 185  
 gsl\_fft\_complex\_radix2\_dif\_forward (*C function*), 185  
 gsl\_fft\_complex\_radix2\_dif\_inverse (*C function*), 185

- gsl\_fft\_complex\_radix2\_dif\_transform (C function), 185  
 gsl\_fft\_complex\_radix2\_forward (C function), 185  
 gsl\_fft\_complex\_radix2\_inverse (C function), 185  
 gsl\_fft\_complex\_radix2\_transform (C function), 185  
 gsl\_fft\_complex\_transform (C function), 189  
 gsl\_fft\_complex\_wavetable (C type), 188  
 gsl\_fft\_complex\_wavetable\_alloc (C function), 188  
 gsl\_fft\_complex\_wavetable\_free (C function), 188  
 gsl\_fft\_complex\_workspace (C type), 189  
 gsl\_fft\_complex\_workspace\_alloc (C function), 189  
 gsl\_fft\_complex\_workspace\_free (C function), 189  
 gsl\_fft\_halfcomplex\_radix2\_backward (C function), 192  
 gsl\_fft\_halfcomplex\_radix2\_inverse (C function), 192  
 gsl\_fft\_halfcomplex\_radix2\_unpack (C function), 192  
 gsl\_fft\_halfcomplex\_transform (C function), 194  
 gsl\_fft\_halfcomplex\_unpack (C function), 195  
 gsl\_fft\_halfcomplex\_wavetable (C type), 194  
 gsl\_fft\_halfcomplex\_wavetable\_alloc (C function), 194  
 gsl\_fft\_halfcomplex\_wavetable\_free (C function), 194  
 gsl\_fft\_real\_radix2\_transform (C function), 192  
 gsl\_fft\_real\_transform (C function), 194  
 gsl\_fft\_real\_unpack (C function), 195  
 gsl\_fft\_real\_wavetable (C type), 194  
 gsl\_fft\_real\_wavetable\_alloc (C function), 194  
 gsl\_fft\_real\_wavetable\_free (C function), 194  
 gsl\_fft\_real\_workspace (C type), 194  
 gsl\_fft\_real\_workspace\_alloc (C function), 194  
 gsl\_fft\_real\_workspace\_free (C function), 194  
 gsl\_filter\_end\_t (C type), 319  
 gsl\_filter\_end\_t.GSL\_FILTER\_END\_PADVALUE (C macro), 319  
 gsl\_filter\_end\_t.GSL\_FILTER\_END\_PADZERO (C macro), 319  
 gsl\_filter\_end\_t.GSL\_FILTER\_END\_TRUNCATE (C macro), 319  
 gsl\_filter\_gaussian (C function), 320  
 gsl\_filter\_gaussian\_alloc (C function), 320  
 gsl\_filter\_gaussian\_free (C function), 320  
 gsl\_filter\_gaussian\_kernel (C function), 320  
 gsl\_filter\_impulse (C function), 323  
 gsl\_filter\_impulse\_alloc (C function), 323  
 gsl\_filter\_impulse\_free (C function), 323  
 gsl\_filter\_median (C function), 321  
 gsl\_filter\_median\_alloc (C function), 321  
 gsl\_filter\_median\_free (C function), 321  
 gsl\_filter\_rmedian (C function), 321  
 gsl\_filter\_rmedian\_alloc (C function), 321  
 gsl\_filter\_rmedian\_free (C function), 321  
 gsl\_filter\_scale\_t (C type), 322  
 gsl\_filter\_scale\_t.GSL\_FILTER\_SCALE\_IQR (C macro), 322  
 gsl\_filter\_scale\_t.GSL\_FILTER\_SCALE\_MAD (C macro), 322  
 gsl\_filter\_scale\_t.GSL\_FILTER\_SCALE\_QN (C macro), 322  
 gsl\_filter\_scale\_t.GSL\_FILTER\_SCALE\_SN (C macro), 322  
 gsl\_finite (C function), 18  
 gsl\_fit\_linear (C function), 482  
 gsl\_fit\_linear\_est (C function), 482  
 gsl\_fit\_mul (C function), 483  
 gsl\_fit\_mul\_est (C function), 483  
 gsl\_fit\_wlinear (C function), 482  
 gsl\_fit\_wmul (C function), 483  
 gsl\_frexp (C function), 19  
 gsl\_function (C type), 435  
 gsl\_function\_fdf (C type), 436  
 gsl\_heapsort (C function), 123  
 gsl\_heapsort.gsl\_comparison\_fn\_t (C type), 123  
 gsl\_heapsort\_index (C function), 124  
 gsl\_histogram (C type), 335  
 gsl\_histogram2d (C type), 342  
 gsl\_histogram2d\_accumulate (C function), 344  
 gsl\_histogram2d\_add (C function), 346  
 gsl\_histogram2d\_alloc (C function), 343  
 gsl\_histogram2d\_clone (C function), 343  
 gsl\_histogram2d\_cov (C function), 345  
 gsl\_histogram2d\_div (C function), 346  
 gsl\_histogram2d\_equal\_bins\_p (C function), 346  
 gsl\_histogram2d\_find (C function), 345  
 gsl\_histogram2d\_fprintf (C function), 346  
 gsl\_histogram2d\_fread (C function), 346  
 gsl\_histogram2d\_free (C function), 343  
 gsl\_histogram2d\_fscanf (C function), 347  
 gsl\_histogram2d\_fwrite (C function), 346  
 gsl\_histogram2d\_get (C function), 344  
 gsl\_histogram2d\_get\_xrange (C function), 344  
 gsl\_histogram2d\_get\_yrange (C function), 344  
 gsl\_histogram2d\_increment (C function), 344  
 gsl\_histogram2d\_max\_bin (C function), 345  
 gsl\_histogram2d\_max\_val (C function), 345  
 gsl\_histogram2d\_memcpy (C function), 343  
 gsl\_histogram2d\_min\_bin (C function), 345  
 gsl\_histogram2d\_min\_val (C function), 345  
 gsl\_histogram2d\_mul (C function), 346  
 gsl\_histogram2d\_nx (C function), 344  
 gsl\_histogram2d\_ny (C function), 344  
 gsl\_histogram2d\_pdf (C type), 347  
 gsl\_histogram2d\_pdf\_alloc (C function), 348

gsl\_histogram2d\_pdf\_free (C function), 348  
 gsl\_histogram2d\_pdf\_init (C function), 348  
 gsl\_histogram2d\_pdf\_sample (C function), 348  
 gsl\_histogram2d\_reset (C function), 344  
 gsl\_histogram2d\_scale (C function), 346  
 gsl\_histogram2d\_set\_ranges (C function), 343  
 gsl\_histogram2d\_set\_ranges\_uniform (C function), 343  
 gsl\_histogram2d\_shift (C function), 346  
 gsl\_histogram2d\_sub (C function), 346  
 gsl\_histogram2d\_sum (C function), 345  
 gsl\_histogram2d\_xmax (C function), 344  
 gsl\_histogram2d\_xmean (C function), 345  
 gsl\_histogram2d\_xmin (C function), 344  
 gsl\_histogram2d\_xsigma (C function), 345  
 gsl\_histogram2d\_ymax (C function), 344  
 gsl\_histogram2d\_ymean (C function), 345  
 gsl\_histogram2d\_ymin (C function), 344  
 gsl\_histogram2d\_ysigma (C function), 345  
 gsl\_histogram\_accumulate (C function), 337  
 gsl\_histogram\_add (C function), 338  
 gsl\_histogram\_alloc (C function), 336  
 gsl\_histogram\_bins (C function), 337  
 gsl\_histogram\_clone (C function), 337  
 gsl\_histogram\_div (C function), 339  
 gsl\_histogram\_equal\_bins\_p (C function), 338  
 gsl\_histogram\_find (C function), 338  
 gsl\_histogram\_fprintf (C function), 339  
 gsl\_histogram\_fread (C function), 339  
 gsl\_histogram\_free (C function), 336  
 gsl\_histogram\_fscanf (C function), 339  
 gsl\_histogram\_fwrite (C function), 339  
 gsl\_histogram\_get (C function), 337  
 gsl\_histogram\_get\_range (C function), 337  
 gsl\_histogram\_increment (C function), 337  
 gsl\_histogram\_max (C function), 337  
 gsl\_histogram\_max\_bin (C function), 338  
 gsl\_histogram\_max\_val (C function), 338  
 gsl\_histogram\_mean (C function), 338  
 gsl\_histogram\_memcpy (C function), 337  
 gsl\_histogram\_min (C function), 337  
 gsl\_histogram\_min\_bin (C function), 338  
 gsl\_histogram\_min\_val (C function), 338  
 gsl\_histogram\_mul (C function), 338  
 gsl\_histogram\_pdf (C type), 340  
 gsl\_histogram\_pdf\_alloc (C function), 340  
 gsl\_histogram\_pdf\_free (C function), 340  
 gsl\_histogram\_pdf\_init (C function), 340  
 gsl\_histogram\_pdf\_sample (C function), 340  
 gsl\_histogram\_reset (C function), 337  
 gsl\_histogram\_scale (C function), 339  
 gsl\_histogram\_set\_ranges (C function), 336  
 gsl\_histogram\_set\_ranges\_uniform (C function), 336

gsl\_histogram\_shift (C function), 339  
 gsl\_histogram\_sigma (C function), 338  
 gsl\_histogram\_sub (C function), 338  
 gsl\_histogram\_sum (C function), 338  
 gsl\_hypot (C function), 18  
 gsl\_hypot3 (C function), 18  
 gsl\_ieee\_env\_setup (C function), 639  
 gsl\_ieee\_fprintf\_double (C function), 638  
 gsl\_ieee\_fprintf\_float (C function), 638  
 GSL\_IEEE\_MODE (C macro), 639  
 gsl\_ieee\_printf\_double (C function), 638  
 gsl\_ieee\_printf\_float (C function), 638  
 GSL\_IMAG (C macro), 24  
 gsl\_integration\_cquad (C function), 206  
 gsl\_integration\_cquad\_workspace, 206  
 gsl\_integration\_cquad\_workspace\_alloc (C function), 206  
 gsl\_integration\_cquad\_workspace\_free (C function), 206  
 gsl\_integration\_fixed (C function), 209  
 gsl\_integration\_fixed\_alloc (C function), 208  
 gsl\_integration\_fixed\_alloc.gsl\_integration\_fixed\_type (C type), 209  
 gsl\_integration\_fixed\_alloc.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type (C var), 209  
 gsl\_integration\_fixed\_alloc.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type (C var), 209  
 gsl\_integration\_fixed\_alloc.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type (C var), 209  
 gsl\_integration\_fixed\_alloc.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type (C var), 209  
 gsl\_integration\_fixed\_alloc.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type.gsl\_integration\_fixed\_type (C var), 209  
 gsl\_integration\_fixed\_free (C function), 209  
 gsl\_integration\_fixed\_n (C function), 209  
 gsl\_integration\_fixed\_nodes (C function), 209  
 gsl\_integration\_fixed\_weights (C function), 209  
 gsl\_integration\_fixed\_workspace (C type), 208  
 gsl\_integration\_glfixed (C function), 207  
 gsl\_integration\_glfixed\_point (C function), 207  
 gsl\_integration\_glfixed\_table, 207, 208  
 gsl\_integration\_glfixed\_table\_alloc (C function), 207  
 gsl\_integration\_glfixed\_table\_free (C function), 208  
 gsl\_integration\_qag (C function), 201



- gsl\_integration\_qagi (C function), 202  
 gsl\_integration\_qagil (C function), 203  
 gsl\_integration\_qagiui (C function), 202  
 gsl\_integration\_qagp (C function), 202  
 gsl\_integration\_qags (C function), 202  
 gsl\_integration\_qawc (C function), 203  
 gsl\_integration\_qawf (C function), 205  
 gsl\_integration\_qawo (C function), 205  
 gsl\_integration\_qawo\_table, 204  
 gsl\_integration\_qawo\_table\_alloc (C function), 204  
 gsl\_integration\_qawo\_table\_alloc.GSL\_INTEG\_COST (C macro), 204  
 gsl\_integration\_qawo\_table\_alloc.GSL\_INTEG\_SING (C macro), 204  
 gsl\_integration\_qawo\_table\_free (C function), 205  
 gsl\_integration\_qawo\_table\_set (C function), 205  
 gsl\_integration\_qawo\_table\_set\_length (C function), 205  
 gsl\_integration\_qaws (C function), 204  
 gsl\_integration\_qaws\_table (C type), 203  
 gsl\_integration\_qaws\_table\_alloc (C function), 203  
 gsl\_integration\_qaws\_table\_free (C function), 204  
 gsl\_integration\_qaws\_table\_set (C function), 204  
 gsl\_integration\_qng (C function), 201  
 gsl\_integration\_romberg (C function), 207  
 gsl\_integration\_romberg\_alloc (C function), 207  
 gsl\_integration\_romberg\_free (C function), 207  
 gsl\_integration\_workspace, 201  
 gsl\_integration\_workspace (C type), 201  
 gsl\_integration\_workspace\_alloc (C function), 201  
 gsl\_integration\_workspace\_free (C function), 201  
 gsl\_interp (C type), 395  
 gsl\_interp2d (C type), 403  
 gsl\_interp2d\_alloc (C function), 403  
 gsl\_interp2d\_eval (C function), 405  
 gsl\_interp2d\_eval\_deriv\_x (C function), 405  
 gsl\_interp2d\_eval\_deriv\_x\_e (C function), 405  
 gsl\_interp2d\_eval\_deriv\_xx (C function), 405  
 gsl\_interp2d\_eval\_deriv\_xx\_e (C function), 405  
 gsl\_interp2d\_eval\_deriv\_xy (C function), 406  
 gsl\_interp2d\_eval\_deriv\_xy\_e (C function), 406  
 gsl\_interp2d\_eval\_deriv\_y (C function), 405  
 gsl\_interp2d\_eval\_deriv\_y\_e (C function), 405  
 gsl\_interp2d\_eval\_deriv\_yy (C function), 405  
 gsl\_interp2d\_eval\_deriv\_yy\_e (C function), 405  
 gsl\_interp2d\_eval\_e (C function), 405  
 gsl\_interp2d\_eval\_extrap (C function), 405  
 gsl\_interp2d\_eval\_extrap\_e (C function), 405  
 gsl\_interp2d\_free (C function), 404  
 gsl\_interp2d\_get (C function), 404  
 gsl\_interp2d\_idx (C function), 404  
 gsl\_interp2d\_init (C function), 403  
 gsl\_interp2d\_min\_size (C function), 404  
 gsl\_interp2d\_name (C function), 404  
 gsl\_interp2d\_set (C function), 404  
 gsl\_interp2d\_type (C type), 404  
 gsl\_interp2d\_type.gsl\_interp2d\_bicubic (C var), 404  
 gsl\_interp2d\_type.gsl\_interp2d\_bilinear (C var), 404  
 gsl\_interp2d\_type.gsl\_interp2d\_min\_size (C function), 404  
 gsl\_interp\_accel (C type), 397  
 gsl\_interp\_accel\_alloc (C function), 397  
 gsl\_interp\_accel\_find (C function), 397  
 gsl\_interp\_accel\_free (C function), 397  
 gsl\_interp\_accel\_reset (C function), 397  
 gsl\_interp\_alloc (C function), 395  
 gsl\_interp\_bsearch (C function), 397  
 gsl\_interp\_eval (C function), 397  
 gsl\_interp\_eval\_deriv (C function), 397  
 gsl\_interp\_eval\_deriv2 (C function), 397  
 gsl\_interp\_eval\_deriv2\_e (C function), 397  
 gsl\_interp\_eval\_deriv\_e (C function), 397  
 gsl\_interp\_eval\_e (C function), 397  
 gsl\_interp\_eval\_integ (C function), 398  
 gsl\_interp\_eval\_integ\_e (C function), 398  
 gsl\_interp\_free (C function), 395  
 gsl\_interp\_init (C function), 395  
 gsl\_interp\_min\_size (C function), 396  
 gsl\_interp\_name (C function), 396  
 gsl\_interp\_type (C type), 396  
 gsl\_interp\_type.gsl\_interp\_akima (C var), 396  
 gsl\_interp\_type.gsl\_interp\_akima\_periodic (C var), 396  
 gsl\_interp\_type.gsl\_interp\_cspline (C var), 396  
 gsl\_interp\_type.gsl\_interp\_cspline\_periodic (C var), 396  
 gsl\_interp\_type.gsl\_interp\_linear (C var), 396  
 gsl\_interp\_type.gsl\_interp\_polynomial (C var), 396  
 gsl\_interp\_type.gsl\_interp\_steffen (C var), 396  
 gsl\_interp\_type\_min\_size (C function), 396  
 GSL\_IS\_EVEN (C macro), 20  
 GSL\_IS\_ODD (C macro), 20  
 gsl\_isinf (C function), 18  
 gsl\_isnan (C function), 18  
 gsl\_ldexp (C function), 19  
 gsl\_linalg\_balance\_matrix (C function), 168  
 gsl\_linalg\_bidiag\_decomp (C function), 160  
 gsl\_linalg\_bidiag\_unpack (C function), 160  
 gsl\_linalg\_bidiag\_unpack2 (C function), 160  
 gsl\_linalg\_bidiag\_unpack\_B (C function), 160

- gsl\_linalg\_cholesky\_band\_decomp (*C function*), 166  
 gsl\_linalg\_cholesky\_band\_invert (*C function*), 166  
 gsl\_linalg\_cholesky\_band\_rcond (*C function*), 167  
 gsl\_linalg\_cholesky\_band\_scale (*C function*), 167  
 gsl\_linalg\_cholesky\_band\_scale\_apply (*C function*), 167  
 gsl\_linalg\_cholesky\_band\_solve (*C function*), 166  
 gsl\_linalg\_cholesky\_band\_solvem (*C function*), 166  
 gsl\_linalg\_cholesky\_band\_svx (*C function*), 166  
 gsl\_linalg\_cholesky\_band\_svxm (*C function*), 166  
 gsl\_linalg\_cholesky\_band\_unpack (*C function*), 166  
 gsl\_linalg\_cholesky\_decomp (*C function*), 154  
 gsl\_linalg\_cholesky\_decomp1 (*C function*), 154  
 gsl\_linalg\_cholesky\_decomp2 (*C function*), 155  
 gsl\_linalg\_cholesky\_invert (*C function*), 155  
 gsl\_linalg\_cholesky\_rcond (*C function*), 155  
 gsl\_linalg\_cholesky\_scale (*C function*), 155  
 gsl\_linalg\_cholesky\_scale\_apply (*C function*), 155  
 gsl\_linalg\_cholesky\_solve (*C function*), 155  
 gsl\_linalg\_cholesky\_solve2 (*C function*), 155  
 gsl\_linalg\_cholesky\_svx (*C function*), 155  
 gsl\_linalg\_cholesky\_svx2 (*C function*), 155  
 gsl\_linalg\_COD\_decomp (*C function*), 152  
 gsl\_linalg\_COD\_decomp\_e (*C function*), 152  
 gsl\_linalg\_COD\_lassolve (*C function*), 152  
 gsl\_linalg\_COD\_lassolve2 (*C function*), 153  
 gsl\_linalg\_COD\_matZ (*C function*), 153  
 gsl\_linalg\_COD\_unpack (*C function*), 153  
 gsl\_linalg\_complex\_cholesky\_decomp (*C function*), 154  
 gsl\_linalg\_complex\_cholesky\_invert (*C function*), 155  
 gsl\_linalg\_complex\_cholesky\_solve (*C function*), 155  
 gsl\_linalg\_complex\_cholesky\_svx (*C function*), 155  
 gsl\_linalg\_complex\_householder\_hm (*C function*), 161  
 gsl\_linalg\_complex\_householder\_hv (*C function*), 162  
 gsl\_linalg\_complex\_householder\_mh (*C function*), 162  
 gsl\_linalg\_complex\_householder\_transform (*C function*), 161  
 gsl\_linalg\_complex\_LU\_decomp (*C function*), 141  
 gsl\_linalg\_complex\_LU\_det (*C function*), 142  
 gsl\_linalg\_complex\_LU\_invert (*C function*), 142  
 gsl\_linalg\_complex\_LU\_invx (*C function*), 142  
 gsl\_linalg\_complex\_LU\_lndet (*C function*), 142  
 gsl\_linalg\_complex\_LU\_refine (*C function*), 142  
 gsl\_linalg\_complex\_LU\_sgndet (*C function*), 142  
 gsl\_linalg\_complex\_LU\_solve (*C function*), 141  
 gsl\_linalg\_complex\_LU\_svx (*C function*), 141  
 gsl\_linalg\_complex\_QR\_decomp (*C function*), 145  
 gsl\_linalg\_complex\_QR\_decomp\_r (*C function*), 143  
 gsl\_linalg\_complex\_QR\_lassolve (*C function*), 145  
 gsl\_linalg\_complex\_QR\_lassolve\_r (*C function*), 144  
 gsl\_linalg\_complex\_QR\_QHvec (*C function*), 145  
 gsl\_linalg\_complex\_QR\_QHvec\_r (*C function*), 144  
 gsl\_linalg\_complex\_QR\_Qvec (*C function*), 145  
 gsl\_linalg\_complex\_QR\_solve (*C function*), 145  
 gsl\_linalg\_complex\_QR\_solve\_r (*C function*), 143  
 gsl\_linalg\_complex\_QR\_svx (*C function*), 145  
 gsl\_linalg\_complex\_QR\_unpack\_r (*C function*), 144  
 gsl\_linalg\_complex\_tri\_invert (*C function*), 163  
 gsl\_linalg\_complex\_tri\_LHL (*C function*), 163  
 gsl\_linalg\_complex\_tri\_UL (*C function*), 163  
 gsl\_linalg\_givens (*C function*), 161  
 gsl\_linalg\_givens\_gv (*C function*), 161  
 gsl\_linalg\_hermted\_decomp (*C function*), 159  
 gsl\_linalg\_hermted\_unpack (*C function*), 159  
 gsl\_linalg\_hermted\_unpack\_T (*C function*), 159  
 gsl\_linalg\_hessenberg\_decomp (*C function*), 159  
 gsl\_linalg\_hessenberg\_set\_zero (*C function*), 160  
 gsl\_linalg\_hessenberg\_unpack (*C function*), 159  
 gsl\_linalg\_hessenberg\_unpack\_accum (*C function*), 159  
 gsl\_linalg\_hesstri\_decomp (*C function*), 160  
 gsl\_linalg\_HH\_solve (*C function*), 162  
 gsl\_linalg\_HH\_svx (*C function*), 162  
 gsl\_linalg\_householder\_hm (*C function*), 161  
 gsl\_linalg\_householder\_hv (*C function*), 162  
 gsl\_linalg\_householder\_mh (*C function*), 162  
 gsl\_linalg\_householder\_transform (*C function*), 161  
 gsl\_linalg\_ldlt\_band\_decomp (*C function*), 167  
 gsl\_linalg\_ldlt\_band\_rcond (*C function*), 167  
 gsl\_linalg\_ldlt\_band\_solve (*C function*), 167  
 gsl\_linalg\_ldlt\_band\_svx (*C function*), 167  
 gsl\_linalg\_ldlt\_band\_unpack (*C function*), 167  
 gsl\_linalg\_ldlt\_decomp (*C function*), 158  
 gsl\_linalg\_ldlt\_rcond (*C function*), 158  
 gsl\_linalg\_ldlt\_solve (*C function*), 158  
 gsl\_linalg\_ldlt\_svx (*C function*), 158  
 gsl\_linalg\_LQ\_decomp (*C function*), 151  
 gsl\_linalg\_LQ\_lassolve (*C function*), 151  
 gsl\_linalg\_LQ\_QTvec (*C function*), 151  
 gsl\_linalg\_LQ\_unpack (*C function*), 151  
 gsl\_linalg\_LU\_band\_decomp (*C function*), 165  
 gsl\_linalg\_LU\_band\_solve (*C function*), 165  
 gsl\_linalg\_LU\_band\_svx (*C function*), 166  
 gsl\_linalg\_LU\_band\_unpack (*C function*), 166

- gsl\_linalg\_LU\_decomp (C function), 141  
 gsl\_linalg\_LU\_det (C function), 142  
 gsl\_linalg\_LU\_invert (C function), 142  
 gsl\_linalg\_LU\_invx (C function), 142  
 gsl\_linalg\_LU\_lndet (C function), 142  
 gsl\_linalg\_LU\_refine (C function), 142  
 gsl\_linalg\_LU\_sgndet (C function), 142  
 gsl\_linalg\_LU\_solve (C function), 141  
 gsl\_linalg\_LU\_svx (C function), 141  
 gsl\_linalg\_mcholesky\_decomp (C function), 157  
 gsl\_linalg\_mcholesky\_rcond (C function), 157  
 gsl\_linalg\_mcholesky\_solve (C function), 157  
 gsl\_linalg\_mcholesky\_svx (C function), 157  
 gsl\_linalg\_pcholesky\_decomp (C function), 156  
 gsl\_linalg\_pcholesky\_decomp2 (C function), 156  
 gsl\_linalg\_pcholesky\_invert (C function), 157  
 gsl\_linalg\_pcholesky\_rcond (C function), 157  
 gsl\_linalg\_pcholesky\_solve (C function), 156  
 gsl\_linalg\_pcholesky\_solve2 (C function), 156  
 gsl\_linalg\_pcholesky\_svx (C function), 156  
 gsl\_linalg\_pcholesky\_svx2 (C function), 156  
 gsl\_linalg\_QL\_decomp (C function), 151  
 gsl\_linalg\_QL\_unpack (C function), 151  
 gsl\_linalg\_QR\_decomp (C function), 145  
 gsl\_linalg\_QR\_decomp\_r (C function), 143  
 gsl\_linalg\_QR\_lssolve (C function), 145  
 gsl\_linalg\_QR\_lssolve\_r (C function), 144  
 gsl\_linalg\_QR\_QRsolve (C function), 146  
 gsl\_linalg\_QR\_QTmat (C function), 146  
 gsl\_linalg\_QR\_QTmat\_r (C function), 144  
 gsl\_linalg\_QR\_QTvec (C function), 145  
 gsl\_linalg\_QR\_QTvec\_r (C function), 144  
 gsl\_linalg\_QR\_Qvec (C function), 145  
 gsl\_linalg\_QR\_rcond (C function), 144  
 gsl\_linalg\_QR\_Rsolve (C function), 146  
 gsl\_linalg\_QR\_Rsvx (C function), 146  
 gsl\_linalg\_QR\_solve (C function), 145  
 gsl\_linalg\_QR\_solve\_r (C function), 143  
 gsl\_linalg\_QR\_svx (C function), 145  
 gsl\_linalg\_QR\_UD\_decomp (C function), 148  
 gsl\_linalg\_QR\_UD\_lssolve (C function), 148  
 gsl\_linalg\_QR\_unpack (C function), 146  
 gsl\_linalg\_QR\_unpack\_r (C function), 144  
 gsl\_linalg\_QR\_update (C function), 146  
 gsl\_linalg\_QR\_UR\_decomp (C function), 146  
 gsl\_linalg\_QR\_UU\_decomp (C function), 147  
 gsl\_linalg\_QR\_UU\_lssolve (C function), 147  
 gsl\_linalg\_QR\_UU\_QTec (C function), 147  
 gsl\_linalg\_QR\_UZ\_decomp (C function), 148  
 gsl\_linalg\_QRPT\_decomp (C function), 149  
 gsl\_linalg\_QRPT\_decomp2 (C function), 149  
 gsl\_linalg\_QRPT\_lssolve (C function), 149  
 gsl\_linalg\_QRPT\_lssolve2 (C function), 150  
 gsl\_linalg\_QRPT\_QRsolve (C function), 150  
 gsl\_linalg\_QRPT\_rank (C function), 150  
 gsl\_linalg\_QRPT\_rcond (C function), 150  
 gsl\_linalg\_QRPT\_Rsolve (C function), 150  
 gsl\_linalg\_QRPT\_Rsvx (C function), 150  
 gsl\_linalg\_QRPT\_solve (C function), 149  
 gsl\_linalg\_QRPT\_svx (C function), 149  
 gsl\_linalg\_QRPT\_update (C function), 150  
 gsl\_linalg\_R\_solve (C function), 146  
 gsl\_linalg\_R\_svx (C function), 146  
 gsl\_linalg\_solve\_cyc\_tridiag (C function), 163  
 gsl\_linalg\_solve\_symm\_cyc\_tridiag (C function), 163  
 gsl\_linalg\_solve\_symm\_tridiag (C function), 162  
 gsl\_linalg\_solve\_tridiag (C function), 162  
 gsl\_linalg\_SV\_decomp (C function), 153  
 gsl\_linalg\_SV\_decomp\_jacobi (C function), 154  
 gsl\_linalg\_SV\_decomp\_mod (C function), 154  
 gsl\_linalg\_SV\_leverage (C function), 154  
 gsl\_linalg\_SV\_solve (C function), 154  
 gsl\_linalg\_symmtd\_decomp (C function), 158  
 gsl\_linalg\_symmtd\_unpack (C function), 158  
 gsl\_linalg\_symmtd\_unpack\_T (C function), 159  
 gsl\_linalg\_tri\_invert (C function), 163  
 gsl\_linalg\_tri\_LTL (C function), 163  
 gsl\_linalg\_tri\_rcond (C function), 163  
 gsl\_linalg\_tri\_UL (C function), 163  
 gsl\_log1p (C function), 18  
 gsl\_matrix (C type), 94  
 gsl\_matrix\_add (C function), 101  
 gsl\_matrix\_add\_constant (C function), 101  
 gsl\_matrix\_alloc (C function), 95  
 gsl\_matrix\_calloc (C function), 95  
 gsl\_matrix\_column (C function), 98  
 gsl\_matrix\_complex\_conjtrans\_memcpy (C function), 100  
 gsl\_matrix\_const\_column (C function), 98  
 gsl\_matrix\_const\_diagonal (C function), 99  
 gsl\_matrix\_const\_ptr (C function), 95  
 gsl\_matrix\_const\_row (C function), 98  
 gsl\_matrix\_const\_subcolumn (C function), 99  
 gsl\_matrix\_const\_subdiagonal (C function), 99  
 gsl\_matrix\_const\_submatrix (C function), 96  
 gsl\_matrix\_const\_subrow (C function), 99  
 gsl\_matrix\_const\_superdiagonal (C function), 99  
 gsl\_matrix\_const\_view (C type), 96  
 gsl\_matrix\_const\_view\_array (C function), 97  
 gsl\_matrix\_const\_view\_array\_with\_tda (C function), 97  
 gsl\_matrix\_const\_view\_vector (C function), 98  
 gsl\_matrix\_const\_view\_vector\_with\_tda (C function), 98  
 gsl\_matrix\_diagonal (C function), 99  
 gsl\_matrix\_div\_elements (C function), 101  
 gsl\_matrix\_equal (C function), 102

- gsl\_matrix\_fprintf (C function), 96  
 gsl\_matrix\_fread (C function), 96  
 gsl\_matrix\_free (C function), 95  
 gsl\_matrix\_fscanf (C function), 96  
 gsl\_matrix\_fwrite (C function), 96  
 gsl\_matrix\_get (C function), 95  
 gsl\_matrix\_get\_col (C function), 100  
 gsl\_matrix\_get\_row (C function), 100  
 gsl\_matrix\_isneg (C function), 102  
 gsl\_matrix\_isononneg (C function), 102  
 gsl\_matrix\_isnull (C function), 102  
 gsl\_matrix\_ispos (C function), 102  
 gsl\_matrix\_max (C function), 102  
 gsl\_matrix\_max\_index (C function), 102  
 gsl\_matrix\_memcpy (C function), 100  
 gsl\_matrix\_min (C function), 102  
 gsl\_matrix\_min\_index (C function), 102  
 gsl\_matrix\_minmax (C function), 102  
 gsl\_matrix\_minmax\_index (C function), 102  
 gsl\_matrix\_mul\_elements (C function), 101  
 gsl\_matrix\_norm1 (C function), 102  
 gsl\_matrix\_ptr (C function), 95  
 gsl\_matrix\_row (C function), 98  
 gsl\_matrix\_scale (C function), 101  
 gsl\_matrix\_scale\_columns (C function), 101  
 gsl\_matrix\_scale\_rows (C function), 101  
 gsl\_matrix\_set (C function), 95  
 gsl\_matrix\_set\_all (C function), 96  
 gsl\_matrix\_set\_col (C function), 100  
 gsl\_matrix\_set\_identity (C function), 96  
 gsl\_matrix\_set\_row (C function), 100  
 gsl\_matrix\_set\_zero (C function), 96  
 gsl\_matrix\_sub (C function), 101  
 gsl\_matrix\_subcolumn (C function), 99  
 gsl\_matrix\_subdiagonal (C function), 99  
 gsl\_matrix\_submatrix (C function), 96  
 gsl\_matrix\_subrow (C function), 99  
 gsl\_matrix\_superdiagonal (C function), 99  
 gsl\_matrix\_swap (C function), 100  
 gsl\_matrix\_swap\_columns (C function), 100  
 gsl\_matrix\_swap\_rowcol (C function), 100  
 gsl\_matrix\_swap\_rows (C function), 100  
 gsl\_matrix\_transpose (C function), 100  
 gsl\_matrix\_transpose\_memcpy (C function), 100  
 gsl\_matrix\_view (C type), 96  
 gsl\_matrix\_view\_array (C function), 97  
 gsl\_matrix\_view\_array\_with\_tda (C function), 97  
 gsl\_matrix\_view\_vector (C function), 98  
 gsl\_matrix\_view\_vector\_with\_tda (C function), 98  
 GSL\_MAX (C macro), 20  
 GSL\_MAX\_DBL (C function), 20  
 GSL\_MAX\_INT (C function), 20  
 GSL\_MAX\_LDBL (C function), 20  
 GSL\_MIN (C macro), 20  
 GSL\_MIN\_DBL (C function), 20  
 gsl\_min\_fminimizer (C type), 448  
 gsl\_min\_fminimizer\_alloc (C function), 448  
 gsl\_min\_fminimizer\_f\_lower (C function), 449  
 gsl\_min\_fminimizer\_f\_minimum (C function), 449  
 gsl\_min\_fminimizer\_f\_upper (C function), 449  
 gsl\_min\_fminimizer\_free (C function), 449  
 gsl\_min\_fminimizer\_iterate (C function), 449  
 gsl\_min\_fminimizer\_name (C function), 449  
 gsl\_min\_fminimizer\_set (C function), 448  
 gsl\_min\_fminimizer\_set\_with\_values (C function), 448  
 gsl\_min\_fminimizer\_type (C type), 450  
 gsl\_min\_fminimizer\_type.gsl\_min\_fminimizer\_brent (C var), 450  
 gsl\_min\_fminimizer\_type.gsl\_min\_fminimizer\_goldensection (C var), 450  
 gsl\_min\_fminimizer\_type.gsl\_min\_fminimizer\_quad\_golden (C var), 451  
 gsl\_min\_fminimizer\_x\_lower (C function), 449  
 gsl\_min\_fminimizer\_x\_minimum (C function), 449  
 gsl\_min\_fminimizer\_x\_upper (C function), 449  
 GSL\_MIN\_INT (C function), 20  
 GSL\_MIN\_LDBL (C function), 20  
 gsl\_min\_test\_interval (C function), 450  
 gsl\_mode\_t (C type), 36  
 gsl\_mode\_t.GSL\_PREC\_APPROX (C macro), 36  
 gsl\_mode\_t.GSL\_PREC\_DOUBLE (C macro), 36  
 gsl\_mode\_t.GSL\_PREC\_SINGLE (C macro), 36  
 gsl\_monte\_function (C type), 357  
 gsl\_monte\_miser\_alloc (C function), 359  
 gsl\_monte\_miser\_free (C function), 360  
 gsl\_monte\_miser\_init (C function), 359  
 gsl\_monte\_miser\_integrate (C function), 360  
 gsl\_monte\_miser\_params (C type), 360  
 gsl\_monte\_miser\_params.alpha (C var), 360  
 gsl\_monte\_miser\_params.dither (C var), 360  
 gsl\_monte\_miser\_params.estimate\_frac (C var), 360  
 gsl\_monte\_miser\_params.min\_calls (C var), 360  
 gsl\_monte\_miser\_params.min\_calls\_per\_bisection (C var), 360  
 gsl\_monte\_miser\_params\_get (C function), 360  
 gsl\_monte\_miser\_params\_set (C function), 360  
 gsl\_monte\_miser\_state (C type), 359  
 gsl\_monte\_plain\_alloc (C function), 358  
 gsl\_monte\_plain\_free (C function), 359  
 gsl\_monte\_plain\_init (C function), 359  
 gsl\_monte\_plain\_integrate (C function), 359  
 gsl\_monte\_plain\_state (C type), 358  
 gsl\_monte\_vegas\_alloc (C function), 361  
 gsl\_monte\_vegas\_chisq (C function), 362  
 gsl\_monte\_vegas\_free (C function), 361  
 gsl\_monte\_vegas\_init (C function), 361



- gsl\_monte\_vegas\_integrate (*C function*), 361  
 gsl\_monte\_vegas\_params (*C type*), 362  
 gsl\_monte\_vegas\_params.alpha (*C var*), 362  
 gsl\_monte\_vegas\_params.iterations (*C var*), 362  
 gsl\_monte\_vegas\_params.mode (*C var*), 363  
 gsl\_monte\_vegas\_params ostream (*C var*), 363  
 gsl\_monte\_vegas\_params.stage (*C var*), 362  
 gsl\_monte\_vegas\_params.verbose (*C var*), 363  
 gsl\_monte\_vegas\_params\_get (*C function*), 362  
 gsl\_monte\_vegas\_params\_set (*C function*), 362  
 gsl\_monte\_vegas\_runval (*C function*), 362  
 gsl\_monte\_vegas\_state (*C type*), 361  
 gsl\_movstat\_accum (*C type*), 310  
 gsl\_movstat\_accum.delete (*C member*), 311  
 gsl\_movstat\_accum.get (*C member*), 311  
 gsl\_movstat\_accum.init (*C member*), 311  
 gsl\_movstat\_accum.insert (*C member*), 311  
 gsl\_movstat\_accum.size (*C member*), 311  
 gsl\_movstat\_accum\_max (*C var*), 311  
 gsl\_movstat\_accum\_mean (*C var*), 311  
 gsl\_movstat\_accum\_median (*C var*), 311  
 gsl\_movstat\_accum\_min (*C var*), 311  
 gsl\_movstat\_accum\_minmax (*C var*), 311  
 gsl\_movstat\_accum\_Qn (*C var*), 311  
 gsl\_movstat\_accum\_qqr (*C var*), 311  
 gsl\_movstat\_accum\_sd (*C var*), 311  
 gsl\_movstat\_accum\_Sn (*C var*), 311  
 gsl\_movstat\_accum\_sum (*C var*), 311  
 gsl\_movstat\_accum\_variance (*C var*), 311  
 gsl\_movstat\_alloc (*C function*), 306  
 gsl\_movstat\_alloc2 (*C function*), 306  
 gsl\_movstat\_apply (*C function*), 310  
 gsl\_movstat\_end\_t (*C type*), 305  
 gsl\_movstat\_end\_t.GSL\_MOVSTAT\_END\_PADVALUE  
   (*C macro*), 305  
 gsl\_movstat\_end\_t.GSL\_MOVSTAT\_END\_PADZERO (*C  
   macro*), 305  
 gsl\_movstat\_end\_t.GSL\_MOVSTAT\_END\_TRUNCATE  
   (*C macro*), 306  
 gsl\_movstat\_fill (*C function*), 310  
 gsl\_movstat\_free (*C function*), 306  
 gsl\_movstat\_function (*C type*), 310  
 gsl\_movstat\_function.function (*C member*), 310  
 gsl\_movstat\_function.params (*C member*), 310  
 gsl\_movstat\_mad (*C function*), 308  
 gsl\_movstat\_mad0 (*C function*), 308  
 gsl\_movstat\_max (*C function*), 307  
 gsl\_movstat\_mean (*C function*), 306  
 gsl\_movstat\_median (*C function*), 308  
 gsl\_movstat\_min (*C function*), 307  
 gsl\_movstat\_minmax (*C function*), 307  
 gsl\_movstat\_Qn (*C function*), 309  
 gsl\_movstat\_qqr (*C function*), 309  
 gsl\_movstat\_sd (*C function*), 307  
 gsl\_movstat\_Sn (*C function*), 309  
 gsl\_movstat\_sum (*C function*), 308  
 gsl\_movstat\_variance (*C function*), 307  
 gsl\_movstat\_workspace (*C type*), 306  
 gsl\_multifit\_linear (*C function*), 484  
 gsl\_multifit\_linear\_alloc (*C function*), 484  
 gsl\_multifit\_linear\_applyW (*C function*), 488  
 gsl\_multifit\_linear\_bsvd (*C function*), 484  
 gsl\_multifit\_linear\_est (*C function*), 485  
 gsl\_multifit\_linear\_free (*C function*), 484  
 gsl\_multifit\_linear\_gcv (*C function*), 489  
 gsl\_multifit\_linear\_gcv\_calc (*C function*), 489  
 gsl\_multifit\_linear\_gcv\_curve (*C function*), 489  
 gsl\_multifit\_linear\_gcv\_init (*C function*), 488  
 gsl\_multifit\_linear\_gcv\_min (*C function*), 489  
 gsl\_multifit\_linear\_genform1 (*C function*), 487  
 gsl\_multifit\_linear\_genform2 (*C function*), 487  
 gsl\_multifit\_linear\_L\_decomp (*C function*), 487  
 gsl\_multifit\_linear\_lcorner (*C function*), 488  
 gsl\_multifit\_linear\_lcorner2 (*C function*), 488  
 gsl\_multifit\_linear\_lcurvature (*C function*), 488  
 gsl\_multifit\_linear\_lcurve (*C function*), 488  
 gsl\_multifit\_linear\_Lk (*C function*), 489  
 gsl\_multifit\_linear\_Lsobolev (*C function*), 489  
 gsl\_multifit\_linear\_rank (*C function*), 485  
 gsl\_multifit\_linear\_rcond (*C function*), 489  
 gsl\_multifit\_linear\_residuals (*C function*), 485  
 gsl\_multifit\_linear\_solve (*C function*), 487  
 gsl\_multifit\_linear\_stdform1 (*C function*), 486  
 gsl\_multifit\_linear\_stdform2 (*C function*), 487  
 gsl\_multifit\_linear\_svd (*C function*), 484  
 gsl\_multifit\_linear\_tsvd (*C function*), 484  
 gsl\_multifit\_linear\_wgenform2 (*C function*), 487  
 gsl\_multifit\_linear\_workspace (*C type*), 483  
 gsl\_multifit\_linear\_wstdform1 (*C function*), 486  
 gsl\_multifit\_linear\_wstdform2 (*C function*), 487  
 gsl\_multifit\_nlinear\_alloc (*C function*), 529  
 gsl\_multifit\_nlinear\_avratio (*C function*), 533  
 gsl\_multifit\_nlinear\_covar (*C function*), 535  
 gsl\_multifit\_nlinear\_default\_parameters (*C  
   function*), 529  
 gsl\_multifit\_nlinear\_driver (*C function*), 534  
 gsl\_multifit\_nlinear\_fdf (*C type*), 530  
 gsl\_multifit\_nlinear\_fdtype (*C type*), 528  
 gsl\_multifit\_nlinear\_fdtype.GSL\_MULTIFIT\_NLINEAR\_CTRDIFF  
   (*C macro*), 528  
 gsl\_multifit\_nlinear\_fdtype.GSL\_MULTIFIT\_NLINEAR\_FWDIFF  
   (*C macro*), 528  
 gsl\_multifit\_nlinear\_free (*C function*), 530  
 gsl\_multifit\_nlinear\_init (*C function*), 529  
 gsl\_multifit\_nlinear\_iterate (*C function*), 532  
 gsl\_multifit\_nlinear\_jac (*C function*), 533  
 gsl\_multifit\_nlinear\_name (*C function*), 530  
 gsl\_multifit\_nlinear\_niter (*C function*), 533



gsl\_multilarge\_nlinear\_solver.gsl\_multifit\_nlinear\_subline\_qdfminimizer\_type.gsl\_multimin\_fdfminimizer\_v  
 (C var), 527 (C var), 474  
 gsl\_multilarge\_nlinear\_solver.gsl\_multifit\_nlinear\_subline\_sdfminimizer\_type.gsl\_multimin\_fdfminimizer\_v  
 (C var), 528 (C var), 474  
 gsl\_multilarge\_nlinear\_solver.gsl\_multilarge\_nshear\_solver\_fdfminimizer\_type.gsl\_multimin\_fdfminimizer\_v  
 (C var), 527 (C var), 474  
 gsl\_multilarge\_nlinear\_solver.gsl\_multilarge\_nshear\_solver\_fdfminimizer\_x (C function), 473  
 (C var), 527 gsl\_multimin\_fminimizer (C type), 470  
 gsl\_multilarge\_nlinear\_test (C function), 534 gsl\_multimin\_fminimizer\_alloc (C function), 470  
 gsl\_multilarge\_nlinear\_trs (C type), 526 gsl\_multimin\_fminimizer\_free (C function), 470  
 gsl\_multilarge\_nlinear\_trs.gsl\_multifit\_nlineargsl\_multimin\_fminimizer\_iterate (C function),  
 (C var), 526 473  
 gsl\_multilarge\_nlinear\_trs.gsl\_multifit\_nlineargsl\_multimin\_fminimizer\_minimum (C function),  
 (C var), 526 473  
 gsl\_multilarge\_nlinear\_trs.gsl\_multifit\_nlineargsl\_multimin\_fminimizer\_name (C function), 470  
 (C var), 526 gsl\_multimin\_fminimizer\_set (C function), 470  
 gsl\_multilarge\_nlinear\_trs.gsl\_multifit\_nlineargsl\_multimin\_fminimizer\_size (C function), 473  
 (C var), 526 gsl\_multimin\_fminimizer\_type (C type), 475  
 gsl\_multilarge\_nlinear\_trs.gsl\_multifit\_nlineargsl\_multimin\_fminimizer\_type.gsl\_multimin\_fminimizer\_nmsin  
 (C var), 526 (C var), 475  
 gsl\_multilarge\_nlinear\_trs.gsl\_multilarge\_nlineargsl\_multimin\_fminimizer\_type.gsl\_multimin\_fminimizer\_nmsin  
 (C var), 526 (C var), 475  
 gsl\_multilarge\_nlinear\_trs.gsl\_multilarge\_nlineargsl\_multimin\_fminimizer\_type.gsl\_multimin\_fminimizer\_nmsin  
 (C var), 526 (C var), 475  
 gsl\_multilarge\_nlinear\_trs.gsl\_multilarge\_nlineargsl\_multimin\_fminimizer\_x (C function), 473  
 (C var), 526 gsl\_multimin\_function (C type), 471  
 gsl\_multilarge\_nlinear\_trs.gsl\_multilarge\_nlineargsl\_multimin\_function\_fdf (C type), 471  
 (C var), 526 gsl\_multimin\_test\_gradient (C function), 473  
 gsl\_multilarge\_nlinear\_trs.gsl\_multilarge\_nlineargsl\_multimin\_fminimizer\_size (C function), 474  
 (C var), 526 gsl\_multiroot\_fdfsolver (C type), 456  
 gsl\_multilarge\_nlinear\_trs.gsl\_multilarge\_nlineargsl\_multimin\_fminimizer\_type.gsl\_multiroot\_fdfsolver\_alloc (C function), 456  
 (C var), 526 gsl\_multiroot\_fdfsolver\_dx (C function), 459  
 gsl\_multilarge\_nlinear\_trs\_name (C function), 530 gsl\_multiroot\_fdfsolver\_f (C function), 459  
 gsl\_multimin\_fdfminimizer (C type), 470 gsl\_multiroot\_fdfsolver\_free (C function), 456  
 gsl\_multimin\_fdfminimizer\_alloc (C function), 470 gsl\_multiroot\_fdfsolver\_iterate (C function),  
 470 459  
 gsl\_multimin\_fdfminimizer\_dx (C function), 473 gsl\_multiroot\_fdfsolver\_name (C function), 456  
 gsl\_multimin\_fdfminimizer\_free (C function), 470 gsl\_multiroot\_fdfsolver\_root (C function), 459  
 gsl\_multimin\_fdfminimizer\_gradient (C function), 473 gsl\_multiroot\_fdfsolver\_set (C function), 456  
 gsl\_multimin\_fdfminimizer\_iterate (C function), 473 gsl\_multiroot\_fdfsolver\_type (C type), 460  
 gsl\_multimin\_fdfminimizer\_minimum (C function), 473 gsl\_multiroot\_fdfsolver\_type.gsl\_multiroot\_fdfsolver\_gnewt  
 (C var), 461  
 gsl\_multimin\_fdfminimizer\_name (C function), 470 gsl\_multiroot\_fdfsolver\_type.gsl\_multiroot\_fdfsolver\_hybrid  
 (C var), 461  
 gsl\_multimin\_fdfminimizer\_restart (C function), 473 gsl\_multiroot\_fdfsolver\_type.gsl\_multiroot\_fdfsolver\_hybrid  
 (C var), 460  
 gsl\_multimin\_fdfminimizer\_set (C function), 470 gsl\_multiroot\_fdfsolver\_type.gsl\_multiroot\_fdfsolver\_newt  
 (C var), 461  
 gsl\_multimin\_fdfminimizer\_type (C type), 474 gsl\_multiroot\_fsolver (C type), 456  
 gsl\_multimin\_fdfminimizer\_type.gsl\_multimin\_fdfminimizer\_type.gsl\_multiroot\_fsolver\_alloc (C function), 456  
 (C var), 474 gsl\_multiroot\_fsolver\_iterate (C function), 459  
 gsl\_multimin\_fdfminimizer\_type.gsl\_multimin\_fdfminimizer\_type.gsl\_multiroot\_fsolver\_f (C function), 459  
 (C var), 474 gsl\_multiroot\_fsolver\_free (C function), 456  
 gsl\_multimin\_fdfminimizer\_type.gsl\_multimin\_fdfminimizer\_type.gsl\_multiroot\_fsolver\_iterate (C function), 459  
 (C var), 474

- gsl\_multiroot\_fsolver\_name (*C function*), 456  
 gsl\_multiroot\_fsolver\_root (*C function*), 459  
 gsl\_multiroot\_fsolver\_set (*C function*), 456  
 gsl\_multiroot\_fsolver\_type (*C type*), 462  
 gsl\_multiroot\_fsolver\_type.gsl\_multiroot\_fsolver\_type  
 (C var), 462  
 gsl\_multiroot\_fsolver\_type.gsl\_multiroot\_fsolver\_type  
 (C var), 462  
 gsl\_multiroot\_fsolver\_type.gsl\_multiroot\_fsolver\_type  
 (C var), 462  
 gsl\_multiroot\_fsolver\_type.gsl\_multiroot\_fsolver\_type  
 (C var), 462  
 gsl\_multiroot\_function (*C type*), 457  
 gsl\_multiroot\_function\_fdf (*C type*), 457  
 gsl\_multiroot\_test\_delta (*C function*), 460  
 gsl\_multiroot\_test\_residual (*C function*), 460  
 gsl\_multiset (*C type*), 117  
 gsl\_multiset\_alloc (*C function*), 117  
 gsl\_multiset\_calloc (*C function*), 117  
 gsl\_multiset\_data (*C function*), 118  
 gsl\_multiset\_fprintf (*C function*), 119  
 gsl\_multiset\_fread (*C function*), 119  
 gsl\_multiset\_free (*C function*), 117  
 gsl\_multiset\_fscanf (*C function*), 119  
 gsl\_multiset\_fwrite (*C function*), 119  
 gsl\_multiset\_get (*C function*), 118  
 gsl\_multiset\_init\_first (*C function*), 117  
 gsl\_multiset\_init\_last (*C function*), 117  
 gsl\_multiset\_k (*C function*), 118  
 gsl\_multiset\_memcpy (*C function*), 118  
 gsl\_multiset\_n (*C function*), 118  
 gsl\_multiset\_next (*C function*), 118  
 gsl\_multiset\_prev (*C function*), 118  
 gsl\_multiset\_valid (*C function*), 118  
 GSL\_NAN (*C macro*), 18  
 GSL\_NEGINF (*C macro*), 18  
 gsl\_ntuple (*C type*), 351  
 gsl\_ntuple\_bookdata (*C function*), 352  
 gsl\_ntuple\_close (*C function*), 352  
 gsl\_ntuple\_create (*C function*), 351  
 gsl\_ntuple\_open (*C function*), 352  
 gsl\_ntuple\_project (*C function*), 353  
 gsl\_ntuple\_read (*C function*), 352  
 gsl\_ntuple\_select\_fn (*C type*), 352  
 gsl\_ntuple\_value\_fn (*C type*), 352  
 gsl\_ntuple\_write (*C function*), 352  
 gsl\_odeiv2\_control (*C type*), 386  
 gsl\_odeiv2\_control\_alloc (*C function*), 387  
 gsl\_odeiv2\_control\_errlevel (*C function*), 387  
 gsl\_odeiv2\_control\_free (*C function*), 387  
 gsl\_odeiv2\_control\_hadjust (*C function*), 387  
 gsl\_odeiv2\_control\_init (*C function*), 387  
 gsl\_odeiv2\_control\_name (*C function*), 387  
 gsl\_odeiv2\_control\_scaled\_new (*C function*), 387  
 gsl\_odeiv2\_control\_set\_driver (*C function*), 387  
 gsl\_odeiv2\_control\_standard\_new (*C function*),  
 386  
 gsl\_odeiv2\_control\_type (*C type*), 386  
 gsl\_odeiv2\_control\_y\_new (*C function*), 386  
 gsl\_odeiv2\_control\_yp\_new (*C function*), 386  
 gsl\_odeiv2\_driver\_alloc\_scaled\_new (*C func-*  
*tion*), 389  
 gsl\_odeiv2\_driver\_alloc\_standard\_new (*C func-*  
*tion*), 389  
 gsl\_odeiv2\_driver\_alloc\_y\_new (*C function*), 389  
 gsl\_odeiv2\_driver\_alloc\_yp\_new (*C function*), 389  
 gsl\_odeiv2\_driver\_apply (*C function*), 389  
 gsl\_odeiv2\_driver\_apply\_fixed\_step (*C func-*  
*tion*), 389  
 gsl\_odeiv2\_driver\_free (*C function*), 390  
 gsl\_odeiv2\_driver\_reset (*C function*), 389  
 gsl\_odeiv2\_driver\_reset\_hstart (*C function*), 389  
 gsl\_odeiv2\_driver\_set\_hmax (*C function*), 389  
 gsl\_odeiv2\_driver\_set\_hmin (*C function*), 389  
 gsl\_odeiv2\_driver\_set\_nmax (*C function*), 389  
 gsl\_odeiv2\_evolve (*C type*), 388  
 gsl\_odeiv2\_evolve\_alloc (*C function*), 388  
 gsl\_odeiv2\_evolve\_apply (*C function*), 388  
 gsl\_odeiv2\_evolve\_apply\_fixed\_step (*C func-*  
*tion*), 388  
 gsl\_odeiv2\_evolve\_free (*C function*), 388  
 gsl\_odeiv2\_evolve\_reset (*C function*), 388  
 gsl\_odeiv2\_evolve\_set\_driver (*C function*), 388  
 gsl\_odeiv2\_step (*C type*), 384  
 gsl\_odeiv2\_step\_alloc (*C function*), 384  
 gsl\_odeiv2\_step\_apply (*C function*), 384  
 gsl\_odeiv2\_step\_free (*C function*), 384  
 gsl\_odeiv2\_step\_name (*C function*), 384  
 gsl\_odeiv2\_step\_order (*C function*), 384  
 gsl\_odeiv2\_step\_reset (*C function*), 384  
 gsl\_odeiv2\_step\_set\_driver (*C function*), 384  
 gsl\_odeiv2\_step\_type (*C type*), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_bsimp  
 (C var), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_msadams  
 (C var), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_msbdf  
 (C var), 386  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rk1imp  
 (C var), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rk2 (C  
 var), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rk2imp  
 (C var), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rk4 (C  
 var), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rk4imp  
 (C var), 385



- gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rk8pd  
   (*C var*), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rkck (*C var*), 385  
 gsl\_odeiv2\_step\_type.gsl\_odeiv2\_step\_rkf45  
   (*C var*), 385  
 gsl\_odeiv2\_system (*C type*), 383  
 gsl\_permutation (*C type*), 107  
 gsl\_permutation\_alloc (*C function*), 107  
 gsl\_permutation\_calloc (*C function*), 107  
 gsl\_permutation\_canonical\_cycles (*C function*),  
   110  
 gsl\_permutation\_canonical\_to\_linear (*C func-*  
*tion*), 110  
 gsl\_permutation\_data (*C function*), 108  
 gsl\_permutation\_fprintf (*C function*), 109  
 gsl\_permutation\_fread (*C function*), 109  
 gsl\_permutation\_free (*C function*), 107  
 gsl\_permutation\_fscanf (*C function*), 109  
 gsl\_permutation\_fwrite (*C function*), 109  
 gsl\_permutation\_get (*C function*), 108  
 gsl\_permutation\_init (*C function*), 107  
 gsl\_permutation\_inverse (*C function*), 108  
 gsl\_permutation\_inversions (*C function*), 110  
 gsl\_permutation\_linear\_cycles (*C function*), 110  
 gsl\_permutation\_linear\_to\_canonical (*C func-*  
*tion*), 110  
 gsl\_permutation\_memcpy (*C function*), 108  
 gsl\_permutation\_mul (*C function*), 109  
 gsl\_permutation\_next (*C function*), 108  
 gsl\_permutation\_prev (*C function*), 108  
 gsl\_permutation\_reverse (*C function*), 108  
 gsl\_permutation\_size (*C function*), 108  
 gsl\_permutation\_swap (*C function*), 108  
 gsl\_permutation\_valid (*C function*), 108  
 gsl\_permute (*C function*), 109  
 gsl\_permute\_inverse (*C function*), 109  
 gsl\_permute\_matrix (*C function*), 109  
 gsl\_permute\_vector (*C function*), 109  
 gsl\_permute\_vector\_inverse (*C function*), 109  
 gsl\_poly\_complex\_eval (*C function*), 29  
 gsl\_poly\_complex\_solve (*C function*), 31  
 gsl\_poly\_complex\_solve\_cubic (*C function*), 31  
 gsl\_poly\_complex\_solve\_quadratic (*C function*),  
   30  
 gsl\_poly\_complex\_workspace (*C type*), 31  
 gsl\_poly\_complex\_workspace\_alloc (*C function*),  
   31  
 gsl\_poly\_complex\_workspace\_free (*C function*), 31  
 gsl\_poly\_dd\_eval (*C function*), 30  
 gsl\_poly\_dd\_hermite\_init (*C function*), 30  
 gsl\_poly\_dd\_init (*C function*), 30  
 gsl\_poly\_dd\_taylor (*C function*), 30  
 gsl\_poly\_eval (*C function*), 29  
 gsl\_poly\_eval\_derivs (*C function*), 29  
 gsl\_poly\_solve\_cubic (*C function*), 31  
 gsl\_poly\_solve\_quadratic (*C function*), 30  
 GSL\_POSINF (*C macro*), 18  
 gsl\_pow\_2 (*C function*), 19  
 gsl\_pow\_3 (*C function*), 19  
 gsl\_pow\_4 (*C function*), 19  
 gsl\_pow\_5 (*C function*), 19  
 gsl\_pow\_6 (*C function*), 19  
 gsl\_pow\_7 (*C function*), 19  
 gsl\_pow\_8 (*C function*), 19  
 gsl\_pow\_9 (*C function*), 19  
 gsl\_pow\_int (*C function*), 19  
 gsl\_pow\_uint (*C function*), 19  
 gsl\_qrng (*C type*), 231  
 gsl\_qrng\_alloc (*C function*), 231  
 gsl\_qrng\_clone (*C function*), 232  
 gsl\_qrng\_free (*C function*), 231  
 gsl\_qrng\_get (*C function*), 231  
 gsl\_qrng\_init (*C function*), 231  
 gsl\_qrng\_memcpy (*C function*), 232  
 gsl\_qrng\_name (*C function*), 232  
 gsl\_qrng\_size (*C function*), 232  
 gsl\_qrng\_state (*C function*), 232  
 gsl\_qrng\_type (*C type*), 232  
 gsl\_qrng\_type.gsl\_qrng\_halton (*C var*), 232  
 gsl\_qrng\_type.gsl\_qrng\_niederreiter\_2 (*C var*),  
   232  
 gsl\_qrng\_type.gsl\_qrng\_reversehalton (*C var*),  
   232  
 gsl\_qrng\_type.gsl\_qrng\_sobol (*C var*), 232  
 gsl\_ran\_bernoulli (*C function*), 272  
 gsl\_ran\_bernoulli\_pdf (*C function*), 272  
 gsl\_ran\_beta (*C function*), 262  
 gsl\_ran\_beta\_pdf (*C function*), 262  
 gsl\_ran\_binomial (*C function*), 273  
 gsl\_ran\_binomial\_pdf (*C function*), 273  
 gsl\_ran\_bivariate\_gaussian (*C function*), 241  
 gsl\_ran\_bivariate\_gaussian\_pdf (*C function*), 241  
 gsl\_ran\_cauchy (*C function*), 246  
 gsl\_ran\_cauchy\_pdf (*C function*), 246  
 gsl\_ran\_chisq (*C function*), 256  
 gsl\_ran\_chisq\_pdf (*C function*), 256  
 gsl\_ran\_choose (*C function*), 281  
 gsl\_ran\_dir\_2d (*C function*), 265  
 gsl\_ran\_dir\_2d\_trig\_method (*C function*), 265  
 gsl\_ran\_dir\_3d (*C function*), 265  
 gsl\_ran\_dir\_nd (*C function*), 265  
 gsl\_ran\_dirichlet (*C function*), 269  
 gsl\_ran\_dirichlet\_lnpdf (*C function*), 269  
 gsl\_ran\_dirichlet\_pdf (*C function*), 269  
 gsl\_ran\_discrete (*C function*), 270  
 gsl\_ran\_discrete\_free (*C function*), 270  
 gsl\_ran\_discrete\_pdf (*C function*), 270

- gsl\_ran\_discrete\_preproc (*C function*), 270  
 gsl\_ran\_discrete\_t (*C type*), 270  
 gsl\_ran\_exponential (*C function*), 243  
 gsl\_ran\_exponential\_pdf (*C function*), 243  
 gsl\_ran\_exppow (*C function*), 245  
 gsl\_ran\_exppow\_pdf (*C function*), 245  
 gsl\_ran\_fdist (*C function*), 258  
 gsl\_ran\_fdist\_pdf (*C function*), 258  
 gsl\_ran\_flat (*C function*), 254  
 gsl\_ran\_flat\_pdf (*C function*), 254  
 gsl\_ran\_gamma (*C function*), 252  
 gsl\_ran\_gamma\_knuth (*C function*), 252  
 gsl\_ran\_gamma\_pdf (*C function*), 252  
 gsl\_ran\_gaussian (*C function*), 237  
 gsl\_ran\_gaussian\_pdf (*C function*), 237  
 gsl\_ran\_gaussian\_ratio\_method (*C function*), 237  
 gsl\_ran\_gaussian\_tail (*C function*), 239  
 gsl\_ran\_gaussian\_tail\_pdf (*C function*), 239  
 gsl\_ran\_gaussian\_ziggurat (*C function*), 237  
 gsl\_ran\_geometric (*C function*), 277  
 gsl\_ran\_geometric\_pdf (*C function*), 277  
 gsl\_ran\_gumbel1 (*C function*), 267  
 gsl\_ran\_gumbel1\_pdf (*C function*), 267  
 gsl\_ran\_gumbel2 (*C function*), 268  
 gsl\_ran\_gumbel2\_pdf (*C function*), 268  
 gsl\_ran\_hypergeometric (*C function*), 278  
 gsl\_ran\_hypergeometric\_pdf (*C function*), 278  
 gsl\_ran\_landau (*C function*), 249  
 gsl\_ran\_landau\_pdf (*C function*), 249  
 gsl\_ran\_laplace (*C function*), 244  
 gsl\_ran\_laplace\_pdf (*C function*), 244  
 gsl\_ran\_levy (*C function*), 250  
 gsl\_ran\_levy\_skew (*C function*), 251  
 gsl\_ran\_logarithmic (*C function*), 279  
 gsl\_ran\_logarithmic\_pdf (*C function*), 279  
 gsl\_ran\_logistic (*C function*), 263  
 gsl\_ran\_logistic\_pdf (*C function*), 263  
 gsl\_ran\_lognormal (*C function*), 255  
 gsl\_ran\_lognormal\_pdf (*C function*), 255  
 gsl\_ran\_multinomial (*C function*), 274  
 gsl\_ran\_multinomial\_lnpdf (*C function*), 274  
 gsl\_ran\_multinomial\_pdf (*C function*), 274  
 gsl\_ran\_multivariate\_gaussian (*C function*), 242  
 gsl\_ran\_multivariate\_gaussian\_log\_pdf (*C function*), 242  
 gsl\_ran\_multivariate\_gaussian\_mean (*C function*), 242  
 gsl\_ran\_multivariate\_gaussian\_pdf (*C function*), 242  
 gsl\_ran\_multivariate\_gaussian\_vcov (*C function*), 242  
 gsl\_ran\_negative\_binomial (*C function*), 275  
 gsl\_ran\_negative\_binomial\_pdf (*C function*), 275  
 gsl\_ran\_pareto (*C function*), 264  
 gsl\_ran\_pareto\_pdf (*C function*), 264  
 gsl\_ran\_pascal (*C function*), 276  
 gsl\_ran\_pascal\_pdf (*C function*), 276  
 gsl\_ran\_poisson (*C function*), 271  
 gsl\_ran\_poisson\_pdf (*C function*), 271  
 gsl\_ran\_rayleigh (*C function*), 247  
 gsl\_ran\_rayleigh\_pdf (*C function*), 247  
 gsl\_ran\_rayleigh\_tail (*C function*), 248  
 gsl\_ran\_rayleigh\_tail\_pdf (*C function*), 248  
 gsl\_ran\_sample (*C function*), 281  
 gsl\_ran\_shuffle (*C function*), 281  
 gsl\_ran\_tdist (*C function*), 260  
 gsl\_ran\_tdist\_pdf (*C function*), 260  
 gsl\_ran\_ugaussian (*C function*), 237  
 gsl\_ran\_ugaussian\_pdf (*C function*), 237  
 gsl\_ran\_ugaussian\_ratio\_method (*C function*), 237  
 gsl\_ran\_ugaussian\_tail (*C function*), 239  
 gsl\_ran\_ugaussian\_tail\_pdf (*C function*), 239  
 gsl\_ran\_weibull (*C function*), 266  
 gsl\_ran\_weibull\_pdf (*C function*), 266  
 gsl\_ran\_wishart (*C function*), 280  
 gsl\_ran\_wishart\_log\_pdf (*C function*), 280  
 gsl\_ran\_wishart\_pdf (*C function*), 280  
 GSL\_RANGE\_CHECK\_OFF (*C macro*), 86  
 GSL\_REAL (*C macro*), 24  
 gsl\_rng (*C type*), 215  
 gsl\_rng\_alloc (*C function*), 216  
 gsl\_rng\_borosh13 (*C var*), 226  
 gsl\_rng\_clone (*C function*), 219  
 gsl\_rng\_cmrg (*C var*), 221  
 gsl\_rng\_coveyou (*C var*), 226  
 gsl\_rng\_default (*C var*), 218  
 gsl\_rng\_default\_seed (*C var*), 218  
 gsl\_rng\_env\_setup (*C function*), 218  
 gsl\_rng\_fishman18 (*C var*), 226  
 gsl\_rng\_fishman20 (*C var*), 226  
 gsl\_rng\_fishman2x (*C var*), 226  
 gsl\_rng\_fread (*C function*), 220  
 gsl\_rng\_free (*C function*), 216  
 gsl\_rng\_fwrite (*C function*), 220  
 gsl\_rng\_get (*C function*), 216  
 gsl\_rng\_gfsr4 (*C var*), 222  
 gsl\_rng\_knuthran (*C var*), 226  
 gsl\_rng\_knuthran2 (*C var*), 226  
 gsl\_rng\_knuthran2002 (*C var*), 226  
 gsl\_rng\_lecuyer21 (*C var*), 226  
 gsl\_rng\_max (*C function*), 217  
 gsl\_rng\_memcpy (*C function*), 219  
 gsl\_rng\_min (*C function*), 217  
 gsl\_rng\_minstd (*C var*), 225  
 gsl\_rng\_mrg (*C var*), 221  
 gsl\_rng\_mt19937 (*C var*), 220  
 gsl\_rng\_name (*C function*), 217  
 gsl\_rng\_r250 (*C var*), 224

- gsl\_rng\_rand (C var), 223  
 gsl\_rng\_rand48 (C var), 223  
 gsl\_rng\_random\_bsd (C var), 223  
 gsl\_rng\_random\_glibc2 (C var), 223  
 gsl\_rng\_random\_libc5 (C var), 223  
 gsl\_rng\_randu (C var), 225  
 gsl\_rng\_ranf (C var), 224  
 gsl\_rng\_ranlux (C var), 221  
 gsl\_rng\_ranlux389 (C var), 221  
 gsl\_rng\_ranlxd1 (C var), 221  
 gsl\_rng\_ranlxd2 (C var), 221  
 gsl\_rng\_ranlxs0 (C var), 220  
 gsl\_rng\_ranlxs1 (C var), 220  
 gsl\_rng\_ranlxs2 (C var), 220  
 gsl\_rng\_ranmar (C var), 224  
 GSL\_RNG\_SEED (C macro), 218  
 gsl\_rng\_set (C function), 216  
 gsl\_rng\_size (C function), 217  
 gsl\_rng\_slatec (C var), 225  
 gsl\_rng\_state (C function), 217  
 gsl\_rng\_taus (C var), 222  
 gsl\_rng\_taus2 (C var), 222  
 gsl\_rng\_transputer (C var), 225  
 gsl\_rng\_tt800 (C var), 224  
 GSL\_RNG\_TYPE (C macro), 218  
 gsl\_rng\_type (C type), 215  
 gsl\_rng\_types\_setup (C function), 217  
 gsl\_rng\_uni (C var), 225  
 gsl\_rng\_uni32 (C var), 225  
 gsl\_rng\_uniform (C function), 216  
 gsl\_rng\_uniform\_int (C function), 217  
 gsl\_rng\_uniform\_pos (C function), 217  
 gsl\_rng\_vax (C var), 225  
 gsl\_rng\_waterman14 (C var), 226  
 gsl\_rng\_zuf (C var), 225  
 gsl\_root\_condsolver (C type), 434  
 gsl\_root\_condsolver\_alloc (C function), 434  
 gsl\_root\_condsolver\_free (C function), 434  
 gsl\_root\_condsolver\_iterate (C function), 437  
 gsl\_root\_condsolver\_name (C function), 435  
 gsl\_root\_condsolver\_root (C function), 437  
 gsl\_root\_condsolver\_set (C function), 434  
 gsl\_root\_condsolver\_type (C type), 439  
 gsl\_root\_condsolver\_type.gsl\_root\_condsolver\_newton  
 (C var), 439  
 gsl\_root\_condsolver\_type.gsl\_root\_condsolver\_secant  
 (C var), 439  
 gsl\_root\_condsolver\_type.gsl\_root\_condsolver\_stepson  
 (C var), 440  
 gsl\_root\_fsolver (C type), 434  
 gsl\_root\_fsolver\_alloc (C function), 434  
 gsl\_root\_fsolver\_free (C function), 434  
 gsl\_root\_fsolver\_iterate (C function), 437  
 gsl\_root\_fsolver\_name (C function), 435  
 gsl\_root\_fsolver\_root (C function), 437  
 gsl\_root\_fsolver\_set (C function), 434  
 gsl\_root\_fsolver\_type (C type), 438  
 gsl\_root\_fsolver\_type.gsl\_root\_fsolver\_bisection  
 (C var), 438  
 gsl\_root\_fsolver\_type.gsl\_root\_fsolver\_brent  
 (C var), 439  
 gsl\_root\_fsolver\_type.gsl\_root\_fsolver\_falsepos  
 (C var), 439  
 gsl\_root\_fsolver\_x\_lower (C function), 437  
 gsl\_root\_fsolver\_x\_upper (C function), 437  
 gsl\_root\_test\_delta (C function), 438  
 gsl\_root\_test\_interval (C function), 438  
 gsl\_root\_test\_residual (C function), 438  
 gsl\_rstat\_add (C function), 299  
 gsl\_rstat\_alloc (C function), 299  
 gsl\_rstat\_free (C function), 299  
 gsl\_rstat\_kurtosis (C function), 300  
 gsl\_rstat\_max (C function), 300  
 gsl\_rstat\_mean (C function), 300  
 gsl\_rstat\_median (C function), 300  
 gsl\_rstat\_min (C function), 300  
 gsl\_rstat\_n (C function), 299  
 gsl\_rstat\_quantile\_add (C function), 301  
 gsl\_rstat\_quantile\_alloc (C function), 301  
 gsl\_rstat\_quantile\_free (C function), 301  
 gsl\_rstat\_quantile\_get (C function), 301  
 gsl\_rstat\_quantile\_reset (C function), 301  
 gsl\_rstat\_quantile\_workspace (C type), 301  
 gsl\_rstat\_reset (C function), 299  
 gsl\_rstat\_rms (C function), 300  
 gsl\_rstat\_sd (C function), 300  
 gsl\_rstat\_sd\_mean (C function), 300  
 gsl\_rstat\_skew (C function), 300  
 gsl\_rstat\_variance (C function), 300  
 gsl\_rstat\_workspace (C type), 299  
 GSL\_SET\_COMPLEX (C macro), 24  
 gsl\_set\_error\_handler (C function), 15  
 gsl\_set\_error\_handler\_off (C function), 15  
 gsl\_sfairy\_Ai (C function), 37  
 gsl\_sfairy\_Ai\_deriv (C function), 37  
 gsl\_sfairy\_Ai\_deriv\_e (C function), 37  
 gsl\_sfairy\_Ai\_deriv\_scaled (C function), 37  
 gsl\_sfairy\_Ai\_deriv\_scaled\_e (C function), 37  
 gsl\_sfairy\_Ai\_e (C function), 37  
 gsl\_sfairy\_Ai\_scaled (C function), 37  
 gsl\_sfairy\_Ai\_scaled\_e (C function), 37  
 gsl\_sfairy\_Bi (C function), 37  
 gsl\_sfairy\_Bi\_deriv (C function), 37  
 gsl\_sfairy\_Bi\_deriv\_e (C function), 37  
 gsl\_sfairy\_Bi\_deriv\_scaled (C function), 38  
 gsl\_sfairy\_Bi\_deriv\_scaled\_e (C function), 38  
 gsl\_sfairy\_Bi\_e (C function), 37  
 gsl\_sfairy\_Bi\_scaled (C function), 37

`gsl_sf_airy_Bi_scaled_e` (C function), 37  
`gsl_sf_airy_zero_Ai` (C function), 38  
`gsl_sf_airy_zero_Ai_deriv` (C function), 38  
`gsl_sf_airy_zero_Ai_deriv_e` (C function), 38  
`gsl_sf_airy_zero_Ai_e` (C function), 38  
`gsl_sf_airy_zero_Bi` (C function), 38  
`gsl_sf_airy_zero_Bi_deriv` (C function), 38  
`gsl_sf_airy_zero_Bi_deriv_e` (C function), 38  
`gsl_sf_airy_zero_Bi_e` (C function), 38  
`gsl_sf_angle_restrict_pos` (C function), 79  
`gsl_sf_angle_restrict_pos_e` (C function), 79  
`gsl_sf_angle_restrict_symm` (C function), 79  
`gsl_sf_angle_restrict_symm_e` (C function), 79  
`gsl_sf_atanint` (C function), 56  
`gsl_sf_atanint_e` (C function), 56  
`gsl_sf_bessel_I0` (C function), 39  
`gsl_sf_bessel_I0_e` (C function), 39  
`gsl_sf_bessel_I0_scaled` (C function), 39  
`gsl_sf_bessel_i0_scaled` (C function), 42  
`gsl_sf_bessel_I0_scaled_e` (C function), 39  
`gsl_sf_bessel_i0_scaled_e` (C function), 42  
`gsl_sf_bessel_I1` (C function), 39  
`gsl_sf_bessel_I1_e` (C function), 39  
`gsl_sf_bessel_I1_scaled` (C function), 40  
`gsl_sf_bessel_i1_scaled` (C function), 42  
`gsl_sf_bessel_I1_scaled_e` (C function), 40  
`gsl_sf_bessel_i1_scaled_e` (C function), 42  
`gsl_sf_bessel_i2_scaled` (C function), 42  
`gsl_sf_bessel_i2_scaled_e` (C function), 42  
`gsl_sf_bessel_il_scaled` (C function), 42  
`gsl_sf_bessel_il_scaled_array` (C function), 42  
`gsl_sf_bessel_il_scaled_e` (C function), 42  
`gsl_sf_bessel_In` (C function), 39  
`gsl_sf_bessel_In_array` (C function), 39  
`gsl_sf_bessel_In_e` (C function), 39  
`gsl_sf_bessel_In_scaled` (C function), 40  
`gsl_sf_bessel_In_scaled_array` (C function), 40  
`gsl_sf_bessel_In_scaled_e` (C function), 40  
`gsl_sf_bessel_Inu` (C function), 44  
`gsl_sf_bessel_Inu_e` (C function), 44  
`gsl_sf_bessel_Inu_scaled` (C function), 44  
`gsl_sf_bessel_Inu_scaled_e` (C function), 44  
`gsl_sf_bessel_J0` (C function), 38  
`gsl_sf_bessel_j0` (C function), 41  
`gsl_sf_bessel_J0_e` (C function), 38  
`gsl_sf_bessel_j0_e` (C function), 41  
`gsl_sf_bessel_J1` (C function), 38  
`gsl_sf_bessel_j1` (C function), 41  
`gsl_sf_bessel_J1_e` (C function), 38  
`gsl_sf_bessel_j1_e` (C function), 41  
`gsl_sf_bessel_j2` (C function), 41  
`gsl_sf_bessel_j2_e` (C function), 41  
`gsl_sf_bessel_jl` (C function), 41  
`gsl_sf_bessel_jl_array` (C function), 41  
`gsl_sf_bessel_jl_e` (C function), 41  
`gsl_sf_bessel_jl_stepped_array` (C function), 41  
`gsl_sf_bessel_Jn` (C function), 38  
`gsl_sf_bessel_Jn_array` (C function), 39  
`gsl_sf_bessel_Jn_e` (C function), 38  
`gsl_sf_bessel_Jnu` (C function), 43  
`gsl_sf_bessel_Jnu_e` (C function), 43  
`gsl_sf_bessel_K0` (C function), 40  
`gsl_sf_bessel_K0_e` (C function), 40  
`gsl_sf_bessel_K0_scaled` (C function), 40  
`gsl_sf_bessel_k0_scaled` (C function), 43  
`gsl_sf_bessel_K0_scaled_e` (C function), 40  
`gsl_sf_bessel_k0_scaled_e` (C function), 43  
`gsl_sf_bessel_K1` (C function), 40  
`gsl_sf_bessel_K1_e` (C function), 40  
`gsl_sf_bessel_K1_scaled` (C function), 40  
`gsl_sf_bessel_k1_scaled` (C function), 43  
`gsl_sf_bessel_K1_scaled_e` (C function), 40  
`gsl_sf_bessel_k1_scaled_e` (C function), 43  
`gsl_sf_bessel_k2_scaled` (C function), 43  
`gsl_sf_bessel_k2_scaled_e` (C function), 43  
`gsl_sf_bessel_kl_scaled` (C function), 43  
`gsl_sf_bessel_kl_scaled_array` (C function), 43  
`gsl_sf_bessel_kl_scaled_e` (C function), 43  
`gsl_sf_bessel_Kn` (C function), 40  
`gsl_sf_bessel_Kn_array` (C function), 40  
`gsl_sf_bessel_Kn_e` (C function), 40  
`gsl_sf_bessel_Kn_scaled` (C function), 40  
`gsl_sf_bessel_Kn_scaled_array` (C function), 41  
`gsl_sf_bessel_Kn_scaled_e` (C function), 40  
`gsl_sf_bessel_Knu` (C function), 44  
`gsl_sf_bessel_Knu_e` (C function), 44  
`gsl_sf_bessel_Knu_scaled` (C function), 44  
`gsl_sf_bessel_Knu_scaled_e` (C function), 44  
`gsl_sf_bessel_lnKnu` (C function), 44  
`gsl_sf_bessel_lnKnu_e` (C function), 44  
`gsl_sf_bessel_sequence_Jnu_e` (C function), 43  
`gsl_sf_bessel_Y0` (C function), 39  
`gsl_sf_bessel_y0` (C function), 42  
`gsl_sf_bessel_Y0_e` (C function), 39  
`gsl_sf_bessel_y0_e` (C function), 42  
`gsl_sf_bessel_Y1` (C function), 39  
`gsl_sf_bessel_y1` (C function), 42  
`gsl_sf_bessel_Y1_e` (C function), 39  
`gsl_sf_bessel_y1_e` (C function), 42  
`gsl_sf_bessel_y2` (C function), 42  
`gsl_sf_bessel_y2_e` (C function), 42  
`gsl_sf_bessel_yl` (C function), 42  
`gsl_sf_bessel_yl_array` (C function), 42  
`gsl_sf_bessel_yl_e` (C function), 42  
`gsl_sf_bessel_Yn` (C function), 39  
`gsl_sf_bessel_Yn_array` (C function), 39  
`gsl_sf_bessel_Yn_e` (C function), 39  
`gsl_sf_bessel_Ynu` (C function), 44



gsl\_sf\_bessel\_Ynu\_e (C function), 44  
 gsl\_sf\_bessel\_zero\_J0 (C function), 44  
 gsl\_sf\_bessel\_zero\_J0\_e (C function), 44  
 gsl\_sf\_bessel\_zero\_J1 (C function), 44  
 gsl\_sf\_bessel\_zero\_J1\_e (C function), 44  
 gsl\_sf\_bessel\_zero\_Jnu (C function), 44  
 gsl\_sf\_bessel\_zero\_Jnu\_e (C function), 44  
 gsl\_sf\_beta (C function), 61  
 gsl\_sf\_beta\_e (C function), 61  
 gsl\_sf\_beta\_inc (C function), 61  
 gsl\_sf\_beta\_inc\_e (C function), 61  
 gsl\_sf\_Chi (C function), 56  
 gsl\_sf\_Chi\_e (C function), 56  
 gsl\_sf\_choose (C function), 59  
 gsl\_sf\_choose\_e (C function), 59  
 gsl\_sf\_Ci (C function), 56  
 gsl\_sf\_Ci\_e (C function), 56  
 gsl\_sf\_clausen (C function), 45  
 gsl\_sf\_clausen\_e (C function), 45  
 gsl\_sf\_complex\_cos\_e (C function), 78  
 gsl\_sf\_complex\_dilog\_e (C function), 49  
 gsl\_sf\_complex\_log\_e (C function), 73  
 gsl\_sf\_complex\_logsin\_e (C function), 78  
 gsl\_sf\_complex\_sin\_e (C function), 78  
 gsl\_sf\_conicalP\_0 (C function), 72  
 gsl\_sf\_conicalP\_0\_e (C function), 72  
 gsl\_sf\_conicalP\_1 (C function), 72  
 gsl\_sf\_conicalP\_1\_e (C function), 72  
 gsl\_sf\_conicalP\_cyl\_reg (C function), 72  
 gsl\_sf\_conicalP\_cyl\_reg\_e (C function), 72  
 gsl\_sf\_conicalP\_half (C function), 72  
 gsl\_sf\_conicalP\_half\_e (C function), 72  
 gsl\_sf\_conicalP\_mhalf (C function), 72  
 gsl\_sf\_conicalP\_mhalf\_e (C function), 72  
 gsl\_sf\_conicalP\_sph\_reg (C function), 72  
 gsl\_sf\_conicalP\_sph\_reg\_e (C function), 72  
 gsl\_sf\_cos (C function), 78  
 gsl\_sf\_cos\_e (C function), 78  
 gsl\_sf\_cos\_err\_e (C function), 79  
 gsl\_sf\_coulomb\_CL\_array (C function), 46  
 gsl\_sf\_coulomb\_CL\_e (C function), 46  
 gsl\_sf\_coulomb\_wave\_F\_array (C function), 46  
 gsl\_sf\_coulomb\_wave\_FG\_array (C function), 46  
 gsl\_sf\_coulomb\_wave\_FG\_e (C function), 46  
 gsl\_sf\_coulomb\_wave\_FGp\_array (C function), 46  
 gsl\_sf\_coulomb\_wave\_sphF\_array (C function), 46  
 gsl\_sf\_coupling\_3j (C function), 47  
 gsl\_sf\_coupling\_3j\_e (C function), 47  
 gsl\_sf\_coupling\_6j (C function), 47  
 gsl\_sf\_coupling\_6j\_e (C function), 47  
 gsl\_sf\_coupling\_9j (C function), 47  
 gsl\_sf\_coupling\_9j\_e (C function), 47  
 gsl\_sf\_dawson (C function), 48  
 gsl\_sf\_dawson\_e (C function), 48  
 gsl\_sf\_debye\_1 (C function), 48  
 gsl\_sf\_debye\_1\_e (C function), 48  
 gsl\_sf\_debye\_2 (C function), 48  
 gsl\_sf\_debye\_2\_e (C function), 48  
 gsl\_sf\_debye\_3 (C function), 48  
 gsl\_sf\_debye\_3\_e (C function), 48  
 gsl\_sf\_debye\_4 (C function), 48  
 gsl\_sf\_debye\_4\_e (C function), 48  
 gsl\_sf\_debye\_5 (C function), 48  
 gsl\_sf\_debye\_5\_e (C function), 48  
 gsl\_sf\_debye\_6 (C function), 48  
 gsl\_sf\_debye\_6\_e (C function), 48  
 gsl\_sf\_dilog (C function), 49  
 gsl\_sf\_dilog\_e (C function), 49  
 gsl\_sf\_doublefact (C function), 59  
 gsl\_sf\_doublefact\_e (C function), 59  
 gsl\_sf\_ellint\_D (C function), 51  
 gsl\_sf\_ellint\_D\_e (C function), 51  
 gsl\_sf\_ellint\_E (C function), 51  
 gsl\_sf\_ellint\_E\_e (C function), 51  
 gsl\_sf\_ellint\_Ecomp (C function), 50  
 gsl\_sf\_ellint\_Ecomp\_e (C function), 50  
 gsl\_sf\_ellint\_F (C function), 51  
 gsl\_sf\_ellint\_F\_e (C function), 51  
 gsl\_sf\_ellint\_Kcomp (C function), 50  
 gsl\_sf\_ellint\_Kcomp\_e (C function), 50  
 gsl\_sf\_ellint\_P (C function), 51  
 gsl\_sf\_ellint\_P\_e (C function), 51  
 gsl\_sf\_ellint\_Pcomp (C function), 50  
 gsl\_sf\_ellint\_Pcomp\_e (C function), 50  
 gsl\_sf\_ellint\_RC (C function), 51  
 gsl\_sf\_ellint\_RC\_e (C function), 51  
 gsl\_sf\_ellint\_RD (C function), 51  
 gsl\_sf\_ellint\_RD\_e (C function), 51  
 gsl\_sf\_ellint\_RF (C function), 51  
 gsl\_sf\_ellint\_RF\_e (C function), 51  
 gsl\_sf\_ellint\_RJ (C function), 52  
 gsl\_sf\_ellint\_RJ\_e (C function), 52  
 gsl\_sf\_elljac\_e (C function), 52  
 gsl\_sf\_erf (C function), 52  
 gsl\_sf\_erf\_e (C function), 52  
 gsl\_sf\_erf\_Q (C function), 53  
 gsl\_sf\_erf\_Q\_e (C function), 53  
 gsl\_sf\_erf\_Z (C function), 53  
 gsl\_sf\_erf\_Z\_e (C function), 53  
 gsl\_sf\_erfc (C function), 52  
 gsl\_sf\_erfc\_e (C function), 52  
 gsl\_sf\_eta (C function), 81  
 gsl\_sf\_eta\_e (C function), 81  
 gsl\_sf\_eta\_int (C function), 81  
 gsl\_sf\_eta\_int\_e (C function), 81  
 gsl\_sf\_exp (C function), 53  
 gsl\_sf\_exp\_e (C function), 53  
 gsl\_sf\_exp\_e10\_e (C function), 53

- gsl\_sf\_exp\_err\_e (C function), 54  
 gsl\_sf\_exp\_err\_e10\_e (C function), 54  
 gsl\_sf\_exp\_mult (C function), 53  
 gsl\_sf\_exp\_mult\_e (C function), 53  
 gsl\_sf\_exp\_mult\_e10\_e (C function), 53  
 gsl\_sf\_exp\_mult\_err\_e (C function), 54  
 gsl\_sf\_exp\_mult\_err\_e10\_e (C function), 54  
 gsl\_sf\_expint\_3 (C function), 56  
 gsl\_sf\_expint\_3\_e (C function), 56  
 gsl\_sf\_expint\_E1 (C function), 55  
 gsl\_sf\_expint\_E1\_e (C function), 55  
 gsl\_sf\_expint\_E2 (C function), 55  
 gsl\_sf\_expint\_E2\_e (C function), 55  
 gsl\_sf\_expint\_Ei (C function), 55  
 gsl\_sf\_expint\_Ei\_e (C function), 55  
 gsl\_sf\_expint\_En (C function), 55  
 gsl\_sf\_expint\_En\_e (C function), 55  
 gsl\_sf\_expm1 (C function), 54  
 gsl\_sf\_expm1\_e (C function), 54  
 gsl\_sf\_exprel (C function), 54  
 gsl\_sf\_exprel\_2 (C function), 54  
 gsl\_sf\_exprel\_2\_e (C function), 54  
 gsl\_sf\_exprel\_e (C function), 54  
 gsl\_sf\_exprel\_n (C function), 54  
 gsl\_sf\_exprel\_n\_e (C function), 54  
 gsl\_sf\_fact (C function), 59  
 gsl\_sf\_fact\_e (C function), 59  
 gsl\_sf\_fermi\_dirac\_0 (C function), 57  
 gsl\_sf\_fermi\_dirac\_0\_e (C function), 57  
 gsl\_sf\_fermi\_dirac\_1 (C function), 57  
 gsl\_sf\_fermi\_dirac\_1\_e (C function), 57  
 gsl\_sf\_fermi\_dirac\_2 (C function), 57  
 gsl\_sf\_fermi\_dirac\_2\_e (C function), 57  
 gsl\_sf\_fermi\_dirac\_3half (C function), 57  
 gsl\_sf\_fermi\_dirac\_3half\_e (C function), 57  
 gsl\_sf\_fermi\_dirac\_half (C function), 57  
 gsl\_sf\_fermi\_dirac\_half\_e (C function), 57  
 gsl\_sf\_fermi\_dirac\_inc\_0 (C function), 58  
 gsl\_sf\_fermi\_dirac\_inc\_0\_e (C function), 58  
 gsl\_sf\_fermi\_dirac\_int (C function), 57  
 gsl\_sf\_fermi\_dirac\_int\_e (C function), 57  
 gsl\_sf\_fermi\_dirac\_m1 (C function), 57  
 gsl\_sf\_fermi\_dirac\_m1\_e (C function), 57  
 gsl\_sf\_fermi\_dirac\_mhalf (C function), 57  
 gsl\_sf\_fermi\_dirac\_mhalf\_e (C function), 57  
 gsl\_sf\_gamma (C function), 58  
 gsl\_sf\_gamma\_e (C function), 58  
 gsl\_sf\_gamma\_inc (C function), 60  
 gsl\_sf\_gamma\_inc\_e (C function), 60  
 gsl\_sf\_gamma\_inc\_P (C function), 60  
 gsl\_sf\_gamma\_inc\_P\_e (C function), 60  
 gsl\_sf\_gamma\_inc\_Q (C function), 60  
 gsl\_sf\_gamma\_inc\_Q\_e (C function), 60  
 gsl\_sf\_gammainv (C function), 59  
 gsl\_sf\_gammainv\_e (C function), 59  
 gsl\_sf\_gammastar (C function), 58  
 gsl\_sf\_gammastar\_e (C function), 58  
 gsl\_sf\_gegenpoly\_1 (C function), 61  
 gsl\_sf\_gegenpoly\_1\_e (C function), 61  
 gsl\_sf\_gegenpoly\_2 (C function), 61  
 gsl\_sf\_gegenpoly\_2\_e (C function), 61  
 gsl\_sf\_gegenpoly\_3 (C function), 61  
 gsl\_sf\_gegenpoly\_3\_e (C function), 61  
 gsl\_sf\_gegenpoly\_array (C function), 62  
 gsl\_sf\_gegenpoly\_n (C function), 62  
 gsl\_sf\_gegenpoly\_n\_e (C function), 62  
 gsl\_sf\_hazard (C function), 53  
 gsl\_sf\_hazard\_e (C function), 53  
 gsl\_sf\_hermite (C function), 62  
 gsl\_sf\_hermite\_array (C function), 62  
 gsl\_sf\_hermite\_array\_deriv (C function), 63  
 gsl\_sf\_hermite\_deriv (C function), 63  
 gsl\_sf\_hermite\_deriv\_array (C function), 63  
 gsl\_sf\_hermite\_deriv\_e (C function), 63  
 gsl\_sf\_hermite\_e (C function), 62  
 gsl\_sf\_hermite\_func (C function), 64  
 gsl\_sf\_hermite\_func\_array (C function), 64  
 gsl\_sf\_hermite\_func\_der (C function), 64  
 gsl\_sf\_hermite\_func\_der\_e (C function), 64  
 gsl\_sf\_hermite\_func\_e (C function), 64  
 gsl\_sf\_hermite\_func\_fast (C function), 64  
 gsl\_sf\_hermite\_func\_fast\_e (C function), 64  
 gsl\_sf\_hermite\_func\_series (C function), 64  
 gsl\_sf\_hermite\_func\_series\_e (C function), 64  
 gsl\_sf\_hermite\_func\_zero (C function), 65  
 gsl\_sf\_hermite\_func\_zero\_e (C function), 65  
 gsl\_sf\_hermite\_prob (C function), 63  
 gsl\_sf\_hermite\_prob\_array (C function), 63  
 gsl\_sf\_hermite\_prob\_array\_deriv (C function), 63  
 gsl\_sf\_hermite\_prob\_deriv (C function), 63  
 gsl\_sf\_hermite\_prob\_deriv\_array (C function), 63  
 gsl\_sf\_hermite\_prob\_deriv\_e (C function), 63  
 gsl\_sf\_hermite\_prob\_e (C function), 63  
 gsl\_sf\_hermite\_prob\_series (C function), 63  
 gsl\_sf\_hermite\_prob\_series\_e (C function), 63  
 gsl\_sf\_hermite\_prob\_zero (C function), 65  
 gsl\_sf\_hermite\_prob\_zero\_e (C function), 65  
 gsl\_sf\_hermite\_series (C function), 62  
 gsl\_sf\_hermite\_series\_e (C function), 62  
 gsl\_sf\_hermite\_zero (C function), 65  
 gsl\_sf\_hermite\_zero\_e (C function), 65  
 gsl\_sf\_hydrogenicR (C function), 45  
 gsl\_sf\_hydrogenicR\_1 (C function), 45  
 gsl\_sf\_hydrogenicR\_1\_e (C function), 45  
 gsl\_sf\_hydrogenicR\_e (C function), 45  
 gsl\_sf\_hyperg\_0F1 (C function), 65  
 gsl\_sf\_hyperg\_0F1\_e (C function), 65  
 gsl\_sf\_hyperg\_1F1 (C function), 65

- gsl\_sf\_hyperg\_1F1\_e (C function), 65  
 gsl\_sf\_hyperg\_1F1\_int (C function), 65  
 gsl\_sf\_hyperg\_1F1\_int\_e (C function), 65  
 gsl\_sf\_hyperg\_2F0 (C function), 66  
 gsl\_sf\_hyperg\_2F0\_e (C function), 66  
 gsl\_sf\_hyperg\_2F1 (C function), 66  
 gsl\_sf\_hyperg\_2F1\_conj (C function), 66  
 gsl\_sf\_hyperg\_2F1\_conj\_e (C function), 66  
 gsl\_sf\_hyperg\_2F1\_conj\_renorm (C function), 66  
 gsl\_sf\_hyperg\_2F1\_conj\_renorm\_e (C function), 66  
 gsl\_sf\_hyperg\_2F1\_e (C function), 66  
 gsl\_sf\_hyperg\_2F1\_renorm (C function), 66  
 gsl\_sf\_hyperg\_2F1\_renorm\_e (C function), 66  
 gsl\_sf\_hyperg\_U (C function), 66  
 gsl\_sf\_hyperg\_U\_e (C function), 66  
 gsl\_sf\_hyperg\_U\_e10\_e (C function), 66  
 gsl\_sf\_hyperg\_U\_int (C function), 65  
 gsl\_sf\_hyperg\_U\_int\_e (C function), 65  
 gsl\_sf\_hyperg\_U\_int\_e10\_e (C function), 66  
 gsl\_sf\_hypot (C function), 78  
 gsl\_sf\_hypot\_e (C function), 78  
 gsl\_sf\_hzeta (C function), 80  
 gsl\_sf\_hzeta\_e (C function), 80  
 gsl\_sf\_laguerre\_1 (C function), 67  
 gsl\_sf\_laguerre\_1\_e (C function), 67  
 gsl\_sf\_laguerre\_2 (C function), 67  
 gsl\_sf\_laguerre\_2\_e (C function), 67  
 gsl\_sf\_laguerre\_3 (C function), 67  
 gsl\_sf\_laguerre\_3\_e (C function), 67  
 gsl\_sf\_laguerre\_n (C function), 67  
 gsl\_sf\_laguerre\_n\_e (C function), 67  
 gsl\_sf\_lambert\_W0 (C function), 67  
 gsl\_sf\_lambert\_W0\_e (C function), 67  
 gsl\_sf\_lambert\_Wm1 (C function), 68  
 gsl\_sf\_lambert\_Wm1\_e (C function), 68  
 gsl\_sf\_legendre\_array (C function), 70  
 gsl\_sf\_legendre\_array\_e (C function), 70  
 gsl\_sf\_legendre\_array\_index (C function), 71  
 gsl\_sf\_legendre\_array\_n (C function), 71  
 gsl\_sf\_legendre\_array\_size (C function), 72  
 gsl\_sf\_legendre\_deriv2\_alt\_array (C function), 71  
 gsl\_sf\_legendre\_deriv2\_alt\_array\_e (C function), 71  
 gsl\_sf\_legendre\_deriv2\_array (C function), 70  
 gsl\_sf\_legendre\_deriv2\_array\_e (C function), 70  
 gsl\_sf\_legendre\_deriv\_alt\_array (C function), 70  
 gsl\_sf\_legendre\_deriv\_alt\_array\_e (C function), 70  
 gsl\_sf\_legendre\_deriv\_array (C function), 70  
 gsl\_sf\_legendre\_deriv\_array\_e (C function), 70  
 gsl\_sf\_legendre\_H3d (C function), 73  
 gsl\_sf\_legendre\_H3d\_0 (C function), 72  
 gsl\_sf\_legendre\_H3d\_0\_e (C function), 72  
 gsl\_sf\_legendre\_H3d\_1 (C function), 73  
 gsl\_sf\_legendre\_H3d\_1\_e (C function), 73  
 gsl\_sf\_legendre\_H3d\_array (C function), 73  
 gsl\_sf\_legendre\_H3d\_e (C function), 73  
 gsl\_sf\_legendre\_nlm (C function), 71  
 gsl\_sf\_legendre\_P1 (C function), 68  
 gsl\_sf\_legendre\_P1\_e (C function), 68  
 gsl\_sf\_legendre\_P2 (C function), 68  
 gsl\_sf\_legendre\_P2\_e (C function), 68  
 gsl\_sf\_legendre\_P3 (C function), 68  
 gsl\_sf\_legendre\_P3\_e (C function), 68  
 gsl\_sf\_legendre\_Pl (C function), 68  
 gsl\_sf\_legendre\_Pl\_array (C function), 68  
 gsl\_sf\_legendre\_Pl\_deriv\_array (C function), 68  
 gsl\_sf\_legendre\_Pl\_e (C function), 68  
 gsl\_sf\_legendre\_Pl\_m (C function), 71  
 gsl\_sf\_legendre\_Pl\_m\_array (C function), 71  
 gsl\_sf\_legendre\_Pl\_m\_deriv\_array (C function), 71  
 gsl\_sf\_legendre\_Pl\_m\_e (C function), 71  
 gsl\_sf\_legendre\_Q0 (C function), 68  
 gsl\_sf\_legendre\_Q0\_e (C function), 68  
 gsl\_sf\_legendre\_Q1 (C function), 68  
 gsl\_sf\_legendre\_Q1\_e (C function), 68  
 gsl\_sf\_legendre\_Ql (C function), 68  
 gsl\_sf\_legendre\_Ql\_e (C function), 68  
 gsl\_sf\_legendre\_sphPl\_m (C function), 71  
 gsl\_sf\_legendre\_sphPl\_m\_array (C function), 71  
 gsl\_sf\_legendre\_sphPl\_m\_deriv\_array (C function), 71  
 gsl\_sf\_legendre\_sphPl\_m\_e (C function), 71  
 gsl\_sf\_legendre\_t (C type), 69  
 gsl\_sf\_lnbeta (C function), 61  
 gsl\_sf\_lnbeta\_e (C function), 61  
 gsl\_sf\_lnchoose (C function), 60  
 gsl\_sf\_lnchoose\_e (C function), 60  
 gsl\_sf\_lncosh (C function), 78  
 gsl\_sf\_lncosh\_e (C function), 78  
 gsl\_sf\_lndoublefact (C function), 59  
 gsl\_sf\_lndoublefact\_e (C function), 59  
 gsl\_sf\_lnfact (C function), 59  
 gsl\_sf\_lnfact\_e (C function), 59  
 gsl\_sf\_lngamma (C function), 58  
 gsl\_sf\_lngamma\_complex\_e (C function), 59  
 gsl\_sf\_lngamma\_e (C function), 58  
 gsl\_sf\_lngamma\_sgn\_e (C function), 58  
 gsl\_sf\_lnpoch (C function), 60  
 gsl\_sf\_lnpoch\_e (C function), 60  
 gsl\_sf\_lnpoch\_sgn\_e (C function), 60  
 gsl\_sf\_lnsinh (C function), 78  
 gsl\_sf\_lnsinh\_e (C function), 78  
 gsl\_sf\_log (C function), 73  
 gsl\_sf\_log\_1plusx (C function), 73  
 gsl\_sf\_log\_1plusx\_e (C function), 73  
 gsl\_sf\_log\_1plusx\_mx (C function), 73

- gsl\_sf\_log\_1plusx\_mx\_e (C function), 73  
 gsl\_sf\_log\_abs (C function), 73  
 gsl\_sf\_log\_abs\_e (C function), 73  
 gsl\_sf\_log\_e (C function), 73  
 gsl\_sf\_log\_erfc (C function), 52  
 gsl\_sf\_log\_erfc\_e (C function), 52  
 gsl\_sf\_mathieu\_a (C function), 74  
 gsl\_sf\_mathieu\_a\_array (C function), 74  
 gsl\_sf\_mathieu\_a\_e (C function), 74  
 gsl\_sf\_mathieu\_alloc (C function), 74  
 gsl\_sf\_mathieu\_b (C function), 74  
 gsl\_sf\_mathieu\_b\_array (C function), 74  
 gsl\_sf\_mathieu\_b\_e (C function), 74  
 gsl\_sf\_mathieu\_ce (C function), 75  
 gsl\_sf\_mathieu\_ce\_array (C function), 75  
 gsl\_sf\_mathieu\_ce\_e (C function), 75  
 gsl\_sf\_mathieu\_free (C function), 74  
 gsl\_sf\_mathieu\_Mc (C function), 75  
 gsl\_sf\_mathieu\_Mc\_array (C function), 75  
 gsl\_sf\_mathieu\_Mc\_e (C function), 75  
 gsl\_sf\_mathieu\_Ms (C function), 75  
 gsl\_sf\_mathieu\_Ms\_array (C function), 75  
 gsl\_sf\_mathieu\_Ms\_e (C function), 75  
 gsl\_sf\_mathieu\_se (C function), 75  
 gsl\_sf\_mathieu\_se\_array (C function), 75  
 gsl\_sf\_mathieu\_se\_e (C function), 75  
 gsl\_sf\_mathieu\_workspace (C type), 74  
 gsl\_sf\_multiply (C function), 49  
 gsl\_sf\_multiply\_e (C function), 49  
 gsl\_sf\_multiply\_err\_e (C function), 49  
 gsl\_sf\_poch (C function), 60  
 gsl\_sf\_poch\_e (C function), 60  
 gsl\_sf\_pochrel (C function), 60  
 gsl\_sf\_pochrel\_e (C function), 60  
 gsl\_sf\_polar\_to\_rect (C function), 79  
 gsl\_sf\_pow\_int (C function), 75  
 gsl\_sf\_pow\_int\_e (C function), 75  
 gsl\_sf\_psi (C function), 76  
 gsl\_sf\_psi\_1 (C function), 76  
 gsl\_sf\_psi\_1\_e (C function), 76  
 gsl\_sf\_psi\_1\_int (C function), 76  
 gsl\_sf\_psi\_1\_int\_e (C function), 76  
 gsl\_sf\_psi\_1piy (C function), 76  
 gsl\_sf\_psi\_1piy\_e (C function), 76  
 gsl\_sf\_psi\_e (C function), 76  
 gsl\_sf\_psi\_int (C function), 76  
 gsl\_sf\_psi\_int\_e (C function), 76  
 gsl\_sf\_psi\_n (C function), 77  
 gsl\_sf\_psi\_n\_e (C function), 77  
 gsl\_sf\_rect\_to\_polar (C function), 79  
 gsl\_sf\_result (C type), 36  
 gsl\_sf\_result\_e10 (C type), 36  
 gsl\_sf\_Shi (C function), 55  
 gsl\_sf\_Shi\_e (C function), 55  
 gsl\_sf\_Si (C function), 56  
 gsl\_sf\_Si\_e (C function), 56  
 gsl\_sf\_sin (C function), 78  
 gsl\_sf\_sin\_e (C function), 78  
 gsl\_sf\_sin\_err\_e (C function), 79  
 gsl\_sf\_sinc (C function), 78  
 gsl\_sf\_sinc\_e (C function), 78  
 gsl\_sf\_synchrotron\_1 (C function), 77  
 gsl\_sf\_synchrotron\_1\_e (C function), 77  
 gsl\_sf\_synchrotron\_2 (C function), 77  
 gsl\_sf\_synchrotron\_2\_e (C function), 77  
 gsl\_sf\_taylorcoeff (C function), 60  
 gsl\_sf\_taylorcoeff\_e (C function), 60  
 gsl\_sf\_transport\_2 (C function), 77  
 gsl\_sf\_transport\_2\_e (C function), 77  
 gsl\_sf\_transport\_3 (C function), 77  
 gsl\_sf\_transport\_3\_e (C function), 77  
 gsl\_sf\_transport\_4 (C function), 77  
 gsl\_sf\_transport\_4\_e (C function), 77  
 gsl\_sf\_transport\_5 (C function), 77  
 gsl\_sf\_transport\_5\_e (C function), 77  
 gsl\_sf\_zeta (C function), 80  
 gsl\_sf\_zeta\_e (C function), 80  
 gsl\_sf\_zeta\_int (C function), 80  
 gsl\_sf\_zeta\_int\_e (C function), 80  
 gsl\_sf\_zetam1 (C function), 80  
 gsl\_sf\_zetam1\_e (C function), 80  
 gsl\_sf\_zetam1\_int (C function), 80  
 gsl\_sf\_zetam1\_int\_e (C function), 80  
 GSL\_SIGN (C macro), 20  
 gsl\_siman\_copy\_construct\_t (C type), 368  
 gsl\_siman\_copy\_t (C type), 368  
 gsl\_siman\_destroy\_t (C type), 368  
 gsl\_siman\_Efunc\_t (C type), 368  
 gsl\_siman\_metric\_t (C type), 368  
 gsl\_siman\_params\_t (C type), 369  
 gsl\_siman\_print\_t (C type), 368  
 gsl\_siman\_solve (C function), 367  
 gsl\_siman\_step\_t (C type), 368  
 gsl\_sort (C function), 124  
 gsl\_sort2 (C function), 124  
 gsl\_sort\_index (C function), 124  
 gsl\_sort\_largest (C function), 125  
 gsl\_sort\_largest\_index (C function), 125  
 gsl\_sort\_smallest (C function), 125  
 gsl\_sort\_smallest\_index (C function), 125  
 gsl\_sort\_vector (C function), 124  
 gsl\_sort\_vector2 (C function), 124  
 gsl\_sort\_vector\_index (C function), 124  
 gsl\_sort\_vector\_largest (C function), 125  
 gsl\_sort\_vector\_largest\_index (C function), 125  
 gsl\_sort\_vector\_smallest (C function), 125  
 gsl\_sort\_vector\_smallest\_index (C function), 125  
 gsl\_spblas\_dgemm (C function), 619



- gsl\_splblas\_dgemv (*C function*), 619  
 gsl\_splinalg\_itsolve\_alloc (*C function*), 622  
 gsl\_splinalg\_itsolve\_free (*C function*), 622  
 gsl\_splinalg\_itsolve\_iterate (*C function*), 622  
 gsl\_splinalg\_itsolve\_name (*C function*), 622  
 gsl\_splinalg\_itsolve\_normr (*C function*), 622  
 gsl\_splinalg\_itsolve\_type (*C type*), 621  
 gsl\_splinalg\_itsolve\_type.gsl\_splinalg\_itsolve\_type (*C var*), 621  
 gsl\_spline (*C type*), 398  
 gsl\_spline2d (*C type*), 406  
 gsl\_spline2d\_alloc (*C function*), 406  
 gsl\_spline2d\_eval (*C function*), 406  
 gsl\_spline2d\_eval\_deriv\_x (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_x\_e (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_xx (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_xx\_e (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_xy (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_xy\_e (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_y (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_y\_e (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_yy (*C function*), 407  
 gsl\_spline2d\_eval\_deriv\_yy\_e (*C function*), 407  
 gsl\_spline2d\_eval\_e (*C function*), 406  
 gsl\_spline2d\_eval\_extrap (*C function*), 406  
 gsl\_spline2d\_eval\_extrap\_e (*C function*), 406  
 gsl\_spline2d\_free (*C function*), 406  
 gsl\_spline2d\_get (*C function*), 407  
 gsl\_spline2d\_init (*C function*), 406  
 gsl\_spline2d\_min\_size (*C function*), 406  
 gsl\_spline2d\_name (*C function*), 406  
 gsl\_spline2d\_set (*C function*), 407  
 gsl\_spline\_alloc (*C function*), 398  
 gsl\_spline\_eval (*C function*), 398  
 gsl\_spline\_eval\_deriv (*C function*), 398  
 gsl\_spline\_eval\_deriv2 (*C function*), 398  
 gsl\_spline\_eval\_deriv2\_e (*C function*), 398  
 gsl\_spline\_eval\_deriv\_e (*C function*), 398  
 gsl\_spline\_eval\_e (*C function*), 398  
 gsl\_spline\_eval\_integ (*C function*), 398  
 gsl\_spline\_eval\_integ\_e (*C function*), 398  
 gsl\_spline\_free (*C function*), 398  
 gsl\_spline\_init (*C function*), 398  
 gsl\_spline\_min\_size (*C function*), 398  
 gsl\_spline\_name (*C function*), 398  
 gsl\_spmatrix (*C type*), 609  
 gsl\_spmatrix\_add (*C function*), 613  
 gsl\_spmatrix\_alloc (*C function*), 610  
 gsl\_spmatrix\_alloc\_nzmax (*C function*), 610  
 gsl\_spmatrix\_alloc\_nzmax.GSL\_SPMATRIX\_C00 (*C macro*), 610  
 gsl\_spmatrix\_alloc\_nzmax.GSL\_SPMATRIX\_CSC (*C macro*), 610  
 gsl\_spmatrix\_alloc\_nzmax.GSL\_SPMATRIX\_CSR (*C macro*), 610  
 gsl\_spmatrix\_compress (*C function*), 614  
 gsl\_spmatrix\_csc (*C function*), 614  
 gsl\_spmatrix\_csr (*C function*), 614  
 gsl\_spmatrix\_d2sp (*C function*), 614  
 gsl\_spmatrix\_dense\_add (*C function*), 613  
 gsl\_spmatrix\_dense\_sub (*C function*), 613  
 gsl\_spmatrix\_equal (*C function*), 613  
 gsl\_spmatrix\_fprintf (*C function*), 611  
 gsl\_spmatrix\_fread (*C function*), 611  
 gsl\_spmatrix\_free (*C function*), 610  
 gsl\_spmatrix\_fscanf (*C function*), 611  
 gsl\_spmatrix\_fwrite (*C function*), 611  
 gsl\_spmatrix\_get (*C function*), 611  
 gsl\_spmatrix\_memcpy (*C function*), 612  
 gsl\_spmatrix\_min\_index (*C function*), 614  
 gsl\_spmatrix\_minmax (*C function*), 614  
 gsl\_spmatrix\_nnz (*C function*), 613  
 gsl\_spmatrix\_norm1 (*C function*), 613  
 gsl\_spmatrix\_ptr (*C function*), 611  
 gsl\_spmatrix\_realloc (*C function*), 610  
 gsl\_spmatrix\_scale (*C function*), 612  
 gsl\_spmatrix\_scale\_columns (*C function*), 612  
 gsl\_spmatrix\_scale\_rows (*C function*), 612  
 gsl\_spmatrix\_set (*C function*), 611  
 gsl\_spmatrix\_set\_zero (*C function*), 611  
 gsl\_spmatrix\_sp2d (*C function*), 614  
 gsl\_spmatrix\_transpose (*C function*), 612  
 gsl\_spmatrix\_transpose\_memcpy (*C function*), 612  
 gsl\_spmatrix\_type (*C function*), 613  
 gsl\_stats\_absdev (*C function*), 288  
 gsl\_stats\_absdev\_m (*C function*), 288  
 gsl\_stats\_correlation (*C function*), 290  
 gsl\_stats\_covariance (*C function*), 290  
 gsl\_stats\_covariance\_m (*C function*), 290  
 gsl\_stats\_gastwirth\_from\_sorted\_data (*C function*), 294  
 gsl\_stats\_kurtosis (*C function*), 289  
 gsl\_stats\_kurtosis\_m\_sd (*C function*), 289  
 gsl\_stats\_lag1\_autocorrelation (*C function*), 289  
 gsl\_stats\_lag1\_autocorrelation\_m (*C function*), 289  
 gsl\_stats\_mad (*C function*), 295  
 gsl\_stats\_mad0 (*C function*), 294  
 gsl\_stats\_max (*C function*), 292  
 gsl\_stats\_max\_index (*C function*), 292  
 gsl\_stats\_mean (*C function*), 287  
 gsl\_stats\_median (*C function*), 293  
 gsl\_stats\_median\_from\_sorted\_data (*C function*), 293  
 gsl\_stats\_min (*C function*), 292  
 gsl\_stats\_min\_index (*C function*), 292  
 gsl\_stats\_minmax (*C function*), 292

- gsl\_stats\_minmax\_index (C function), 292  
 gsl\_stats\_Qn0\_from\_sorted\_data (C function), 295  
 gsl\_stats\_Qn\_from\_sorted\_data (C function), 295  
 gsl\_stats\_quantile\_from\_sorted\_data (C function), 293  
 gsl\_stats\_sd (C function), 288  
 gsl\_stats\_sd\_m (C function), 288  
 gsl\_stats\_sd\_with\_fixed\_mean (C function), 288  
 gsl\_stats\_select (C function), 293  
 gsl\_stats\_skew (C function), 289  
 gsl\_stats\_skew\_m\_sd (C function), 289  
 gsl\_stats\_Sn0\_from\_sorted\_data (C function), 295  
 gsl\_stats\_Sn\_from\_sorted\_data (C function), 295  
 gsl\_stats\_spearman (C function), 290  
 gsl\_stats\_trmean\_from\_sorted\_data (C function), 294  
 gsl\_stats\_tss (C function), 288  
 gsl\_stats\_tss\_m (C function), 288  
 gsl\_stats\_variance (C function), 287  
 gsl\_stats\_variance\_m (C function), 287  
 gsl\_stats\_variance\_with\_fixed\_mean (C function), 288  
 gsl\_stats\_wabsdev (C function), 291  
 gsl\_stats\_wabsdev\_m (C function), 291  
 gsl\_stats\_wkurtosis (C function), 292  
 gsl\_stats\_wkurtosis\_m\_sd (C function), 292  
 gsl\_stats\_wmean (C function), 290  
 gsl\_stats\_wsd (C function), 291  
 gsl\_stats\_wsd\_m (C function), 291  
 gsl\_stats\_wsd\_with\_fixed\_mean (C function), 291  
 gsl\_stats\_wskew (C function), 291  
 gsl\_stats\_wskew\_m\_sd (C function), 292  
 gsl\_stats\_wtss (C function), 291  
 gsl\_stats\_wtss\_m (C function), 291  
 gsl\_stats\_wvariance (C function), 290  
 gsl\_stats\_wvariance\_m (C function), 291  
 gsl\_stats\_wvariance\_with\_fixed\_mean (C function), 291  
 gsl\_strerror (C function), 14  
 gsl\_sum\_levin\_u\_accel (C function), 419  
 gsl\_sum\_levin\_u\_alloc (C function), 419  
 gsl\_sum\_levin\_u\_free (C function), 419  
 gsl\_sum\_levin\_u\_workspace (C type), 419  
 gsl\_sum\_levin\_utrunc\_accel (C function), 420  
 gsl\_sum\_levin\_utrunc\_alloc (C function), 420  
 gsl\_sum\_levin\_utrunc\_free (C function), 420  
 gsl\_sum\_levin\_utrunc\_workspace (C type), 420  
 gsl\_vector (C type), 85  
 gsl\_vector\_add (C function), 91  
 gsl\_vector\_add\_constant (C function), 91  
 gsl\_vector\_alloc (C function), 86  
 gsl\_vector\_axpby (C function), 91  
 gsl\_vector\_calloc (C function), 86  
 gsl\_vector\_complex\_conj\_memcpy (C function), 90  
 gsl\_vector\_complex\_const\_imag (C function), 89  
 gsl\_vector\_complex\_const\_real (C function), 89  
 gsl\_vector\_complex\_imag (C function), 89  
 gsl\_vector\_complex\_real (C function), 89  
 gsl\_vector\_const\_ptr (C function), 87  
 gsl\_vector\_const\_subvector (C function), 88  
 gsl\_vector\_const\_subvector\_with\_stride (C function), 88  
 gsl\_vector\_const\_view (C type), 88  
 gsl\_vector\_const\_view\_array (C function), 89  
 gsl\_vector\_const\_view\_array\_with\_stride (C function), 89  
 gsl\_vector\_div (C function), 91  
 gsl\_vector\_equal (C function), 92  
 gsl\_vector\_fprintf (C function), 87  
 gsl\_vector\_fread (C function), 87  
 gsl\_vector\_free (C function), 86  
 gsl\_vector\_fscanf (C function), 88  
 gsl\_vector\_fwrite (C function), 87  
 gsl\_vector\_get (C function), 87  
 gsl\_vector\_isneg (C function), 92  
 gsl\_vector\_isnonneg (C function), 92  
 gsl\_vector\_isnull (C function), 92  
 gsl\_vector\_ispos (C function), 92  
 gsl\_vector\_max (C function), 91  
 gsl\_vector\_max\_index (C function), 91  
 gsl\_vector\_memcpy (C function), 90  
 gsl\_vector\_min (C function), 91  
 gsl\_vector\_min\_index (C function), 91  
 gsl\_vector\_minmax (C function), 91  
 gsl\_vector\_minmax\_index (C function), 91  
 gsl\_vector\_mul (C function), 91  
 gsl\_vector\_ptr (C function), 87  
 gsl\_vector\_reverse (C function), 90  
 gsl\_vector\_scale (C function), 91  
 gsl\_vector\_set (C function), 87  
 gsl\_vector\_set\_all (C function), 87  
 gsl\_vector\_set\_basis (C function), 87  
 gsl\_vector\_set\_zero (C function), 87  
 gsl\_vector\_sub (C function), 91  
 gsl\_vector\_subvector (C function), 88  
 gsl\_vector\_subvector\_with\_stride (C function), 88  
 gsl\_vector\_sum (C function), 91  
 gsl\_vector\_swap (C function), 90  
 gsl\_vector\_swap\_elements (C function), 90  
 gsl\_vector\_view (C type), 88  
 gsl\_vector\_view\_array (C function), 89  
 gsl\_vector\_view\_array\_with\_stride (C function), 89  
 gsl\_wavelet (C type), 423  
 gsl\_wavelet2d\_nstransform (C function), 426  
 gsl\_wavelet2d\_nstransform\_forward (C function), 426

- gsl\_wavelet2d\_nstransform\_inverse (*C function*),  
     426  
 gsl\_wavelet2d\_nstransform\_matrix (*C function*),  
     426  
 gsl\_wavelet2d\_nstransform\_matrix\_forward (*C*  
     *function*), 426  
 gsl\_wavelet2d\_nstransform\_matrix\_inverse (*C*  
     *function*), 426  
 gsl\_wavelet2d\_transform (*C function*), 425  
 gsl\_wavelet2d\_transform\_forward (*C function*),  
     425  
 gsl\_wavelet2d\_transform\_inverse (*C function*),  
     425  
 gsl\_wavelet2d\_transform\_matrix (*C function*), 425  
 gsl\_wavelet2d\_transform\_matrix\_forward (*C*  
     *function*), 425  
 gsl\_wavelet2d\_transform\_matrix\_inverse (*C*  
     *function*), 425  
 gsl\_wavelet\_alloc (*C function*), 423  
 gsl\_wavelet\_free (*C function*), 424  
 gsl\_wavelet\_name (*C function*), 424  
 gsl\_wavelet\_transform (*C function*), 424  
 gsl\_wavelet\_transform\_forward (*C function*), 424  
 gsl\_wavelet\_transform\_inverse (*C function*), 424  
 gsl\_wavelet\_type (*C type*), 423  
 gsl\_wavelet\_type.gsl\_wavelet\_bspline (*C var*),  
     424  
 gsl\_wavelet\_type.gsl\_wavelet\_bspline\_centered  
     (*C var*), 424  
 gsl\_wavelet\_type.gsl\_wavelet\_daubechies (*C*  
     *var*), 423  
 gsl\_wavelet\_type.gsl\_wavelet\_daubechies\_centered  
     (*C var*), 423  
 gsl\_wavelet\_type.gsl\_wavelet\_haar (*C var*), 424  
 gsl\_wavelet\_type.gsl\_wavelet\_haar\_centered  
     (*C var*), 424  
 gsl\_wavelet\_workspace (*C type*), 424  
 gsl\_wavelet\_workspace\_alloc (*C function*), 424  
 gsl\_wavelet\_workspace\_free (*C function*), 424  
 Gumbel distribution (*Type 1*), 267  
 Gumbel distribution (*Type 2*), 268
- ## H
- Haar wavelets, 424  
 Hankel transforms, discrete, 429  
 HAVE\_INLINE, 7  
 hazard function, normal distribution, 53  
 HBOOK, 356  
 header files, including, 5  
 heapsort, 121  
 HEMM, Level-3 BLAS, 136  
 HEMV, Level-2 BLAS, 134  
 HER, Level-2 BLAS, 134  
 HER2, Level-2 BLAS, 135  
 HER2K, Level-3 BLAS, 137  
 HERK, Level-3 BLAS, 137  
 Hermite functions, 63  
 Hermite functions, derivatives, 64  
 Hermite functions, zeros, 65  
 Hermite polynomials, 62  
 Hermite polynomials, derivatives, 63  
 Hermite polynomials, zeros, 65  
 hermitian matrix, complex, eigensystem, 172  
 Hessenberg decomposition, 159  
 Hessenberg triangular decomposition, 160  
 histogram statistics, 338  
 histogram, from ntuple, 353  
 histograms, 333  
 histograms, random sampling from, 340  
 Householder linear solver, 162  
 Householder matrix, 161  
 Householder transformation, 161  
 how to report, 2  
 Hurwitz Zeta Function, 80  
 HYBRID algorithm, unscaled without  
     derivatives, 462  
 HYBRID algorithms for nonlinear systems, 460  
 HYBRIDJ algorithm, 461  
 HYBRIDS algorithm, scaled without  
     derivatives, 462  
 HYBRIDSJ algorithm, 460  
 hydrogen atom, 45  
 hyperbolic cosine, inverse, 18  
 hyperbolic functions, complex numbers, 27  
 hyperbolic integrals, 55  
 hyperbolic sine, inverse, 18  
 hyperbolic space, 68  
 hyperbolic tangent, inverse, 18  
 hypergeometric functions, 65  
 hypergeometric random variates, 278  
 hypot, 18  
 hypot function, special functions, 78
- ## I
- I(x), Bessel Functions, 39  
 i(x), Bessel Functions, 42  
 identity matrix, 95  
 identity permutation, 107  
 IEEE exceptions, 639  
 IEEE floating point, 635  
 IEEE format for floating point numbers, 637  
 IEEE infinity, defined as a macro, 17  
 IEEE NaN, defined as a macro, 18  
 illumination, units of, 632  
 imperial units, 629  
 Implicit Euler method, 385  
 Implicit Runge-Kutta method, 385  
 importance sampling, VEGAS, 361

- including GSL header files, 5
- incomplete Beta function, normalized, 61
- incomplete Fermi-Dirac integral, 58
- incomplete Gamma function, 60
- indirect sorting, 124
- indirect sorting, of vector elements, 124
- infinity, defined as a macro, 17
- infinity, IEEE format, 637
- info-gsl mailing list, 2
- initial value problems, differential equations, 381
- initializing matrices, 95
- initializing vectors, 87
- inline functions, 7
- integer powers, 75
- integrals, exponential, 55
- integration, numerical (*quadrature*), 198
- interpolating quadrature, 208
- interpolation, 394
- interpolation, using Chebyshev polynomials, 413
- inverse complex trigonometric functions, 26
- inverse cumulative distribution functions, 233
- inverse hyperbolic cosine, 18
- inverse hyperbolic functions, complex numbers, 27
- inverse hyperbolic sine, 18
- inverse hyperbolic tangent, 18
- inverse of a matrix, by LU decomposition, 142
- inverting a permutation, 108
- Irregular Cylindrical Bessel Functions, 39
- Irregular Modified Bessel Functions, Fractional Order, 44
- Irregular Modified Cylindrical Bessel Functions, 40
- Irregular Modified Spherical Bessel Functions, 43
- Irregular Spherical Bessel Functions, 42
- iterating through combinations, 114
- iterating through multisets, 118
- iterating through permutations, 108
- iterative refinement of solutions in linear systems, 142

## J

- $J(x)$ , Bessel Functions, 38
- $j(x)$ , Bessel Functions, 41
- Jacobi elliptic functions, 52
- Jacobi orthogonalization, 154
- Jacobian matrix, ODEs, 383
- Jacobian matrix, root finding, 455

## K

- $K(x)$ , Bessel Functions, 40
- $k(x)$ , Bessel Functions, 43
- knots, basis splines, 568
- kurtosis, 288

## L

- Laguerre functions, 67
- Lambert function, 67
- Landau distribution, 249
- LAPACK, 182
- Laplace distribution, 244
- large dense linear least squares, 493
- large linear least squares, normal equations, 494
- large linear least squares, routines, 495
- large linear least squares, steps, 495
- large linear least squares, TSQR, 494
- LD\_LIBRARY\_PATH, 6
- ldexp, 19
- LDL decomposition, 157
- LDLT decomposition, 157
- LDLT decomposition, banded, 167
- leading dimension, matrices, 94
- least squares fit, 480
- least squares troubleshooting, 497
- least squares, covariance of best-fit parameters, 535
- least squares, nonlinear, 520
- least squares, regularized, 485
- least squares, robust, 489
- Legendre forms of elliptic integrals, 50
- Legendre functions, 68
- Legendre polynomials, 68
- length, computed accurately using hypot, 18
- length, computed accurately using hypot3, 18
- Levenberg-Marquardt algorithm, 523
- Levenberg-Marquardt algorithm, geodesic acceleration, 523
- Levin u-transform, 418
- Levy distribution, 250
- Levy distribution, skew, 251
- libraries, linking with, 6
- libraries, shared, 6
- license of GSL, 1
- light, units of, 632
- linear algebra, 139
- linear algebra, BLAS, 127
- linear algebra, sparse, 619
- linear interpolation, 396
- linear least squares, large, 493
- linear regression, 482
- linear systems, 141
- linear systems, refinement of solutions, 142



- linking with GSL libraries, 6
  - location estimation, 293
  - loglp, 18
  - logarithm and related functions, 73
  - logarithm of Beta function, 61
  - logarithm of combinatorial factor  $C(m, 60)$
  - logarithm of complex number, 26
  - logarithm of cosh function, special functions, 78
  - logarithm of double factorial, 59
  - logarithm of factorial, 59
  - logarithm of Gamma function, 58
  - logarithm of Pochhammer symbol, 60
  - logarithm of sinh function, 78
  - logarithm of the determinant of a matrix, 142
  - logarithm, computed accurately near 1, 18
  - Logarithmic random variates, 279
  - Logistic distribution, 263
  - Lognormal distribution, 255
  - low discrepancy sequences, 229
  - Low-level CBLAS, 654
  - LQ decomposition, 150
  - LU decomposition, 141
  - LU decomposition, banded, 165
- ## M
- macros for mathematical constants, 17
  - magnitude of complex number, 25
  - mailing list, 2
  - mailing list archives, 3
  - mailing list for GSL announcements, 2
  - mantissa, IEEE format, 637
  - mass, units of, 630
  - mathematical constants, defined as macros, 17
  - mathematical functions, elementary, 16
  - Mathieu Function Characteristic Values, 74
  - Mathieu functions, 74
  - matrices, 94
  - matrices, banded, 163
  - matrices, initializing, 95
  - matrices, range-checking, 95
  - matrices, sparse, 605
  - matrix determinant, 142
  - matrix diagonal, 99
  - matrix factorization, 139
  - matrix inverse, 142
  - matrix square root, Cholesky decomposition, 154
  - matrix subdiagonal, 99
  - matrix superdiagonal, 99
  - matrix, constant, 95
  - matrix, identity, 95
  - matrix, operations, 127
  - matrix, zero, 95
  - max, 285
  - maximal phase, Daubechies wavelets, 423
  - maximization, see minimization, 445
  - maximum of two numbers, 20
  - maximum value, from histogram, 338
  - mean, 285
  - mean value, from histogram, 338
  - mean, trimmed, 294
  - mean, truncated, 294
  - median absolute deviation, 294
  - Mills' ratio, inverse, 53
  - min, 285
  - minimization, BFGS algorithm, 474
  - minimization, caveats, 448
  - minimization, conjugate gradient algorithm, 474
  - minimization, multidimensional, 467
  - minimization, one-dimensional, 445
  - minimization, overview, 447
  - minimization, Polak-Ribiere algorithm, 474
  - minimization, providing a function to minimize, 449
  - minimization, simplex algorithm, 475
  - minimization, steepest descent algorithm, 474
  - minimization, stopping parameters, 450
  - minimum finding, Brent's method, 450
  - minimum finding, golden section algorithm, 450
  - minimum of two numbers, 20
  - minimum value, from histogram, 338
  - MINPACK, minimization algorithms, 460
  - MISCFUN, 82
  - MISER monte carlo integration, 359
  - Mixed-radix FFT, complex data, 186
  - Mixed-radix FFT, real data, 193
  - Modified Bessel Functions, Fractional Order, 44
  - Modified Cholesky Decomposition, 157
  - Modified Clenshaw-Curtis quadrature, 200
  - Modified Cylindrical Bessel Functions, 39
  - Modified Givens Rotation, BLAS, 132
  - Modified Newton's method for nonlinear systems, 461
  - Modified Spherical Bessel Functions, 42
  - Monte Carlo integration, 356
  - moving maximum, 307
  - moving mean, 306
  - moving median, 308
  - moving median absolute deviation, 308
  - moving minimum, 307
  - moving quantile range, 309
  - moving standard deviation, 306
  - moving sum, 307
  - moving variance, 306

moving window accumulators, 309  
moving window statistics, 304  
moving window, allocation, 306  
MRG, multiple recursive random number generator, 221  
MT19937 random number generator, 220  
multi-parameter regression, 483  
multidimensional integration, 356  
multidimensional root finding, Broyden algorithm, 462  
multidimensional root finding, overview, 455  
multidimensional root finding, providing a function to solve, 456  
Multimin, caveats, 469  
Multinomial distribution, 274  
multiplication, 49  
multisets, 116  
multistep methods, ODEs, 385

## N

n), 59, 60  
N-dimensional random direction vector, 265  
NaN, defined as a macro, 18  
nautical units, 629  
Negative Binomial distribution, random variates, 275  
Nelder-Mead simplex algorithm for minimization, 475  
Newton algorithm, discrete, 462  
Newton algorithm, globally convergent, 461  
Newton's method for finding roots, 439  
Newton's method for systems of nonlinear equations, 461  
Niederreiter sequence, 229  
NIST Statistical Reference Datasets, 520  
non-normalized incomplete Gamma function, 60  
nonlinear equation, solutions of, 432  
nonlinear fitting, stopping parameters, convergence, 533  
nonlinear functions, minimization, 445  
nonlinear least squares, 520  
nonlinear least squares, dogleg, 523  
nonlinear least squares, double dogleg, 524  
nonlinear least squares, levenberg-marquardt, 523  
nonlinear least squares, levenberg-marquardt, geodesic acceleration, 523  
nonlinear least squares, overview, 521  
nonlinear systems of equations, solution of, 453  
nonsymmetric matrix, real, eigensystem, 172  
Nordsieck form, 385  
normalized form, IEEE format, 637

normalized incomplete Beta function, 61  
Not-a-number, defined as a macro, 18  
NRM2, Level-1 BLAS, 131  
ntuples, 349  
nuclear physics, constants, 628  
numerical constants, defined as macros, 17  
numerical derivatives, 409  
numerical integration (*quadrature*), 198

## O

obtaining GSL, 2  
ODEs, initial value problems, 381  
online statistics, 298, 304  
optimization, combinatorial, 366  
optimization, see minimization, 445  
optimized functions, alternatives, 9  
ordering, matrix elements, 94  
ordinary differential equations, initial value problem, 381  
oscillatory functions, numerical integration of, 204  
overflow, IEEE exceptions, 639

## P

Pareto distribution, 264  
PAW, 356  
permutations, 105  
physical constants, 626  
physical dimension, matrices, 94  
pi, defined as a macro, 17  
Pivoted Cholesky Decomposition, 156  
plain Monte Carlo, 358  
Pochhammer symbol, 60  
Poisson random numbers, 271  
Polak-Ribiere algorithm, minimization, 474  
polar form of complex numbers, 23  
polar to rectangular conversion, 79  
polygamma functions, 76  
polynomial evaluation, 29  
polynomial interpolation, 396  
polynomials, roots of, 28  
power function, 75  
power of complex number, 26  
power, units of, 631  
precision, IEEE arithmetic, 639  
predictor-corrector method, ODEs, 385  
prefixes, 633  
pressure, 631  
Prince-Dormand, Runge-Kutta method, 385  
printers units, 630  
probability distribution, from histogram, 340  
probability distributions, from histograms, 339  
projection of ntuples, 353

psi function, 76

## Q

QAG quadrature algorithm, 201  
 QAGI quadrature algorithm, 202  
 QAGP quadrature algorithm, 202  
 QAGS quadrature algorithm, 202  
 QAWC quadrature algorithm, 203  
 QAWF quadrature algorithm, 205  
 QAWO quadrature algorithm, 204  
 QAWS quadrature algorithm, 203  
 QL decomposition, 151  
 Qn statistic, 295  
 QNG quadrature algorithm, 201  
 QR decomposition, 142  
 QR decomposition with column pivoting, 148  
 QUADPACK, 198  
 quadratic equation, solving, 30  
 quadrature, 198  
 quadrature, fixed point, 208  
 quadrature, interpolating, 208  
 quantile functions, 233  
 quasi-random sequences, 229

## R

R250 shift-register random number generator, 224  
 Racah coefficients, 47  
 Radial Mathieu Functions, 75  
 radioactivity, 632  
 Radix-2 FFT for real data, 191  
 Radix-2 FFT, complex data, 185  
 rand, BSD random number generator, 223  
 rand48 random number generator, 223  
 random number distributions, 233  
 random number generators, 213  
 random sampling from histograms, 340  
 RANDU random number generator, 225  
 RANF random number generator, 224  
 range, 285  
 range-checking for matrices, 95  
 range-checking for vectors, 86  
 RANLUX random number generator, 221  
 RANLXD random number generator, 221  
 RANLXS random number generator, 220  
 RANMAR random number generator, 224, 225  
 Rayleigh distribution, 247  
 Rayleigh Tail distribution, 248  
 real nonsymmetric matrix, eigensystem, 172  
 real symmetric matrix, eigensystem, 171  
 Reciprocal Gamma function, 59  
 rectangular to polar conversion, 79  
 recursive stratified sampling, MISER, 359  
 reduction of angular variables, 79

refinement of solutions in linear systems, 142  
 regression, least squares, 480  
 regression, ridge, 485  
 regression, robust, 489  
 regression, Tikhonov, 485  
 Regular Bessel Functions, Fractional Order, 43  
 Regular Bessel Functions, Zeros of, 44  
 Regular Cylindrical Bessel Functions, 38  
 Regular Modified Bessel Functions, Fractional Order, 44  
 Regular Modified Cylindrical Bessel Functions, 39  
 Regular Modified Spherical Bessel Functions, 42  
 Regular Spherical Bessel Functions, 41  
 Regulated Gamma function, 58  
 relative Pochhammer symbol, 60  
 reporting bugs in GSL, 2  
 representations of complex numbers, 23  
 resampling from histograms, 339  
 residual, in nonlinear systems of equations, 460  
 reversing a permutation, 108  
 ridge regression, 485  
 Riemann Zeta Function, 80  
 RK2, Runge-Kutta method, 385  
 RK4, Runge-Kutta method, 385  
 RKF45, Runge-Kutta-Fehlberg method, 385  
 robust location estimators, 293  
 robust regression, 489  
 robust scale estimators, 294  
 rolling maximum, 307  
 rolling mean, 306  
 rolling median, 308  
 rolling median absolute deviation, 308  
 rolling minimum, 307  
 rolling quantile range, 309  
 rolling standard deviation, 306  
 rolling sum, 307  
 rolling variance, 306  
 rolling window accumulators, 309  
 root finding, 432  
 root finding, bisection algorithm, 438  
 root finding, Brent's method, 439  
 root finding, caveats, 433  
 root finding, false position algorithm, 439  
 root finding, initial guess, 437  
 root finding, Newton's method, 439  
 root finding, overview, 433  
 root finding, providing a function to solve, 435  
 root finding, search bounds, 437

root finding, secant method, 439  
root finding, Steffenson's method, 440  
root finding, stopping parameters, 437, 459  
roots, 432  
ROTG, Level-1 BLAS, 132  
rounding mode, 639  
Runge-Kutta Cash-Karp method, 385  
Runge-Kutta methods, ordinary differential equations, 385  
Runge-Kutta Prince-Dormand method, 385  
running statistics, 298

## S

safe comparison of floating point numbers, 21  
safeguarded step-length algorithm, 451  
sampling from histograms, 339, 340  
SAXPY, Level-1 BLAS, 132  
SCAL, Level-1 BLAS, 132  
scale estimation, 294  
schedule, cooling, 367  
se( $q, x$ ), Mathieu function, 75  
secant method for finding roots, 439  
selection function, ntuples, 352  
series, acceleration, 418  
shared libraries, 6  
shell prompt, 3  
Shi( $x$ ), 55  
shift-register random number generator, 224  
Si( $x$ ), 56  
sign bit, IEEE format, 637  
sign of the determinant of a matrix, 142  
simplex algorithm, minimization, 475  
simulated annealing, 366  
sin, of complex number, 26  
sine function, special functions, 78  
single precision, IEEE format, 637  
singular functions, numerical integration of, 203  
singular points, specifying positions in quadrature, 202  
singular value decomposition, 153  
Skew Levy distribution, 251  
skewness, 288  
slope, see numerical derivative, 409  
Sn statistic, 295  
Sobol sequence, 229  
solution of, 141  
solution of linear system by Householder transformations, 162  
solution of linear systems,  $Ax=b$ , 139  
solving a nonlinear equation, 432  
solving nonlinear systems of equations, 453  
sorting, 121  
sorting eigenvalues and eigenvectors, 177

sorting vector elements, 124  
source code, reuse in applications, 11  
sparse BLAS, 617  
sparse BLAS, references, 619  
sparse linear algebra, 619  
sparse linear algebra, examples, 622  
sparse linear algebra, iterative solvers, 621  
sparse linear algebra, overview, 621  
sparse linear algebra, references, 626  
sparse matrices, 605  
sparse matrices, accessing elements, 610  
sparse matrices, allocation, 610  
sparse matrices, BLAS operations, 619  
sparse matrices, compressed column storage, 608  
sparse matrices, compressed row storage, 608  
sparse matrices, compressed sparse column, 608  
sparse matrices, compressed sparse row, 608  
sparse matrices, compression, 614  
sparse matrices, conversion, 614  
sparse matrices, coordinate format, 608  
sparse matrices, copying, 612  
sparse matrices, data types, 607  
sparse matrices, examples, 614  
sparse matrices, exchanging rows and columns, 612  
sparse matrices, initializing elements, 611  
sparse matrices, iterative solvers, 621  
sparse matrices, min/max elements, 613  
sparse matrices, operations, 612  
sparse matrices, overview, 609  
sparse matrices, properties, 613  
sparse matrices, reading, 611  
sparse matrices, references, 617  
sparse matrices, storage formats, 607  
sparse matrices, triplet format, 608  
sparse matrices, writing, 611  
sparse, iterative solvers, 621  
special functions, 33, 78  
Spherical Bessel Functions, 41  
spherical harmonics, 68  
spherical random variates, 2D, 265  
spherical random variates, 3D, 265  
spherical random variates, N-dimensional, 265  
spline, 394  
splines, basis, 565  
square root of a matrix, Cholesky decomposition, 154  
square root of complex number, 26  
standard deviation, 285  
standard deviation, from histogram, 338  
standards conformance, ANSI C, 4  
Statistical Reference Datasets (*StRD*), 520

statistics, 285  
 statistics, from histogram, 338  
 statistics, moving window, 304  
 steepest descent algorithm, minimization, 474  
 Steffenson's method for finding roots, 440  
 stratified sampling in Monte Carlo  
     integration, 356  
 stride, of vector index, 85  
 Student t-distribution, 260  
 subdiagonal, of a matrix, 99  
 summation, acceleration, 418  
 superdiagonal, matrix, 99  
 SVD, 153  
 SWAP, Level-1 BLAS, 131  
 swapping permutation elements, 108  
 SYMM, Level-3 BLAS, 135  
 symmetric matrices, banded, 164  
 symmetric matrix, real, eigensystem, 171  
 SYMV, Level-2 BLAS, 133  
 synchrotron functions, 77  
 SYR, Level-2 BLAS, 134  
 SYR2, Level-2 BLAS, 134  
 SYR2K, Level-3 BLAS, 137  
 SYRK, Level-3 BLAS, 137  
 systems of equations, nonlinear, 453

**T**

t-distribution, 260  
 t-test, 285  
 tangent of complex number, 26  
 Tausworthe random number generator, 222  
 Taylor coefficients, computation of, 60  
 testing combination for validity, 114  
 testing multiset for validity, 118  
 testing permutation for validity, 108  
 thermal energy, units of, 631  
 Tikhonov regression, 485  
 time units, 629  
 trailing dimension, matrices, 94  
 transformation, Householder, 161  
 transforms, Hankel, 429  
 transforms, wavelet, 422  
 transport functions, 77  
 traveling salesman problem, 371  
 triangular systems, 163  
 tridiagonal decomposition, 158, 159  
 tridiagonal systems, 162  
 trigonometric functions, 78  
 trigonometric functions of complex numbers,  
     26  
 trigonometric integrals, 56  
 trimmed mean, 294  
 TRMM, Level-3 BLAS, 136  
 TRMV, Level-2 BLAS, 133

TRSM, Level-3 BLAS, 136  
 TRSV, Level-2 BLAS, 133  
 truncated mean, 294  
 TSP, 371  
 TT800 random number generator, 224  
 two dimensional Gaussian distribution, 241,  
     242  
 two dimensional histograms, 341  
 two-sided exponential distribution, 244  
 Type 1 Gumbel distribution, random variates,  
     267  
 Type 2 Gumbel distribution, 268

**U**

u-transform for series, 418  
 underflow, IEEE exceptions, 639  
 uniform distribution, 254  
 units of, 631–633  
 units, conversion of, 626  
 units, imperial, 629  
 Unix random number generators, rand, 223  
 Unix random number generators, rand48, 223  
 unnormalized incomplete Gamma function, 60  
 unweighted linear fits, 480

**V**

value function, ntuples, 352  
 Van der Pol oscillator, example, 390  
 variance, 285  
 variance, from histogram, 338  
 variance-covariance matrix, linear fits, 482  
 VAX random number generator, 225  
 vector, operations, 127  
 vector, sorting elements of, 124  
 vectors, 85  
 vectors, initializing, 87  
 vectors, range-checking, 86  
 VEGAS Monte Carlo integration, 361  
 viscosity, 632  
 volume units, 630

**W**

W function, 67  
 warning options, 645  
 warranty (*none*), 2  
 wavelet transforms, 422  
 website, developer information, 3  
 Weibull distribution, 266  
 weight, units of, 630  
 weighted linear fits, 480  
 Wigner coefficients, 47  
 Wishart random variates, 280

## Y

$Y(x)$ , Bessel Functions, 39

$y(x)$ , Bessel Functions, 42

## Z

zero finding, 432

zero matrix, 95

zero, IEEE format, 637

Zeros of Regular Bessel Functions, 44

Zeta functions, 79

Ziggurat method, 237