

# The mpFormulaC Library and Toolbox Manual

Dietrich Hadler  
Helge Hadler  
Thomas Hadler

July 2015  
Version 0.1, Alpha 1 (Pre-Release)

Original Authors of the GMP Manual: Torbjörn Granlund and the GMP Development Team

Original Authors of the MPIR Manual: William Hart and the MPIR Team

Original Authors of the MPFR Manual: Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Philippe Théveny, Paul Zimmermann and the MPFR Team

Original Authors of the MPC Manual: Andreas Enge, Philippe Théveny and Paul Zimmermann

Original Authors of the FLINT Manual: William Hart and the FLINT Team

Original Authors of the ARB Manual: Frederik Johanson and the ARB Team

Subsequent modifications: Dietrich Hadler, Helge Hadler and Thomas Hadler

Copyright © 1991, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010 Free Software Foundation, Inc.

Copyright © 2002, 2003, 2004, 2005, 2007, 2008, 2009, 2010 Andreas Enge, Philippe Théveny, Paul Zimmermann

Copyright © 2008, 2009, 2010 William Hart

Copyright © 2010 - 2015 Frederik Johanson

Copyright © 2014 Dietrich Hadler, Helge Hadler, Thomas Hadler

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 the Invariant Sections being “GNU General Public License” and “Free Software Needs Free Documentation”, the Front-Cover text being “A GNU Manual”, and with the Back-Cover Text being (a) (see below). A copy of the license is included in the section entitled “GNU Free Documentation License” (a) The Back-Cover Text is: “You have the freedom to copy and modify this GNU Manual, like GNU software”.

# Preface

The mpFormulaC library provides a comprehensive set of number-theoretical functions, and elementary and special real and complex functions in multiprecision ball arithmetic.

It is based on a number of well-established libraries, which implement or support multiprecision, interval, or ball arithmetic: GMP, MPFR, FLINT, ARB, libmpdec, MPFRC++, Eigen, Boost Math Toolkit and Boost Random.

Additional planned functionality includes integration in LibreOffice Calc (Windows, Mac OSX, GNU/Linux), with multiprecision support for the numerical functions of this spreadsheet program.

The library is currently still in pre-alpha stage, and much of the planned functionality is still missing.

This manual is divided in various parts, which reflect different levels of confidence regarding the accuracy of the results.

Part II: Functions based on GMP and libraries which are related to GMP.

Functions in this part come optionally with a guaranteed error bound, which can (in principle) be made arbitrarily small. Based mostly on MPFR, MPC, FLINT, ARB and libmpdec.

Part III: Functions based on Eigen.

Functions in this part include functions which do not guarantee an error bound, but provide error tracking. This includes a comprehensive selection of complex and real linear algebra functions, based on Eigen.

Part IV: Special Functions based on Boost.

Functions in this part include random numbers, distribution functions, and Ordinary Differential Equations. Partially in double precision.

The use of these functions in various environments is described in some detail in the appendices:

Appendix A: Interfaces to the C family of languages.

Appendix B: An interface to R (Statistical System).

Appendix C: An interface to Python (CPython).

Appendix D: An interface to LibreOffice.

Appendix E: Interfaces to languages with CLR support.

Appendix F: Interfaces to languages with COM support.

If you want to re-build or change the library and/or toolbox, have a look at appendix G.

Finally, the mpFormulaC Library and Toolbox would not exist without the many authors and contributors of the underlying libraries. They are acknowledged in appendix H.

Dietrich Hadler  
Helge Hadler  
Thomas Hadler

# Contents

<b>Preface</b>	<b>i</b>
<b>Contents</b>	<b>iii</b>
<b>List of Tables</b>	<b>xxxv</b>
<b>List of Figures</b>	<b>xxxvi</b>
<b>I Getting Started</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Overview: Features and Setup . . . . .	2
1.1.1 Features . . . . .	2
1.1.2 The mpFormulaC Library . . . . .	2
1.1.3 The mpFormulaC Toolbox . . . . .	3
1.1.4 System Requirement . . . . .	3
1.1.5 Installation . . . . .	4
1.2 License . . . . .	4
1.3 No Warranty . . . . .	4
1.4 Related Software . . . . .	4
<b>2 Tutorials</b>	<b>5</b>
2.1 Why multi-precision arithmetic? . . . . .	5
2.1.1 Example 1: Sums . . . . .	5
2.1.2 Example 2: Standard Deviation . . . . .	5
2.1.3 Example 3: Overflow and underflow . . . . .	5
2.1.4 Example 4: Polynomials . . . . .	6
2.1.5 Example 5: Trigonometric Functions . . . . .	6
2.1.6 Example 6: Logarithms and Exponential Functions . . . . .	6
2.1.7 Example 7: Linear Algebra . . . . .	6
2.2 Graphics using Latex . . . . .	7

2.3	Graphics using .NET Framework . . . . .	8
2.3.1	Surface plots for bivariate real functions . . . . .	9
2.3.2	3D Plots of parametric functions . . . . .	10
2.3.3	Surface plots of complex functions . . . . .	13
<b>3</b>	<b>Python: Built-in numerical types</b>	<b>15</b>
3.1	Truth Value Testing . . . . .	15
3.2	Boolean Operations: and, or, not . . . . .	15
3.3	Comparisons . . . . .	16
3.4	Numeric Types - int, float, complex . . . . .	16
3.5	Long integers . . . . .	18
3.5.1	Bitwise Operations on Integer Types . . . . .	18
3.5.2	Additional Methods on Integer Types . . . . .	18
3.5.3	Additional Methods on Float . . . . .	20
3.6	Fractions . . . . .	22
3.6.1	Properties . . . . .	23
3.6.2	Methods . . . . .	23
<b>II</b>	<b>Functions With Error bounds</b>	<b>25</b>
<b>4</b>	<b>Basic Usage</b>	<b>26</b>
4.1	Number types . . . . .	26
4.1.1	Setting the precision . . . . .	27
4.1.2	Temporarily changing the precision . . . . .	28
4.1.3	Providing correct input . . . . .	29
4.1.4	Printing . . . . .	30
4.1.5	Contexts . . . . .	31
4.1.6	Common interface . . . . .	31
4.1.7	Arbitrary-precision floating-point (mp) . . . . .	33
4.1.8	Arbitrary-precision interval arithmetic (iv) . . . . .	33
4.1.9	Fast low-precision arithmetic (fp) . . . . .	36
4.2	Precision and representation issues . . . . .	38
4.2.1	Precision, error and tolerance . . . . .	38
4.2.2	Representation of numbers . . . . .	39
4.2.3	Decimal issues . . . . .	40
4.2.4	Correctness guarantees . . . . .	41
4.2.5	Double precision emulation . . . . .	42
4.3	Conversion and printing . . . . .	43
4.3.1	convert() . . . . .	43
4.3.2	nstr() . . . . .	43
4.4	Rounding . . . . .	44
4.4.1	Next lower or equal integer: Floor( $x$ ) . . . . .	44

4.4.2	Next integer, rounded toward zero: $\text{Trunc}(x)$ . . . . .	45
4.4.3	Fractional Part . . . . .	45
4.5	Components of Real and Complex Numbers . . . . .	47
4.5.1	Number generated from Significand and Exponent: $\text{Ldexp}(x, y)$ . . . . .	47
4.5.2	Significand and Exponent: $\text{Frexp}(x)$ . . . . .	47
4.5.3	Building a Complex Number from Real Components . . . . .	48
4.5.4	Representations of Complex Numbers . . . . .	48
4.5.5	Real Component . . . . .	48
4.5.6	Imaginary Component . . . . .	49
4.5.7	Absolute Value . . . . .	49
4.5.8	Argument . . . . .	50
4.5.9	Sign . . . . .	50
4.5.10	Conjugate . . . . .	51
4.6	Arithmetic operations . . . . .	52
4.6.1	Addition and Sum . . . . .	52
4.6.2	Sums and Series . . . . .	53
4.6.3	Substraction . . . . .	54
4.6.4	Negation . . . . .	55
4.6.5	Multiplication . . . . .	56
4.6.6	Products . . . . .	57
4.6.7	Multiplication by multiples of 2 (LSH) . . . . .	58
4.6.8	Division . . . . .	58
4.6.9	Division by multiples of 2 (RSH) . . . . .	60
4.6.10	Modulo . . . . .	60
4.6.11	Power . . . . .	61
4.7	Logical Operators . . . . .	62
4.7.1	Bitwise AND . . . . .	62
4.7.2	Bitwise Inclusive OR . . . . .	62
4.7.3	Bitwise Exclusive OR . . . . .	62
4.8	Comparison Operators and Sorting . . . . .	63
4.8.1	Equal . . . . .	63
4.8.2	Greater or equal . . . . .	63
4.8.3	Greater than . . . . .	63
4.8.4	Less or equal . . . . .	63
4.8.5	Less than . . . . .	64
4.8.6	Not equal . . . . .	64
4.8.7	Tolerances and approximate comparisons . . . . .	64
4.9	Properties of numbers . . . . .	66
4.9.1	Testing for special values . . . . .	66
4.9.2	Testing for integers . . . . .	68
4.9.3	Approximating magnitude and precision . . . . .	68
4.10	Number generation . . . . .	70
4.10.1	Random numbers . . . . .	70
4.10.2	Fractions . . . . .	70
4.10.3	Ranges . . . . .	70

4.11	Matrices	72
4.11.1	Basic methods	72
4.11.2	Vectors	74
4.11.3	Other	74
4.11.4	Transposition	75
4.11.5	Matrix Properties	75
4.11.6	Addition	75
4.11.7	Multiplication	77
4.11.8	Python-specific convenience functions	77
<b>5</b>	<b>Elementary Functions</b>	<b>82</b>
5.1	Constants	82
5.1.1	Mathematical constants	82
5.1.2	Special values	83
5.2	Exponential and Logarithmic Functions	85
5.2.1	Exponential Function $e^z = \exp(z)$	85
5.2.2	Exponential Function $10^z = \exp_{10}(z)$	88
5.2.3	Exponential Function $2^z = \exp_2(z)$	89
5.2.4	Natural logarithm $\ln(x) = \log_e(x)$	90
5.2.5	Common (decadic) logarithm $\log_{10}(z)$	92
5.2.6	Binary logarithm $\log_2(z)$	92
5.2.7	Auxiliary Function $\ln(1 + x)$	92
5.3	Roots and Power Functions	93
5.3.1	Square: $z^2$	93
5.3.2	Power Function	94
5.3.3	Square Root: $\sqrt{z}$	96
5.3.4	Auxiliary Function $\sqrt{x^2 + y^2}$	97
5.3.5	Cube Root: $\sqrt[3]{x}, n = 2, 3, \dots$	97
5.3.6	Nth Root: $\sqrt[n]{z}, n = 2, 3, \dots$	98
5.4	Trigonometric Functions	102
5.4.1	Trigonometric functions: overview	102
5.4.2	Conversion between Degrees and radians	102
5.4.3	Sine: $\sin(z)$	103
5.4.4	Cosine: $\cos(z)$	105
5.4.5	Tangent: $\tan(z)$	107
5.4.6	Secant: $\sec(z) = 1 / \cos(z)$	109
5.4.7	Cosecant: $\csc(z) = 1 / \sin(z)$	111
5.4.8	Cotangent: $\cot(z) = 1 / \tan(z)$	113
5.4.9	Sinc function	115
5.4.10	Trigonometric functions with modified argument	115
5.5	Hyperbolic Functions	117
5.5.1	Hyperbolic Sine: $\sinh(z)$	117
5.5.2	Hyperbolic Cosine: $\cosh(z)$	119
5.5.3	Hyperbolic Tangent: $\tanh(z)$	121
5.5.4	Hyperbolic Secant: $\operatorname{sech}(x) = 1 / \cosh(z)$	123

5.5.5	Hyperbolic Cosecant: $\text{csch}(x) = 1/\sinh(z)$	124
5.5.6	Hyperbolic Cotangent: $\coth(x) = 1/\tanh(z)$	125
5.6	Inverse Trigonometric Functions	126
5.6.1	Arccsine: $\text{asin}(z)$	126
5.6.2	Arccosine: $\text{acos}(z)$	128
5.6.3	Arctangent: $\text{atan}(z)$	130
5.6.4	Arc-tangent, version with 2 arguments: $\text{atan2}(x, y)$	132
5.6.5	Arccotangent: $\text{acot}(z)$	133
5.6.6	$\text{asec}(x)$	133
5.6.7	$\text{acsc}(x)$	133
5.7	Inverse Hyperbolic Functions	134
5.7.1	Inverse Hyperbolic Sine: $\text{asinh}(z)$	134
5.7.2	Inverse Hyperbolic Cosine: $\text{acosh}(z)$	135
5.7.3	Inverse Hyperbolic Tangent: $\text{atanh}(z)$	136
5.7.4	Inverse Hyperbolic Cotangent: $\text{acoth}(z)$	137
5.7.5	$\text{asech}(x)$	137
5.7.6	$\text{acsch}(x)$	137
6	<b>Linear Algebra</b>	138
6.1	Norms	138
6.2	Decompositions	140
6.3	Linear Equations	142
6.4	Matrix Factorization	144
<b>III</b>	<b>Special Functions</b>	146
<b>7</b>	<b>Factorials and gamma functions</b>	147
7.1	Factorials	147
7.1.1	Factorial	147
7.1.2	Double factorial	148
7.2	Binomial coefficient	150
7.3	Pochhammer symbol, Rising and falling factorials	151
7.3.1	Relative Pochhammer symbol	151
7.3.2	Rising factorial	151
7.3.3	Falling factorial	152
7.4	Super- and hyperfactorials	153
7.4.1	Superfactorial	153
7.4.2	Hyperfactorial	154
7.4.3	Barnes G-function	155
7.5	Gamma functions	156
7.5.1	Gamma function	156

7.5.2	Reciprocal of the gamma function . . . . .	157
7.5.3	The product / quotient of gamma functions . . . . .	157
7.5.4	The log-gamma function . . . . .	158
7.5.5	Generalized incomplete gamma function . . . . .	159
7.5.6	Derivative of the normalised incomplete gamma function . . . . .	160
7.5.7	Normalised incomplete gamma functions . . . . .	160
7.5.8	Non-Normalised incomplete gamma functions . . . . .	161
7.5.9	Tricomi's entire incomplete gamma function . . . . .	161
7.5.10	Inverse normalised incomplete gamma functions . . . . .	162
7.6	Polygamma functions and harmonic numbers . . . . .	163
7.6.1	Polygamma function . . . . .	163
7.6.2	Digamma function . . . . .	164
7.6.3	Harmonic numbers . . . . .	164
7.7	Beta Functions . . . . .	165
7.7.1	Beta function $B(a, b)$ . . . . .	165
7.7.2	Beta function . . . . .	165
7.7.3	Logarithm of $B(a, b)$ . . . . .	165
7.7.4	Generalized incomplete beta function . . . . .	165
7.7.5	Non-Normalised incomplete beta functions . . . . .	166
7.7.6	Normalised incomplete beta functions . . . . .	167
<b>8</b>	<b>Exponential integrals and error functions</b> . . . . .	<b>168</b>
8.1	Exponential integrals . . . . .	168
8.1.1	Exponential integral $Ei$ . . . . .	168
8.1.2	Exponential integral $E1$ . . . . .	169
8.1.3	Generalized exponential integral $E_n$ . . . . .	169
8.1.4	Generalized Exponential Integrals $E_p$ . . . . .	170
8.2	Logarithmic integral . . . . .	171
8.2.1	logarithmic integral $li$ . . . . .	171
8.3	Trigonometric integrals . . . . .	172
8.3.1	cosine integral $ci$ . . . . .	172
8.3.2	sine integral $si$ . . . . .	172
8.4	Hyperbolic integrals . . . . .	174
8.4.1	hyperbolic cosine integral $chi$ . . . . .	174
8.4.2	hyperbolic sine integral $shi$ . . . . .	174
8.5	Error functions . . . . .	175
8.5.1	Error Function . . . . .	175
8.5.2	Complementary Error Function . . . . .	176
8.5.3	Imaginary Error Function . . . . .	176
8.5.4	Inverse Error Function . . . . .	177
8.6	The normal distribution . . . . .	179
8.6.1	The normal probability density function . . . . .	179
8.6.2	The normal cumulative distribution function . . . . .	179

8.7	Fresnel integrals . . . . .	180
8.7.1	Fresnel sine integral . . . . .	180
8.7.2	Fresnel cosine integral . . . . .	180
8.8	Other Special Functions . . . . .	182
8.8.1	Lambert W function . . . . .	182
8.8.2	Arithmetic-geometric mean . . . . .	183
<b>9</b>	<b>Bessel functions and related functions</b>	<b>184</b>
9.1	Bessel functions . . . . .	184
9.1.1	Exponentially scaled Bessel function $I_{\nu,e}(x)$ . . . . .	184
9.1.2	Exponentially scaled Bessel function $K_{\nu,e}(x)$ . . . . .	184
9.1.3	Bessel function of the first kind . . . . .	184
9.1.4	Bessel function of the second kind . . . . .	186
9.1.5	Modified Bessel function of the first kind . . . . .	187
9.1.6	Modified Bessel function of the second kind . . . . .	188
9.2	Bessel function zeros . . . . .	190
9.2.1	Zeros of the Bessel function of the first kind . . . . .	190
9.2.2	Zeros of the Bessel function of the second kind . . . . .	191
9.3	Hankel functions . . . . .	193
9.3.1	Hankel function of the first kind . . . . .	193
9.3.2	Hankel function of the second kind . . . . .	193
9.4	Kelvin functions . . . . .	194
9.4.1	Kelvin function ber . . . . .	194
9.4.2	Kelvin function bei . . . . .	194
9.4.3	Kelvin function ker . . . . .	194
9.4.4	Kelvin function kei . . . . .	195
9.5	Struve Functions . . . . .	196
9.5.1	Struve function H . . . . .	196
9.5.2	Modified Struve function L . . . . .	197
9.6	Anger-Weber functions . . . . .	198
9.6.1	Anger function J . . . . .	198
9.6.2	Weber function E . . . . .	198
9.7	Lommel functions . . . . .	200
9.7.1	First Lommel function s . . . . .	200
9.7.2	Second Lommel function S . . . . .	200
9.8	Airy and Scorer functions . . . . .	202
9.8.1	Airy function Ai . . . . .	202
9.8.2	Airy function Bi . . . . .	203
9.8.3	Zeros of the Airy function Ai . . . . .	205
9.8.4	Zeros of the Airy function Bi . . . . .	205
9.8.5	Scorer function Gi . . . . .	206
9.8.6	Scorer function Hi . . . . .	206
9.9	Coulomb wave functions . . . . .	208

9.9.1	Regular Coulomb wave function . . . . .	208
9.9.2	Irregular Coulomb wave function . . . . .	209
9.9.3	Normalizing Gamow constant . . . . .	209
9.10	Parabolic cylinder functions . . . . .	211
9.10.1	Parabolic cylinder function D . . . . .	211
9.10.2	Parabolic cylinder function U . . . . .	211
9.10.3	Parabolic cylinder function V . . . . .	212
9.10.4	Parabolic cylinder function W . . . . .	213
<b>10</b>	<b>Orthogonal polynomials</b>	<b>214</b>
10.1	Legendre functions . . . . .	214
10.1.1	Legendre polynomial . . . . .	214
10.1.2	Associated Legendre function of the first kind . . . . .	215
10.1.3	Associated Legendre function of the second kind . . . . .	216
10.1.4	Spherical harmonics . . . . .	217
10.2	Chebyshev polynomials . . . . .	219
10.2.1	Chebyshev polynomial of the first kind . . . . .	219
10.2.2	Chebyshev polynomial of the second kind . . . . .	219
10.3	Jacobi and Gegenbauer polynomials . . . . .	221
10.3.1	Jacobi polynomial . . . . .	221
10.3.2	Zernike Radial Polynomials . . . . .	222
10.3.3	Gegenbauer polynomial . . . . .	222
10.4	Hermite and Laguerre polynomials . . . . .	224
10.4.1	Hermite polynomials . . . . .	224
10.4.2	Laguerre polynomials . . . . .	225
10.4.3	Laguerre Polynomials . . . . .	226
10.4.4	Associated Laguerre Polynomials . . . . .	226
<b>11</b>	<b>Hypergeometric functions</b>	<b>227</b>
11.1	Confluent Hypergeometric Limit Function . . . . .	228
11.1.1	Confluent Hypergeometric Limit Function . . . . .	228
11.1.2	Regularized Confluent Hypergeometric Limit Function . . . . .	229
11.2	Kummer's Confluent Hypergeometric Functions and related functions . . . . .	230
11.2.1	Kummer's Confluent Hypergeometric Function of the first kind . . . . .	230
11.2.2	Kummer's Regularized Confluent Hypergeometric Function . . . . .	231
11.2.3	Kummer's Confluent Hypergeometric Function of the second kind . . . . .	232
11.2.4	Hypergeometric Function 2F0 . . . . .	232
11.3	Whittaker functions M and W . . . . .	234
11.3.1	Whittaker function M . . . . .	234
11.3.2	Whittaker function W . . . . .	234
11.4	Gauss Hypergeometric Function . . . . .	236
11.4.1	Gauss Hypergeometric Function . . . . .	236
11.4.2	Gauss Regularized Hypergeometric Function . . . . .	238

11.5	Additional Hypergeometric Functions . . . . .	239
11.5.1	Hypergeometric Function 1F2 . . . . .	239
11.5.2	Hypergeometric Function 2F2 . . . . .	239
11.5.3	Hypergeometric Function 2F3 . . . . .	240
11.5.4	Hypergeometric Function 3F2 . . . . .	240
11.6	Generalized hypergeometric functions . . . . .	242
11.6.1	Generalized hypergeometric function pFq . . . . .	242
11.6.2	Weighted combination of hypergeometric functions . . . . .	243
11.7	Meijer G-function . . . . .	245
11.8	Bilateral hypergeometric series . . . . .	248
11.9	Hypergeometric functions of two variables . . . . .	249
11.9.1	Generalized 2D hypergeometric series . . . . .	249
11.9.2	Appell F1 hypergeometric function . . . . .	250
11.9.3	Appell F2 hypergeometric function . . . . .	251
11.9.4	Appell F3 hypergeometric function . . . . .	251
11.9.5	Appell F4 hypergeometric function . . . . .	252
<b>12</b>	<b>Elliptic functions</b> . . . . .	<b>254</b>
12.1	Elliptic arguments . . . . .	254
12.2	Legendre elliptic integrals . . . . .	257
12.2.1	Complete elliptic integral of the first kind . . . . .	257
12.2.2	Incomplete elliptic integral of the first kind . . . . .	257
12.2.3	Complete elliptic integral of the second kind . . . . .	259
12.2.4	Incomplete elliptic integral of the second kind . . . . .	259
12.2.5	Complete elliptic integral of the third kind . . . . .	260
12.2.6	Incomplete elliptic integral of the third kind . . . . .	261
12.3	Carlson symmetric elliptic integrals . . . . .	263
12.3.1	Symmetric elliptic integral of the first kind, RF . . . . .	263
12.3.2	Degenerate Carlson symmetric elliptic integral of the first kind, RC . . . . .	264
12.3.3	Symmetric elliptic integral of the third kind, RJ . . . . .	265
12.3.4	Symmetric elliptic integral of the second kind, RD . . . . .	266
12.3.5	Completely symmetric elliptic integral of the second kind, RG . . . . .	266
12.4	Jacobi theta functions . . . . .	268
12.5	Jacobi elliptic functions . . . . .	270
12.6	Klein j-invariant . . . . .	272
<b>13</b>	<b>Zeta functions, L-series and polylogarithms</b> . . . . .	<b>273</b>
13.1	Riemann and Hurwitz zeta functions . . . . .	273
13.2	Dirichlet L-series . . . . .	275
13.2.1	Dirichlet eta function . . . . .	275
13.2.2	Dirichlet $\eta(s) - 1$ . . . . .	275

13.2.3	Dirichlet Beta Function . . . . .	276
13.2.4	Dirichlet Lambda Function . . . . .	276
13.2.5	Dirichlet L-function . . . . .	276
13.3	Stieltjes constants . . . . .	278
13.4	Zeta function zeros . . . . .	279
13.5	Riemann-Siegel Z function and related functions . . . . .	281
13.5.1	Riemann-Siegel Z . . . . .	281
13.5.2	Riemann-Siegel theta function . . . . .	281
13.5.3	Gram point (Riemann-Siegel Z function) . . . . .	282
13.5.4	Backlunds function . . . . .	282
13.6	Lerch transcendent and related functions . . . . .	284
13.6.1	Fermi-Dirac integrals of integer order . . . . .	285
13.6.2	Fermi-Dirac integral $F_{-1/2}(x)$ . . . . .	285
13.6.3	Fermi-Dirac integral $F_{1/2}(x)$ . . . . .	285
13.6.4	Fermi-Dirac integral $F_{3/2}(x)$ . . . . .	285
13.6.5	Legendre Chi-Function . . . . .	286
13.6.6	Inverse Tangent Integral . . . . .	286
13.7	Polylogarithms and Clausen functions . . . . .	287
13.7.1	Polylogarithm . . . . .	287
13.7.2	Dilogarithm Function . . . . .	288
13.7.3	Debye Functions . . . . .	288
13.7.4	Clausen sine function . . . . .	288
13.7.5	Clausen cosine function . . . . .	289
13.7.6	Polyexponential function . . . . .	290
13.8	Zeta function variants . . . . .	291
13.8.1	Prime zeta function . . . . .	291
13.8.2	Secondary zeta function . . . . .	291
<b>14</b>	<b>Number-theoretical, combinatorial and integer functions</b>	<b>293</b>
14.1	Fibonacci numbers . . . . .	293
14.2	Bernoulli numbers and polynomials . . . . .	295
14.2.1	Bernoulli numbers . . . . .	295
14.2.2	Bernoulli polynomials . . . . .	297
14.3	Euler numbers and polynomials . . . . .	299
14.3.1	Euler numbers . . . . .	299
14.3.2	Euler polynomials . . . . .	299
14.4	Bell numbers and polynomials . . . . .	301
14.5	Stirling numbers . . . . .	302
14.5.1	Stirling number of the first kind . . . . .	302
14.5.2	Stirling number of the second kind . . . . .	302
14.6	Prime counting functions . . . . .	304
14.6.1	Exact prime counting function . . . . .	304

14.6.2	Prime counting function interval . . . . .	304
14.6.3	Riemann R function . . . . .	305
14.7	Miscellaneous functions . . . . .	307
14.7.1	Cyclotomic polynomials . . . . .	307
14.7.2	von Mangoldt function . . . . .	308
<b>15</b>	<b>q-functions</b>	<b>309</b>
15.1	q-Pochhammer symbol . . . . .	309
15.2	q-gamma and factorial . . . . .	310
15.2.1	q-gamma . . . . .	310
15.2.2	q-factorial . . . . .	310
15.3	Hypergeometric q-series . . . . .	312
<b>16</b>	<b>Matrix functions</b>	<b>313</b>
16.1	Matrix exponential . . . . .	313
16.2	Matrix cosine . . . . .	315
16.3	Matrix sine . . . . .	316
16.4	Matrix square root . . . . .	317
16.5	Matrix logarithm . . . . .	319
16.6	Matrix power . . . . .	321
<b>17</b>	<b>Eigensystems and related Decompositions</b>	<b>323</b>
17.1	Singular value decomposition . . . . .	323
17.2	The Schur decomposition . . . . .	324
17.3	The eigenvalue problem . . . . .	325
17.4	The symmetric eigenvalue problem . . . . .	326
<b>IV</b>	<b>GMP and related libraries</b>	<b>327</b>
<b>18</b>	<b>GMP, MPD, and related libraries: an overview</b>	<b>328</b>
18.1	Integer Types and Fractions . . . . .	328
18.2	FloatingPoint Types . . . . .	328
18.2.1	Fixed Single Precision . . . . .	328
18.2.2	Fixed Double Precision . . . . .	328
18.3	Arithmetic Operators . . . . .	329
18.3.1	Addition . . . . .	329
18.3.2	Subtraction . . . . .	329

18.3.3	Multiplication (Scalars, Vectors and Matrices) . . . . .	329
18.3.4	Scalar Division . . . . .	330
18.3.5	Modulo . . . . .	330
18.3.6	Power . . . . .	330
18.4	Comparison Operators and Sorting . . . . .	330
18.4.1	Equal . . . . .	330
18.4.2	Greater or equal . . . . .	331
18.4.3	Greater than . . . . .	331
18.4.4	Less or equal . . . . .	331
18.4.5	Less than . . . . .	331
18.4.6	Not equal . . . . .	332
18.4.7	IsApproximate . . . . .	332
18.4.8	IsSmall . . . . .	332
18.5	Vectors, Matrices and Tables . . . . .	332
18.5.1	Dimension (Vectors and Matrices) . . . . .	332
18.5.2	Precision . . . . .	333
18.5.3	Item . . . . .	333
18.5.4	Row . . . . .	333
18.5.5	Column . . . . .	333
18.5.6	Matrix . . . . .	334
18.5.7	Sorting . . . . .	334
18.5.8	Table . . . . .	334
18.5.9	List of Tables . . . . .	335
<b>19 FMPZ</b>		<b>336</b>
19.1	Multiprecision Rational Numbers (GMP: MPQ) . . . . .	336
19.2	Arithmetic Operators . . . . .	337
19.2.1	Unary Minus . . . . .	337
19.2.2	Addition . . . . .	337
19.2.3	Subtraction . . . . .	337
19.2.4	Multiplication . . . . .	338
19.2.5	Fused-Multiply-Add fma . . . . .	338
19.2.6	Fused-Multiply-Subtract fms . . . . .	338
19.2.7	Multiplication by multiples of 2 (LSH) . . . . .	339
19.2.8	Division by multiples of 2 (RSH) . . . . .	339
19.2.9	Exact Division . . . . .	339
19.2.10	Modulo Division . . . . .	339
19.3	Divisions, forming quotients and/or remainder . . . . .	340
19.3.1	Quotient only, rounded up . . . . .	340
19.3.2	Remainder only (Quotient rounded up) . . . . .	341
19.3.3	Quotient and Remainder, Quotient rounded up . . . . .	341
19.3.4	Quotient only, rounded down . . . . .	341
19.3.5	Remainder only (Quotient rounded down) . . . . .	342
19.3.6	Quotient and Remainder, Quotient rounded down . . . . .	342
19.3.7	Quotient only, Quotient truncated . . . . .	342

19.3.8	Remainder only (Quotient truncated) . . . . .	343
19.3.9	Quotient and Remainder, Quotient truncated . . . . .	343
19.4	Logical Operators . . . . .	344
19.4.1	Bitwise AND . . . . .	344
19.4.2	Bitwise Inclusive OR . . . . .	344
19.4.3	Bitwise Exclusive OR . . . . .	344
19.5	Bit-Oriented Functions . . . . .	344
19.5.1	Complement . . . . .	344
19.5.2	Hamming Distance . . . . .	344
19.5.3	Testing , setting, and clearing a Bit . . . . .	345
19.5.4	Scanning for 0 or 1 . . . . .	345
19.5.5	Population Count . . . . .	346
19.6	Sign, Powers and Roots . . . . .	346
19.6.1	Sign . . . . .	346
19.6.2	Absolute value . . . . .	346
19.6.3	Power Function: $n^k$ ; $n, k \in \mathbb{Z}$ . . . . .	347
19.6.4	Power Function modulo m: $n^k \bmod m$ ; $m, n, k \in \mathbb{Z}$ . . . . .	347
19.6.5	Truncated integer part of the square root: $\lfloor \sqrt{n} \rfloor$ . . . . .	347
19.6.6	Truncated integer part of the square root: $\lfloor \sqrt{m} \rfloor$ , with remainder . . . . .	347
19.6.7	Truncated integer part of the nth root: $\lfloor \sqrt[n]{m} \rfloor$ . . . . .	348
19.6.8	Truncated integer part of the nth root: $\lfloor \sqrt[n]{m} \rfloor$ , with remainder . . . . .	348
19.7	Numbertheoretic Functions . . . . .	348
19.7.1	Factorial . . . . .	348
19.7.2	Binomial Coefficient, Combinations . . . . .	348
19.7.3	Next Prime . . . . .	349
19.7.4	Greatest Common Divisor (GCD) . . . . .	349
19.7.5	Greatest Common Divisor, Extended . . . . .	349
19.7.6	Least Common Multiple (LCM) . . . . .	349
19.7.7	Inverse Modulus . . . . .	350
19.7.8	Remove Factor . . . . .	350
19.7.9	Legendre Symbol . . . . .	350
19.7.10	Jacobi Symbol . . . . .	350
19.7.11	Kronecker Symbol . . . . .	351
19.7.12	Fibonacci Numbers . . . . .	351
19.7.13	Lucas Numbers . . . . .	351
19.8	Additional Numbertheoretic Functions . . . . .	351
19.8.1	Pseudoprimes . . . . .	351
19.8.2	Lucas Sequences . . . . .	354
19.9	Random Numbers . . . . .	356
19.9.1	intUrandomb . . . . .	356
19.9.2	intUrandomm . . . . .	356
19.9.3	intRRandomb . . . . .	356
19.10	Information Functions for Integers . . . . .	356
19.10.1	Congruence: IsCongruent( $n, c, d$ ) . . . . .	356
19.10.2	Congruence 2n: IsCongruent2exp( $n, c, b$ ) . . . . .	357

19.10.3	Primality Testing: <code>IsProbablyPrime(n, reps)</code>	357
19.10.4	Divisibility: <code>IsDivisible(n, d)</code>	357
19.10.5	Divisibility by (2 pow b): <code>IsDivisible2exp(n, b)</code>	358
19.10.6	Perfect Power: <code>IsPerfectPower(n)</code>	358
19.10.7	Perfect Square: <code>IsPerfectSquare(n)</code>	358
<b>20</b>	<b>FMPQ</b>	<b>359</b>
<b>21</b>	<b>ARB</b>	<b>360</b>
21.1	Multiprecision Ball Arithmetic (ARB)	360
21.2	Information Functions for Intervals	361
21.2.1	<code>IsEmpty</code>	361
21.2.2	<code>IsInside</code>	361
21.2.3	<code>IsStrictlyInside</code>	361
21.2.4	<code>IsStrictlyNeg</code>	362
21.2.5	<code>IsStrictlyPos</code>	362
<b>22</b>	<b>ACB</b>	<b>363</b>
<b>V</b>	<b>Eigen and related libraries</b>	<b>364</b>
<b>23</b>	<b>BLAS Support (based on Eigen)</b>	<b>365</b>
23.1	BLAS Level 1 Support and related Functions	366
23.1.1	<code>Vector-Vector Product</code>	366
23.1.2	<code>Euclidian Norm</code>	366
23.1.3	<code>Absolute Sum</code>	367
23.1.4	<code>Addition</code>	367
23.2	BLAS Level 2 Support	368
23.2.1	<code>Matrix-Vector Product and Sum (General Matrix)</code>	368
23.2.2	<code>Matrix-Vector Product (Triangular Matrix)</code>	369
23.2.3	<code>Inverse Matrix-Vector Product (Triangular Matrix)</code>	370
23.2.4	<code>Matrix-Vector Product and Sum (Symmetric/Hermitian Matrix)</code>	371
23.2.5	<code>Rank-1 update (General Matrix)</code>	372
23.2.6	<code>Rank-1 update (Symmetric/Hermitian Matrix)</code>	373
23.2.7	<code>Rank-2 update (Symmetric/Hermitian Matrix)</code>	374
23.3	BLAS Level 3 Support	374
23.3.1	<code>Matrix-Matrix-Product and Sum (General Matrix A)</code>	374
23.3.2	<code>Matrix-Matrix-Product and Sum (Symmetric/Hermitian Matrix A)</code>	377
23.3.3	<code>Matrix-Matrix-Product (Triangular Matrix A)</code>	379
23.3.4	<code>Inverse Matrix-Matrix-Product (Triangular Matrix A)</code>	381
23.3.5	<code>Rank-k update (Symmetric/Hermitian Matrix C)</code>	383
23.3.6	<code>Rank-2k update (Symmetric/Hermitian Matrix C)</code>	385

<b>24 Linear Solvers (based on Eigen)</b>	<b>387</b>
24.1 Cholesky Decomposition without Pivoting . . . . .	387
24.1.1 Decomposition . . . . .	387
24.1.2 Linear Solver . . . . .	388
24.1.3 Matrix Inversion . . . . .	389
24.1.4 Determinant . . . . .	389
24.1.5 Example . . . . .	390
24.2 Cholesky Decomposition with Pivoting . . . . .	391
24.2.1 Decomposition . . . . .	391
24.2.2 Linear Solver . . . . .	392
24.2.3 Matrix Inversion . . . . .	393
24.2.4 Determinant . . . . .	393
24.2.5 Example . . . . .	394
24.3 LU Decomposition with partial Pivoting . . . . .	395
24.3.1 Decomposition . . . . .	395
24.3.2 Linear Solver . . . . .	396
24.3.3 Matrix Inversion . . . . .	396
24.3.4 Determinant . . . . .	397
24.3.5 Example . . . . .	397
24.4 LU Decomposition with full Pivoting . . . . .	399
24.4.1 Decomposition . . . . .	399
24.4.2 Linear Solver . . . . .	403
24.4.3 Matrix Inversion . . . . .	403
24.4.4 Determinant . . . . .	404
24.5 QR Decomposition without Pivoting . . . . .	405
24.5.1 Decomposition . . . . .	405
24.5.2 Linear Solver . . . . .	407
24.5.3 Matrix Inversion . . . . .	407
24.5.4 Determinant . . . . .	408
24.5.5 Example . . . . .	408
24.6 QR Decomposition with column Pivoting . . . . .	409
24.6.1 Decomposition . . . . .	409
24.6.2 Linear Solver . . . . .	412
24.6.3 Matrix Inversion . . . . .	412
24.6.4 Determinant . . . . .	413
24.6.5 Example . . . . .	413
24.7 QR Decomposition with full Pivoting . . . . .	414
24.7.1 Decomposition . . . . .	414
24.7.2 Linear Solver . . . . .	417
24.7.3 Matrix Inversion . . . . .	417
24.7.4 Determinant . . . . .	418
24.7.5 Example . . . . .	418
24.8 Singular Value Decomposition . . . . .	419
24.8.1 Decomposition . . . . .	420

24.8.2	Linear Solver . . . . .	421
24.8.3	Matrix Inversion . . . . .	421
24.8.4	Determinant . . . . .	422
24.8.5	Example . . . . .	422
24.9	Householder Transformations . . . . .	424
24.9.1	Overview . . . . .	424
24.9.2	Constructor . . . . .	424
24.9.3	Member Function Documentation . . . . .	426
<b>25</b>	<b>Eigenvalues, (based on Eigen)</b>	<b>428</b>
25.1	Symmetric/Hermitian Eigensystems . . . . .	428
25.1.1	Real Symmetric Matrices . . . . .	428
25.1.2	Complex Hermitian Matrices . . . . .	432
25.2	General (Nonsymmetric) Eigensystems . . . . .	435
25.2.1	Real Nonsymmetric Matrices . . . . .	435
25.2.2	Complex Nonsymmetric Matrices . . . . .	440
25.3	Generalized Eigensystems . . . . .	444
25.3.1	Real Generalized Symmetric-Definite Eigensystems . . . . .	445
25.3.2	Complex Hermitian Generalized Symmetric-Definite Eigensystems . . . . .	449
25.3.3	Real Generalized Nonsymmetric Eigensystem . . . . .	450
25.4	Decompositions . . . . .	453
25.4.1	Tridiagonalization . . . . .	453
25.4.2	Hessenberg Decomposition . . . . .	458
25.4.3	Real QZ Decomposition . . . . .	462
25.4.4	Real Schur Decomposition . . . . .	465
25.4.5	Complex Schur Decomposition . . . . .	468
25.5	Matrix Functions . . . . .	471
25.5.1	Matrix Square Root . . . . .	471
25.5.2	Matrix Exponential . . . . .	473
25.5.3	Matrix Logarithm . . . . .	474
25.5.4	Matrix raised to arbitrary real power . . . . .	476
25.5.5	Matrix General Function . . . . .	479
25.5.6	Matrix Sine . . . . .	480
25.5.7	Matrix Cosine . . . . .	482
25.5.8	Matrix Hyperbolic Sine . . . . .	482
25.5.9	Matrix Hyperbolic Cosine . . . . .	483
<b>26</b>	<b>Polynomials (based on Eigen)</b>	<b>485</b>
26.1	Polynomial Evaluation . . . . .	485
26.1.1	Polynomial Evaluation, Real Coefficients and Argument . . . . .	485
26.1.2	Polynomial Evaluation, Complex Coefficients and Argument . . . . .	485
26.1.3	Examples . . . . .	485
26.2	Quadratic Equations . . . . .	486

26.2.1	Quadratic Equation, Real Coefficients and Zeros . . . . .	486
26.2.2	Quadratic Equation, Complex Coefficients and Zeros . . . . .	486
26.3	Cubic Equations . . . . .	487
26.3.1	Cubic Equation, Real Coefficients and Zeros . . . . .	487
26.3.2	Cubic Equation, Complex Coefficients and Zeros . . . . .	487
26.4	Quartic Equations . . . . .	488
26.4.1	Quartic Equation, Real Coefficients and Zeros . . . . .	488
26.4.2	Quartic Equation, Complex Coefficients and Zeros . . . . .	488
26.5	General Polynomial Equations . . . . .	488
26.5.1	General Polynomial Equation, Real Coefficients and Zeros . . . . .	488
26.5.2	General Polynomial Equation, Complex Coefficients and Zeros . . . . .	489
<b>27</b>	<b>Fast Fourier Transform (based on Eigen)</b>	<b>491</b>
27.1	Discrete Fourier Transforms . . . . .	491
27.1.1	1d Complex Discrete Fourier Transform (DFT) . . . . .	491
27.1.2	1d Real-data DFT . . . . .	492
27.1.3	1d Real-even DFTs (DCTs) . . . . .	493
27.1.4	1d Real-odd DFTs (DSTs) . . . . .	494
<b>28</b>	<b>Minimization and Optimization: Procedures based on MINPACK</b>	<b>497</b>
28.1	Multidimensional Rootfinding: Powell Hybrid . . . . .	498
28.2	Nonlinear LeastSquares: Levenberg-Marquardt . . . . .	499
<b>29</b>	<b>Procedures based on NLOPT</b>	<b>500</b>
29.1	Overview . . . . .	500
29.2	Global optimization . . . . .	501
29.2.1	DIRECT and DIRECT-L . . . . .	501
29.2.2	Controlled Random Search (CRS) with local mutation . . . . .	502
29.2.3	MLSL (Multi-Level Single-Limkage) . . . . .	502
29.2.4	StoGO . . . . .	502
29.2.5	ISRES (Improved Stochastic Ranking Evolution Strategy) . . . . .	503
29.3	Local derivative-free optimization . . . . .	503
29.3.1	COBYLA (Constrained Optimization BY Linear Approximations) . . . . .	503
29.3.2	BOBYQA . . . . .	504
29.3.3	NEWUOA + bound constraints . . . . .	504
29.3.4	PRAxis (Principal AXIS) . . . . .	505
29.3.5	Nelder-Mead Simplex . . . . .	505
29.3.6	Sbplx (based on Subplex) . . . . .	506
29.4	Local gradient-based optimization . . . . .	506
29.4.1	MMA (Method of Moving Asymptotes) and CCSA . . . . .	506
29.4.2	SLSQP . . . . .	507
29.4.3	Low-storage BFGS . . . . .	508

29.4.4	Preconditioned truncated Newton . . . . .	508
29.4.5	Shifted limited-memory variable-metric . . . . .	508
29.5	NLOPT: Augmented Lagrangian algorithm . . . . .	509
29.5.1	Implementation . . . . .	509
<b>VI</b>	<b>Boost and related libraries</b>	<b>510</b>
<b>30</b>	<b>RandomNumbers</b>	<b>511</b>
30.1	Definitions . . . . .	511
30.1.1	Random Device . . . . .	511
30.1.2	Uniform Random Number Generator . . . . .	511
30.1.3	Pseudo-Random Number Generator . . . . .	512
30.2	The Random Number Generator Interface . . . . .	512
30.2.1	Sampling . . . . .	512
30.3	Random number generator algorithms . . . . .	512
30.3.1	Minimal Standard . . . . .	513
30.3.2	rand48 . . . . .	513
30.3.3	Ecuyer 1988 . . . . .	513
30.3.4	Knuth b . . . . .	513
30.3.5	Kreutzer 1986 . . . . .	513
30.3.6	Tauss 88 . . . . .	513
30.3.7	Hellekalek 1995 . . . . .	513
30.3.8	Mersenne-Twister 11213b . . . . .	514
30.3.9	Mersenne-Twister 19937 . . . . .	514
30.3.10	Mersenne-Twister 19937 64 . . . . .	514
30.3.11	Lagged Fibonacci Generators . . . . .	514
30.3.12	Ranlux Generators . . . . .	514
30.4	Random number distributions . . . . .	514
30.4.1	Uniform, small integer . . . . .	514
30.4.2	Uniform, integer . . . . .	515
30.4.3	Uniform, 01 . . . . .	515
30.4.4	Uniform, Real . . . . .	515
30.4.5	Discrete . . . . .	515
30.4.6	Piecewise constant . . . . .	515
30.4.7	Piecewise linear . . . . .	516
30.4.8	Triangle . . . . .	516
30.4.9	Uniform on Sphere . . . . .	516
<b>31</b>	<b>Special Functions (based on Boost)</b>	<b>517</b>
31.1	Gamma and Beta Functions . . . . .	517
31.1.1	Gamma function $\Gamma(x)$ . . . . .	517
31.1.2	Logarithm of $\Gamma(x)$ . . . . .	517
31.1.3	Auxiliary function $\Gamma(x)/\Gamma(x + \delta)$ . . . . .	518
31.1.4	Digamma function $\psi(x)$ . . . . .	518

31.1.5	Ratio of Gamma Functions . . . . .	518
31.1.6	Normalised incomplete gamma functions . . . . .	519
31.1.7	Non-Normalised incomplete gamma functions . . . . .	519
31.1.8	Inverse normalised incomplete gamma functions . . . . .	520
31.1.9	Derivative of the normalised incomplete gamma function . . . . .	521
31.2	Factorials and Binomial Coefficient . . . . .	521
31.2.1	Factorial . . . . .	521
31.2.2	Double Factorial . . . . .	521
31.2.3	Rising Factorial . . . . .	521
31.2.4	Falling Factorial . . . . .	522
31.2.5	Binomial coefficient . . . . .	522
31.3	Beta Functions . . . . .	522
31.3.1	Beta function $B(a, b)$ . . . . .	522
31.4	Error Function and Related Functions . . . . .	523
31.4.1	Error Function $\text{erf}$ . . . . .	523
31.4.2	Complementary Error Function . . . . .	523
31.4.3	Inverse Function of $\text{erf}$ . . . . .	523
31.4.4	Inverse Function of $\text{erfc}$ . . . . .	524
31.5	Polynomials . . . . .	525
31.5.1	Legendre Polynomials/Functions . . . . .	525
31.5.2	Associated Legendre Polynomials/Functions . . . . .	525
31.5.3	Legendre Functions of the Second Kind . . . . .	526
31.5.4	Laguerre Polynomials . . . . .	526
31.5.5	Associated Laguerre Polynomials . . . . .	527
31.5.6	Hermite Polynomials . . . . .	528
31.5.7	Spherical Harmonic Functions . . . . .	528
31.6	Bessel Functions of Real Order . . . . .	529
31.6.1	Bessel Function $J_\nu(x)$ . . . . .	529
31.6.2	Bessel Function $Y_\nu(x)$ . . . . .	529
31.7	Modified Bessel Functions of Real Order . . . . .	529
31.7.1	Bessel Function $I_\nu(x)$ . . . . .	529
31.7.2	Bessel Function $K_\nu(x)$ . . . . .	530
31.8	Spherical Bessel Functions . . . . .	530
31.8.1	Spherical Bessel function $j_n(x)$ . . . . .	530
31.8.2	Spherical Bessel function $y_n(x)$ . . . . .	530
31.9	Hankel Functions . . . . .	531
31.9.1	Hankel Function of the First Kind . . . . .	531
31.9.2	Hankel Function of the Second Kind . . . . .	531
31.9.3	Spherical Hankel Function of the First Kind . . . . .	531
31.9.4	Spherical Hankel Function of the Second Kind . . . . .	532
31.10	Airy Functions . . . . .	532
31.10.1	Airy Function $\text{Ai}(x)$ . . . . .	532
31.10.2	Airy Function $\text{Ai}'(x)$ . . . . .	532
31.10.3	Airy Function $\text{Bi}(x)$ . . . . .	533

31.10.4	Airy Function $\text{Bi}'(x)$	533
31.11	Carlson-style Elliptic Integrals	533
31.11.1	Degenerate elliptic integral $\text{RC}$	534
31.11.2	Integral of the 1st kind $\text{RF}$	534
31.11.3	Integral of the 2nd kind $\text{RD}$	534
31.11.4	Integral of the 3rd kind $\text{RJ}$	535
31.12	Legendre-style Elliptic Integrals	535
31.12.1	Complete elliptic integral of the 1st kind	535
31.12.2	Complete elliptic integral of the 2nd kind	535
31.12.3	Complete elliptic integral of the 3rd kind	536
31.12.4	Legendre elliptic integral of the 1st kind	536
31.12.5	Legendre elliptic integral of the 2nd kind	536
31.12.6	Legendre elliptic integral of the 3rd kind	537
31.13	Jacobi Elliptic Functions	537
31.13.1	Jacobi elliptic function $\text{sn}$	537
31.13.2	Jacobi elliptic function $\text{cn}$	537
31.13.3	Jacobi elliptic function $\text{dn}$	538
31.14	Zeta Functions	538
31.14.1	Riemann $\zeta(s)$ function	538
31.15	Exponential Integral and Related Integrals	538
31.15.1	Exponential Integral $\text{E1}$	538
31.15.2	Exponential Integral $\text{Ei}$	539
31.15.3	Exponential Integrals $\text{En}$	539
31.16	Basic Functions	539
31.16.1	Auxiliary Function $\ln(1+x)$	539
31.16.2	Auxiliary Function $e^x - 1$	540
31.16.3	Cube Root: $\sqrt[3]{x}$	540
31.16.4	Auxiliary Function $\sqrt{x+1} - 1$	540
31.16.5	Auxiliary Function $x^y - 1$	540
31.16.6	Auxiliary Function $\sqrt{x^2 + y^2}$	540
31.17	Sinus Cardinal Function and Hyperbolic Sinus Cardinal Functions	541
31.17.1	Hyperbolic Sinus Cardinal: $\text{Sinhc}_a(x)$	541
31.18	Inverse Hyperbolic Functions	541
31.18.1	Hyperbolic Arc-cosine: $\text{acosh}(x)$	541
31.18.2	Hyperbolic Arc-sine: $\text{asinh}(x)$	541
31.18.3	Hyperbolic Arc-tangent: $\text{atanh}(x)$	542
<b>32</b>	<b>Distribution Functions</b>	<b>543</b>
32.1	Introduction to Distribution Functions	543
32.1.1	Continuous Distribution Functions	543
32.1.2	Discrete Distribution Functions	543
32.1.3	Commonly Used Function Types	544
32.2	Beta-Distribution	549

32.2.1	Definition	549
32.2.2	Density and CDF	550
32.2.3	Quantiles	550
32.2.4	Properties	551
32.2.5	Random Numbers	552
32.3	Binomial Distribution	552
32.3.1	Density and CDF	552
32.3.2	Quantiles	553
32.3.3	Properties	553
32.3.4	Random Numbers	554
32.4	Chi-Square Distribution	555
32.4.1	Definition	555
32.4.2	Density and CDF	555
32.4.3	Quantiles	556
32.4.4	Properties	556
32.4.5	Random Numbers	556
32.4.6	Wishart Matrix	557
32.5	Exponential Distribution	557
32.5.1	Density and CDF	557
32.5.2	Quantiles	557
32.5.3	Properties	558
32.5.4	Random Numbers	558
32.6	Fisher's F-Distribution	559
32.6.1	Definition	559
32.6.2	Density and CDF	559
32.6.3	Quantiles	559
32.6.4	Properties	560
32.6.5	Random Numbers	560
32.7	Gamma (and Erlang) Distribution	560
32.7.1	Density and CDF	560
32.7.2	Quantiles	561
32.7.3	Properties	561
32.7.4	Random Numbers	562
32.8	Hypergeometric Distribution	563
32.8.1	Definition	563
32.8.2	Density and CDF	563
32.8.3	Quantiles	564
32.8.4	Sample Size	564
32.8.5	Properties	564
32.8.6	Random Numbers	565
32.9	Lognormal Distribution	565
32.9.1	Definition	565
32.9.2	Density and CDF	566
32.9.3	Quantiles	566
32.9.4	Properties	566

32.9.5	Random Numbers . . . . .	567
32.10	Negative Binomial Distribution . . . . .	567
32.10.1	Density and CDF . . . . .	568
32.10.2	Quantiles . . . . .	568
32.10.3	Properties . . . . .	568
32.10.4	Random Numbers . . . . .	569
32.11	Normal Distribution . . . . .	570
32.11.1	Definition . . . . .	570
32.11.2	Density and CDF . . . . .	570
32.11.3	Quantiles . . . . .	571
32.11.4	Properties . . . . .	571
32.11.5	Random Numbers . . . . .	572
32.12	Poisson Distribution . . . . .	572
32.12.1	Definition . . . . .	572
32.12.2	Density and CDF . . . . .	572
32.12.3	Quantiles . . . . .	573
32.12.4	Properties . . . . .	573
32.12.5	Random Numbers . . . . .	574
32.13	Student's t-Distribution . . . . .	574
32.13.1	Definition . . . . .	574
32.13.2	Density and CDF . . . . .	574
32.13.3	Quantiles . . . . .	575
32.13.4	Properties . . . . .	575
32.13.5	Random Numbers . . . . .	576
32.13.6	Behrens-Fisher Problem . . . . .	576
32.14	Weibull Distribution . . . . .	576
32.14.1	Density and CDF . . . . .	576
32.14.2	Quantiles . . . . .	577
32.14.3	Properties . . . . .	577
32.14.4	Random Numbers . . . . .	578
32.15	Bernoulli Distribution . . . . .	578
32.15.1	Density and CDF . . . . .	578
32.15.2	Quantiles . . . . .	579
32.15.3	Properties . . . . .	579
32.15.4	Random Numbers . . . . .	580
32.16	Cauchy Distribution . . . . .	580
32.16.1	Density and CDF . . . . .	580
32.16.2	Quantiles . . . . .	580
32.16.3	Properties . . . . .	581
32.16.4	Random Numbers . . . . .	581
32.17	Extreme Value (or Gumbel) Distribution . . . . .	582
32.17.1	Density and CDF . . . . .	582
32.17.2	Quantiles . . . . .	582
32.17.3	Properties . . . . .	583

32.17.4	Random Numbers . . . . .	583
32.18	Geometric Distribution . . . . .	583
32.18.1	Density and CDF . . . . .	583
32.18.2	Quantiles . . . . .	584
32.18.3	Properties . . . . .	584
32.18.4	Random Numbers . . . . .	584
32.19	Inverse Chi Squared Distribution . . . . .	585
32.19.1	Definition . . . . .	585
32.19.2	Density and CDF . . . . .	585
32.19.3	Quantiles . . . . .	586
32.19.4	Properties . . . . .	586
32.19.5	Random Numbers . . . . .	586
32.20	Inverse Gamma Distribution . . . . .	587
32.20.1	Definition . . . . .	587
32.20.2	Density and CDF . . . . .	587
32.20.3	Quantiles . . . . .	588
32.20.4	Properties . . . . .	588
32.20.5	Random Numbers . . . . .	589
32.21	Inverse Gaussian (or Wald) Distribution . . . . .	589
32.21.1	Definition . . . . .	589
32.21.2	Density and CDF . . . . .	589
32.21.3	Quantiles . . . . .	590
32.21.4	Properties . . . . .	590
32.21.5	Random Numbers . . . . .	591
32.22	Laplace Distribution . . . . .	591
32.22.1	Density and CDF . . . . .	591
32.22.2	Quantiles . . . . .	592
32.22.3	Properties . . . . .	592
32.22.4	Random Numbers . . . . .	592
32.23	Logistic Distribution . . . . .	593
32.23.1	Definition . . . . .	593
32.23.2	Density and CDF . . . . .	593
32.23.3	Quantiles . . . . .	593
32.23.4	Properties . . . . .	594
32.23.5	Random Numbers . . . . .	594
32.24	Pareto Distribution . . . . .	594
32.24.1	Definition . . . . .	594
32.24.2	Density and CDF . . . . .	594
32.24.3	Quantiles . . . . .	595
32.24.4	Properties . . . . .	595
32.24.5	Random Numbers . . . . .	596
32.25	Raleigh Distribution . . . . .	596
32.25.1	Definition . . . . .	596
32.25.2	Density and CDF . . . . .	596

32.25.3	Density . . . . .	596
32.25.4	CDF . . . . .	596
32.25.5	Quantiles . . . . .	597
32.25.6	Properties . . . . .	597
32.25.7	Random Numbers . . . . .	597
32.26	Triangular Distribution . . . . .	598
32.26.1	Definition . . . . .	598
32.26.2	Density and CDF . . . . .	598
32.26.3	Quantiles . . . . .	598
32.26.4	Properties . . . . .	599
32.26.5	Random Numbers . . . . .	599
32.27	Uniform Distribution . . . . .	600
32.27.1	Definition . . . . .	600
32.27.2	Density and CDF . . . . .	600
32.27.3	Quantiles . . . . .	600
32.27.4	Properties . . . . .	601
32.27.5	Random Numbers . . . . .	601
<b>33</b>	<b>Noncentral Distribution Functions (based on Boost)</b>	<b>603</b>
33.1	Noncentral Beta-Distribution . . . . .	603
33.1.1	Definition . . . . .	603
33.1.2	Density and CDF . . . . .	603
33.1.3	Quantiles . . . . .	604
33.1.4	Properties . . . . .	604
33.1.5	Random Numbers . . . . .	604
33.2	Noncentral Chi-Square Distribution . . . . .	605
33.2.1	Definition . . . . .	605
33.2.2	Density and CDF . . . . .	605
33.2.3	Quantiles . . . . .	606
33.2.4	Properties . . . . .	606
33.2.5	Random Numbers . . . . .	607
33.3	NonCentral F-Distribution . . . . .	607
33.3.1	Definition . . . . .	607
33.3.2	Density and CDF . . . . .	607
33.3.3	Quantiles . . . . .	608
33.3.4	Properties . . . . .	608
33.3.5	Random Numbers . . . . .	609
33.4	Noncentral Student's t-Distribution . . . . .	609
33.4.1	Definition . . . . .	609
33.4.2	Density and CDF . . . . .	610
33.4.3	Quantiles . . . . .	610
33.4.4	Properties . . . . .	611
33.4.5	Random Numbers . . . . .	611
33.5	Skew Normal Distribution . . . . .	612

33.5.1	Definition . . . . .	612
33.5.2	Density and CDF . . . . .	612
33.5.3	Quantiles . . . . .	613
33.5.4	Properties . . . . .	613
33.5.5	Random Numbers . . . . .	614
33.6	Owen's T-Function . . . . .	614
33.6.1	Owen's T-Function . . . . .	614
<b>34</b>	<b>Ordinary Differential Equations</b>	<b>615</b>
34.1	Defining the ODE System . . . . .	616
34.2	Stepping Functions . . . . .	616
34.2.1	Explicit Euler . . . . .	617
34.2.2	Modified Midpoint . . . . .	617
34.2.3	Runge-Kutta 4 . . . . .	617
34.2.4	Cash-Karp . . . . .	618
34.2.5	Dormand-Prince 5 . . . . .	618
34.2.6	Fehlberg 78 . . . . .	618
34.2.7	Adams-Bashforth . . . . .	619
34.2.8	Adams-Moulton . . . . .	619
34.2.9	Adams-Bashforth-Moulton . . . . .	619
34.2.10	Controlled Runge-Kutta . . . . .	620
34.2.11	Dense Output Runge-Kutta . . . . .	620
34.2.12	Bulirsch-Stoer . . . . .	620
34.2.13	Bulirsch-Stoer Dense Output . . . . .	620
34.2.14	Implicit Euler . . . . .	620
34.2.15	Rosenbrock 4 . . . . .	620
34.2.16	Controlled Rosenbrock 4 . . . . .	621
34.2.17	Dense Output Rosenbrock 4 . . . . .	621
34.2.18	Symplectic Euler . . . . .	621
34.2.19	Symplectic RKN McLachlan . . . . .	621
34.3	Integrate functions: Evolution . . . . .	621
<b>VII</b>	<b>Application Examples</b>	<b>623</b>
<b>35</b>	<b>Examples: Multivariate Special Functions</b>	<b>624</b>
35.1	Functions Of Matrix Arguments . . . . .	624
35.1.1	Multivariate Gamma function $\Gamma_p(x)$ . . . . .	624
35.1.2	Zonal Polynomials . . . . .	625
35.1.3	Gauss Hypergeometric Function of Matrix Argument . . . . .	626
35.1.4	Confluent Hypergeometric Function for Matrix Argument . . . . .	627
35.1.5	Confluent Hypergeometric Limit Function for Matrix Argument . . . . .	628

<b>36 Examples: Moments, cumulants, and expansions</b>	<b>629</b>
36.1 Moments and cumulants . . . . .	629
36.2 The Edgeworth expansion . . . . .	631
36.2.1 Continuous variates . . . . .	631
36.2.2 Lattice (discrete) variates . . . . .	631
36.3 The Cornish-Fisher expansion . . . . .	632
36.4 Saddlepoint approximations . . . . .	633
36.4.1 First order approximations . . . . .	633
36.4.2 Second order approximations . . . . .	633
36.5 Inverse Saddlepoint approximations . . . . .	634
36.6 The Box-Davis expansion . . . . .	635
36.6.1 Definition . . . . .	635
36.6.2 Density and CDF . . . . .	636
36.6.3 Quantiles . . . . .	637
36.6.4 Special Cases: Products of beta variables . . . . .	639
36.6.5 Other Test Criteria concerning Covariance Matrices . . . . .	641
36.6.6 The Lawley-Hotelling and Pillai traces . . . . .	641
36.7 The Product of Independent Beta Variables . . . . .	643
36.7.1 Definition . . . . .	643
36.7.2 Density and CDF . . . . .	643
36.7.3 Quantiles . . . . .	644
36.7.4 Properties . . . . .	645
36.7.5 Random Numbers . . . . .	645
<b>37 Examples: Continuous Distribution Functions</b>	<b>647</b>
37.1 Distribution of the Sample Correlation Coefficient . . . . .	647
37.1.1 Definition . . . . .	647
37.1.2 Density and CDF . . . . .	647
37.1.3 Quantiles . . . . .	651
37.1.4 Properties . . . . .	652
37.1.5 Random Numbers . . . . .	653
37.1.6 Confidence limits for rho . . . . .	654
37.1.7 Sample Size Function . . . . .	655
37.2 Distribution of the Sample Multiple Correlation Coefficient . . . . .	656
37.2.1 Definition . . . . .	656
37.2.2 Density and CDF . . . . .	656
37.2.3 Quantiles . . . . .	659
37.2.4 Properties . . . . .	659
37.2.5 Random Numbers . . . . .	660
37.2.6 Confidence Limits for the Noncentrality Parameter . . . . .	661
37.2.7 Sample Size . . . . .	661
37.3 Skew Normal Distribution . . . . .	663
37.3.1 Definition . . . . .	663

37.3.2	Density and CDF . . . . .	663
37.3.3	Quantiles . . . . .	663
37.3.4	Properties . . . . .	664
37.3.5	Random Numbers . . . . .	664
37.3.6	Owen's T-Function . . . . .	665
37.4	Multivariate Normal Distribution . . . . .	666
37.4.1	Definition . . . . .	666
37.4.2	Density of the multivariate normal distribution . . . . .	666
37.4.3	Bivariate Normal . . . . .	666
37.4.4	Special structure: zero correlation . . . . .	667
37.4.5	Special structure: Equal-correlated case . . . . .	667
37.4.6	Special structure: Dunnett test . . . . .	667
37.4.7	Special structure: normal range . . . . .	668
37.4.8	Normal Orthant Probabilities . . . . .	668
37.4.9	Normal Rank Order Probabilities . . . . .	669
37.4.10	Random Numbers . . . . .	669
37.5	Multivariate t-Distribution . . . . .	671
37.5.1	Definition . . . . .	671
37.5.2	Studentized Maximum and Maximum Modulus . . . . .	671
37.5.3	Dunnett's t . . . . .	671
37.5.4	Studentized Range . . . . .	672
37.5.5	Special case Owen . . . . .	672
37.6	Noncentral Chi-Square Distribution . . . . .	673
37.6.1	Definition . . . . .	673
37.6.2	Density and CDF . . . . .	673
37.6.3	Quantiles . . . . .	675
37.6.4	Properties . . . . .	676
37.6.5	Confidence Limits for the Noncentrality Parameter . . . . .	677
37.6.6	Sample Size Calculation . . . . .	678
37.6.7	Random Numbers . . . . .	678
37.7	Noncentral Beta-Distribution . . . . .	681
37.7.1	Definition . . . . .	681
37.7.2	Density and CDF . . . . .	681
37.7.3	Quantiles . . . . .	681
37.7.4	Properties . . . . .	682
37.7.5	Random Numbers . . . . .	683
37.8	Noncentral Student's t-Distribution . . . . .	684
37.8.1	Definition . . . . .	684
37.8.2	Density and CDF . . . . .	684
37.8.3	Quantiles . . . . .	687
37.8.4	Properties . . . . .	688
37.8.5	Random Numbers . . . . .	688
37.8.6	Confidence Limits for the Noncentrality Parameter . . . . .	689
37.8.7	Sample Size Function . . . . .	690
37.9	Doubly Noncentral Student's t-Distribution . . . . .	691

37.9.1	Definition . . . . .	691
37.9.2	Density and CDF . . . . .	691
37.9.3	Quantiles . . . . .	693
37.9.4	Properties . . . . .	694
37.9.5	Random Numbers . . . . .	694
37.9.6	Confidence Limits for the Noncentrality Parameter . . . . .	695
37.9.7	Sample Size Function . . . . .	695
37.10	NonCentral F-Distribution . . . . .	697
37.10.1	Definition . . . . .	697
37.10.2	Density and CDF . . . . .	697
37.10.3	Quantiles . . . . .	699
37.10.4	Properties . . . . .	700
37.10.5	Random Numbers . . . . .	701
37.10.6	Confidence Limits for the Noncentrality Parameter . . . . .	702
37.10.7	Sample Size . . . . .	702
37.11	Doubly NonCentral F-Distribution . . . . .	703
37.11.1	Definition . . . . .	703
37.11.2	Density and CDF . . . . .	703
37.11.3	Quantiles . . . . .	706
37.11.4	Properties . . . . .	706
37.11.5	Random Numbers . . . . .	708
37.11.6	Confidence Limits for the Noncentrality Parameters . . . . .	708
37.11.7	Sample Size . . . . .	709
37.12	Noncentral Distribution of Roy's Largest Root . . . . .	710
37.12.1	Definition . . . . .	710
37.12.2	The exact distribution (null case). . . . .	710
37.12.3	Approximation of the CDF(null case) . . . . .	711
37.13	Noncentral Distribution of Wilks' Lambda . . . . .	712
37.13.1	Definitions . . . . .	712
37.13.2	Density and CDF (Overview) . . . . .	712
37.13.3	Density: Details . . . . .	713
37.13.4	CDF: Details . . . . .	713
37.13.5	Quantiles . . . . .	717
37.13.6	Properties . . . . .	717
37.13.7	Random Numbers . . . . .	718
37.14	Noncentral Distribution of Hotelling's T2 . . . . .	719
37.14.1	Hotelling's T2, Central Moments . . . . .	719
37.14.2	Hotelling's T2, Exact distributions for p=1 and p=2 . . . . .	719
37.14.3	Hotelling's T2, Approximation . . . . .	719
37.14.4	Hotelling's T2, Noncentral distribution: GLM . . . . .	720
37.14.5	Hotelling's T2, Noncentral distribution: CORR . . . . .	720
37.15	Noncentral Distribution of Pillai's V . . . . .	722
37.15.1	Pillai's V, Central Moments . . . . .	722
37.15.2	Pillai's V, Special cases . . . . .	722
37.15.3	Pillai's V, Other Properties . . . . .	722

37.15.4	Pillai's V, Noncentral distribution: GLM . . . . .	723
37.15.5	Pillai's V, Noncentral distribution: CORR . . . . .	723
37.16	Noncentral Distribution of Bartlett's M (2 samples) . . . . .	725
37.16.1	Definition . . . . .	725
37.16.2	Density and CDF . . . . .	725
37.16.3	Quantiles . . . . .	726
37.16.4	Properties . . . . .	726
37.16.5	Random Numbers . . . . .	727
<b>38</b>	<b>Examples: Discrete Distribution Functions</b>	<b>728</b>
38.1	Noncentral Distribution of Mann-Whitney's U (with Stratification) . . . . .	729
38.1.1	Definition . . . . .	729
38.1.2	Central distribution . . . . .	729
38.1.3	Confidence interval for a location parameter . . . . .	730
38.1.4	Noncentral distribution . . . . .	730
38.1.5	Power for Ordered Categorical Data . . . . .	733
38.2	Noncentral Distribution of Wilcoxon's Signed Rank Test (with Stratification) . . . . .	734
38.2.1	Definition . . . . .	734
38.2.2	Central Density . . . . .	735
38.2.3	Confidence interval for the median . . . . .	735
38.2.4	Noncentral distribution . . . . .	736
38.2.5	Moments of the noncentral distribution . . . . .	737
38.2.6	Sample Size . . . . .	737
38.3	Noncentral Distribution of Kendall's Tau . . . . .	738
38.3.1	Definition . . . . .	738
38.3.2	Central distribution . . . . .	738
38.3.3	Noncentral distribution . . . . .	738
38.4	Noncentral Distribution of Jonckheere-Terpsta's S . . . . .	741
38.4.1	Definition . . . . .	741
38.4.2	Central distribution of Jonckheere-Terpsta's S . . . . .	741
38.5	Noncentral Distribution of Spearman's Rho . . . . .	742
38.5.1	Definition . . . . .	742
38.5.2	Central Cumulants . . . . .	742
38.5.3	Noncentral Moments . . . . .	742
38.5.4	Recursive algorithm . . . . .	742
38.6	Noncentral Distribution of Page's L . . . . .	743
38.6.1	Definition . . . . .	743
38.6.2	Central Cumulants . . . . .	743
38.6.3	Noncentral Moments . . . . .	743
38.6.4	Recursive algorithm . . . . .	743
38.7	Noncentral Distribution of Kruskal-Wallis' H . . . . .	744
38.7.1	Definition . . . . .	744
38.7.2	Density of total scores of a one-way layout . . . . .	744
38.7.3	Moments and cumulants of linear rank statistics for the one-way layout	745

38.7.4	Special cases for the one-way layout (H-type) . . . . .	745
38.8	Noncentral Distributions of Friedman's S . . . . .	746
38.8.1	Density of total scores in a two-way layout . . . . .	746
38.8.2	Moments and cumulants of linear rank statistics for the two-way layout . . . . .	746
38.8.3	Special cases for the two-way layout . . . . .	747
<b>39 Examples: Statistical Procedures</b>		<b>748</b>
39.1	Introduction to Inferential Statistics Functions . . . . .	748
39.1.1	The Linear Model . . . . .	748
39.1.2	Transformations . . . . .	748
39.1.3	Randomisation, Block-Randomisation, and Permutation Tests . . . . .	748
39.1.4	Commonly Used Function Types . . . . .	749
39.2	Tests for the mean from 1 sample (Student's t-test) . . . . .	754
39.2.1	Overview . . . . .	754
39.2.2	Tests and Confidence Intervals . . . . .	754
39.2.3	Power . . . . .	757
39.2.4	Sample Size Calculation . . . . .	758
39.2.5	Confidence Interval (Effect size) . . . . .	759
39.2.6	Tolerance Intervals . . . . .	759
39.2.7	Prediction Intervals . . . . .	760
39.3	Tests for means from 2 independent samples (Student's t-test) . . . . .	761
39.3.1	Overview . . . . .	761
39.3.2	Tests and Confidence Intervals . . . . .	761
39.3.3	Power . . . . .	764
39.3.4	Sample Size Calculation . . . . .	766
39.3.5	Confidence Interval (Effect size) . . . . .	767
<b>VIII Appendices</b>		<b>768</b>
<b>A Interfaces to the C family of languages</b>		<b>769</b>
A.1	Windows, GNU/Linux, Mac OSX: GNU Compiler Collection . . . . .	769
A.2	Windows: MSVC . . . . .	769
A.3	Windows, GNU/Linux, Mac OSX: C . . . . .	771
A.4	Windows, GNU/Linux, Mac OSX: C++ . . . . .	773
A.5	Mac OSX: Objective C . . . . .	775
A.6	Mac OSX: Objective C++ . . . . .	776
<b>B Languages with CLR Support</b>		<b>778</b>
B.1	Visual Basic .NET . . . . .	778
B.2	C# 4.0 . . . . .	786

B.3	JScript 10.0	789
B.4	C++ 10.0, Visual Studio	791
B.5	F# 3.0	793
B.6	IronPython 2.7	795
B.7	MatLab (.NET interface)	796
<b>C</b>	<b>Building the library</b>	<b>798</b>
C.1	Building the Library, Part 1	799
C.1.1	Downloading GCC and the MinGW-w64 toolchain	799
C.1.2	Downloading, installing and configuring MSYS2	799
C.1.3	Downloading, installing and configuring Code::Blocks	800
C.1.4	Downloading, compiling and installing GMP	801
C.1.5	Downloading, compiling and installing MPFR	802
C.1.6	Downloading, compiling and installing FLINT	803
C.1.7	Downloading, compiling and installing ARB	805
C.2	Building the Library, Part 2	807
C.2.1	Boost Math	807
C.2.2	Boost Random	807
C.2.3	Eigen	807
C.2.4	Source Code derived from other libraries	807
C.3	Building the documentation	808
C.4	Additional libraries	809
C.4.1	MPIR	809
C.4.2	gmpfrxx	809
C.5	Working Notes	809
C.6	Where to find VB Code	809
C.7	How to run Permutation Code	809
<b>D</b>	<b>Roadmap</b>	<b>810</b>
D.1	CPython	811
D.1.1	Downloading and installing CPython 2.7	811
D.1.2	Plotting	812
D.1.3	Using the C-API	816
D.1.4	Interfaces to the C family of languages	816
D.1.5	Cython: C extensions for the Python language	822
D.1.6	A Windows-specific interface: using COM	822
D.2	R (Statistical System)	824
<b>E</b>	<b>Acknowledgements</b>	<b>827</b>
E.1	Contributors to libraries used in the numerical routines	827

E.1.1	Contributors to GMP . . . . .	827
E.1.2	Contributors to MPFR . . . . .	829
E.1.3	Contributors to FLINT . . . . .	830
E.1.4	Contributors to ARB . . . . .	830
E.1.5	Contributors to MPFRC++ . . . . .	830
E.1.6	Contributors to Eigen . . . . .	830
E.1.7	Contributors to Boost Multiprecision . . . . .	833
E.1.8	Contributors to Boost Math . . . . .	833
E.1.9	Contributors to Boost Random . . . . .	834
E.1.10	Contributors to Boost Odeint . . . . .	835
E.1.11	Contributors to NLOpt . . . . .	835
<b>F</b>	<b>Licenses</b>	<b>837</b>
F.1	GNU Licenses . . . . .	837
F.1.1	GNU General Public License, Version 2 . . . . .	837
F.1.2	GNU Library General Public License, Version 2 . . . . .	843
F.1.3	GNU Lesser General Public License, Version 3 . . . . .	849
F.1.4	GNU General Public License, Version 3 . . . . .	851
F.1.5	GNU Free Documentation License, Version 1.3 . . . . .	860
F.2	Other Licenses . . . . .	866
F.2.1	Mozilla Public License, Version 2.0 . . . . .	866
F.2.2	Boost Software License, Version 1.0 . . . . .	871
F.2.3	MIT License . . . . .	872
<b>Appendices</b>		<b>769</b>
<b>IX</b>	<b>Back Matter</b>	<b>873</b>
<b>Bibliography</b>		<b>874</b>
<b>Index</b>		<b>895</b>

# List of Tables

39.1	Student's t-test, 1 sample, tests for the mean . . . . .	754
39.2	Student's t-test, 1 sample, confidence intervals for the mean . . . . .	755
39.3	Student's t-test, 1 sample, power calculations . . . . .	757
39.4	Student's t-test, 1 sample, sample size calculations . . . . .	758
39.5	Student's t-test, 2 independent samples, tests for the mean . . . . .	762
39.6	Student's t-test, 2 independent samples, confidence intervals for the difference of the means . . . . .	762
39.7	Student's t-test, 2 independent samples, power calculations . . . . .	764
39.8	Student's t-test, 2 independent samples, sample size calculations . . . . .	766

# List of Figures

2.1	A pdf plot	7
2.2	plot of a 2-dimensional function	8
2.3	Surface plot of the probability density function of the bivariate normal distribution with $\rho = -0.5$	9
2.4	3D plot of a parametric function: Seashell	10
2.5	3D plot of Kuen's surface	11
2.6	3D plot of Klein's Bottle	12
2.7	Surface plot of the real and imaginary component of $z = \tan(x + iy)$	13
2.8	Surface plot of the magnitude of $z = \sin(x + iy)$	14
5.1	Complex Exponential Function	85
5.2	Complex Logarithm	90
5.3	Complex Square	93
5.4	Complex Cube	94
5.5	Complex Square Root	96
5.6	Complex Sine	103
5.7	Complex Cosine	105
5.8	Complex Tangent	107
5.9	Complex Secant	109
5.10	Complex Cosecant	111
5.11	Complex Cotangent	113
5.12	Complex Hyperbolic Sine	117
5.13	Complex Hyperbolic Cosine	119
5.14	Complex Hyperbolic Tangent	121
5.15	Complex Hyperbolic Secant	123
5.16	Complex Hyperbolic Cosecant	124
5.17	Complex Hyperbolic Cotangent	125
5.18	Complex Arcsine	126
5.19	Complex Arccosine	128
5.20	Complex Arctangent	130
5.21	Complex Arccotangent	133
5.22	Complex Inverse Hyperbolic Sine	134
5.23	Complex Inverse Hyperbolic Cosine	135
5.24	Complex Inverse Hyperbolic Tangent	136
5.25	Complex Inverse Hyperbolic Cotangent	137
D.1	MpMath 2Dplot	812
D.2	MpMath cplot	814
D.3	MpMath Surface plot	815

# Part I

## Getting Started

# Chapter 1

## Introduction

### 1.1 Overview: Features and Setup

#### 1.1.1 Features

The mpFormulaC distribution consists of two parts: the mpFormulaC Library and the mpFormulaC Toolbox.

#### 1.1.2 The mpFormulaC Library

The mpFormulaC Library is a collection of numerical functions and procedures in multiprecision arithmetic. It is intended to be usable on multiple platforms (i.e. platforms supported by a recent version of the GNU Compiler Collection, e.g. Windows, GNU/Linux, Mac OS) and is provided in the form of source code, with interfaces to C and C++.

The following multi-precision types are supported:

- The FMPZ arbitrary precision integer type of the FLINT library.
- The FMPQ arbitrary precision rational type of the FLINT library.
- The MPD arbitrary precision decimal floating point type of the libmpdec library.
- The MPFR arbitrary precision real binary floating point type of the MPFR library.
- The MPC arbitrary precision complex binary floating point type of the MPC library.
- The ARB arbitrary precision real binary interval arithmetic floating point type of the ARB library.
- The ACB arbitrary precision complex binary interval arithmetic floating point type of the ARB library.

In addition, the following hardware-based floating-point types are supported:

- The conventional single (32 bit) precision real binary floating point type (float in C).
- The conventional single (32 bit) precision complex binary floating point type (float in C).
- The conventional double (64 bit) precision real binary floating point type (double in C).

- The conventional double (64 bit) precision complex binary floating point type (double in C).
- The extended precision (80 bit) real binary floating point type of the Intel FPU.
- The extended precision (80 bit) complex binary floating point type of the Intel FPU.

All of these types are available as scalars, vectors, and matrices.

The mpFormulaC Library is based on GMP ([Granlund & the GMP development team, 2013](#)), MPFR ([Fousse et al., 2007](#)), MPC ([Enge et al., 2012](#)), MPFRC++ ([Holoborodko, 2008-2012](#)), gmpfrxx ([Wilkening, 2008](#)), libmpdec ([Krah, 2012](#)), Eigen ([Guennebaud et al., 2010](#)), Boost Math ([Bristow et al., 2013](#)), Boost Random ([Maurer & Watanabe, 2013](#)), FLINT [Hart \(2010\)](#), ARB [Johansson \(2013\)](#).

### 1.1.3 The mpFormulaC Toolbox

The mpFormulaC Toolbox provides precompiled binaries for the Windows platform with multiple interfaces:

- A C interface: provides the most direct and efficient access to the numerical routines, and can be used as basis for other interfaces. Is intended to work with most C compilers, and can also be used from Objective C.
- A C++ interface: provides a rich set of multiprecision arithmetic functions, operators and procedures, which are accessible in a familiar syntax, thanks to operator overloading. Both 32 bit and 64 bit versions are provided. Is intended to work with most C++ compilers, and can also be used from Objective C++.
- A COM (Component Object Model) interface: multiprecision arithmetic functions and procedures, with arithmetic operators emulated as properties. Both 32 bit and 64 bit versions are provided. This interface makes the numerical routines available to all languages with COM support, including VBScript, JScript (Windows Script Host), Visual Basic for Applications, Visual Basic 6.0, OpenOffice Basic, Lua, Ruby, PHP CLI, Perl, Python, R (Statistical System) and Mathematica.
- A .NET Framework 4.0 interface: As for C++, arithmetic functions, operators and procedures are accessible in a familiar syntax. Both 32 bit and 64 bit versions are provided. This interface makes the numerical routines available to all languages with .NET Framework support, including VB.NET, C#, JScript 2010, F#, MS C++ (CLI), IronPython and Matlab.
- A Names Pipes and Command Line interface: this is designed to make sure that the calling application and the routines in the library are executed in separate processes, greatly enhancing stability.

### 1.1.4 System Requirement

This mpFormulaC Library and Toolbox has the following system requirement:

- Microsoft Windows with Microsoft .NET Framework version 4.x (Full).

### 1.1.5 Installation

The mpFormulaC Library and Toolbox can be downloaded from

<http://mpFormula.github.io/C/>.

Unzip the downloaded file in a directory for which you have write-access.

## 1.2 License

The mpFormulaC Library and Toolbox is free software. It is overall licensed under the GNU General Public License, Version 3 (see appendix [F.1.4](#)). Note however that the underlying libraries come with their own license terms; see the appendix for details.

The manual for the mpFormulaC Library and Toolbox (this document) is licensed under the GNU Free Documentation License, Version 1.3 (see appendix [F.1.5](#)).

## 1.3 No Warranty

There is no warranty. See the GNU General Public License, Version 3 (see appendix [F.1.4](#)) for details.

## 1.4 Related Software

The mpFormulaPy Library and Toolbox provides multiprecision routines written in Python, with interfaces to CPython, R, .NET and COM. It can be downloaded from

<http://mpFormula.github.io/Py/>.

# Chapter 2

## Tutorials

### 2.1 Why multi-precision arithmetic?

An introduction to the problems of rounding errors and catastrophic cancellation can be found in [Goldberg \(1991\)](#). Excellent reference texts are [Higham \(2002\)](#) and [Higham \(2009\)](#).

In the following paragraphs we will give a few examples of how widely used programs like MS Excel or Libreoffice Calc can give wrong results due to the fact that they are using double precision arithmetic and not multi-precision arithmetic

#### 2.1.1 Example 1: Sums

Sums are often calculated exactly if all summands have an exact representation. If this is not the case, results can be unpredictable. In MS Excel, the formula

```
=SUM(10000000000,-16000000000,6000000000)
```

will give the correct result 0, but the analogous formula

```
=SUM(1E+40,-1.6E+40,6E+39)
```

returns  $1.20893E+24$  instead of the correct result 0.

#### 2.1.2 Example 2: Standard Deviation

Like sums, variances and standard deviations are often calculated exactly if all arguments have an exact representation. If this is not the case, results can again be unpredictable. In MS Excel, the formula

```
=VAR(1E+30,1E+30,1E+30)
```

returns  $2.97106E+28$  instead of the correct result 0, which should be the obvious results since all arguments are the same.

#### 2.1.3 Example 3: Overflow and underflow

In many situations where the final result is representable in double precision, some of the interim results cause overflow or underflow. A popular example is the function  $f(x, y) = \sqrt{x^2 + y^2}$ . With  $x = 3 \cdot 10^{300}$  and  $y = 4 \cdot 10^{300}$  the result  $f(x, y) = 5 \cdot 10^{300}$  is representable in double precision, but the (naive) calculation will overflow.

### 2.1.4 Example 4: Polynomials

Consider the following example [Cuyt et al. \(2001\)](#):

For  $a = 77617$  and  $b = 33096$ , calculate

$$Y = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} \quad (2.1.1)$$

The correct result is  $Y = -54767/66192 = -0.827396\dots$

### 2.1.5 Example 5: Trigonometric Functions

Trigonometric functions are sensitive to small perturbations.

In double precision and binary floating point arithmetic, the tangent of  $x = 1.57079632679489$  is calculated as  $\tan(x) = 1.48752 \cdot 10^{14}$ , whereas the correct result is  $\tan(x) = 1.51075 \cdot 10^{14}$ . This amounts to an absolute error of  $2.32287 \cdot 10^{12}$  and a relative error of 1.54%.

There are also limits on the range of arguments, e.g.  $\sin(10^8)$  returns the value  $0.931639\dots$  (with an relative error of  $-6.22776 \cdot 10^{-13}$ ), whereas  $\sin(10^9)$  returns an invalid result (the exact result is  $0.545843\dots$ )

### 2.1.6 Example 6: Logarithms and Exponential Functions

Consider the following example [\(Ghazi et al., 2010\)](#) :

Determine 10 decimal digits of the constant

$$Y = 173746a + 94228b - 78487c, \quad \text{where} \quad (2.1.2)$$

$$a = \sin(10^{22}), b = \ln(17.1), c = \exp(0.42). \quad (2.1.3)$$

The expected result is  $Y = -1.341818958 \cdot 10^{-12}$ .

### 2.1.7 Example 7: Linear Algebra

#### 2.1.7.1 Linear Solver

The following example is from [Hofschuster & Krämer \(2004\)](#):

We want to solve the (ill-conditioned) system of linear equations  $Ax = b$  with

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix}, b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (2.1.4)$$

The correct solution is  $x_1 = 205117922$ ,  $x_2 = 83739041$ .

To solve this  $2 \times 2$  system numerically we first use the well known formulas

$$x_1 = \frac{a_{22}}{a_{11}a_{22} - a_{12}a_{21}}, \quad x_2 = \frac{-a_{21}}{a_{11}a_{22} - a_{12}a_{21}}, \quad (2.1.5)$$

Calculating this directly in double precision gives the following wrong result:

$x_1 = 102558961$ ,  $x_2 = 41869520.5$

## 2.2 Graphics using Latex

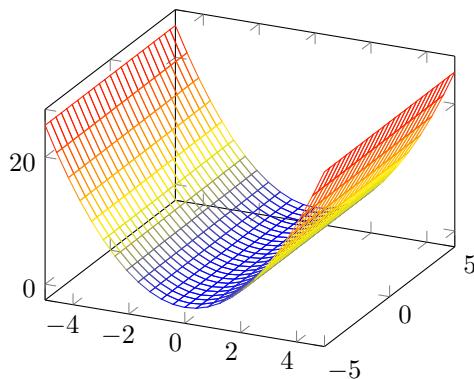
pgfplots ([Feuersänger, 2014](#)) - A TeX package to draw normal and/or logarithmic plots directly in TeX in two and three dimensions with a user-friendly interface and pgfplotstable - a TeX package to round and format numerical tables. Examples in manuals and/or on web site.

<http://pgfplots.net/>.

<http://pgfplots.sourceforge.net/>.

<http://pgfplots.sourceforge.net/gallery.html>.

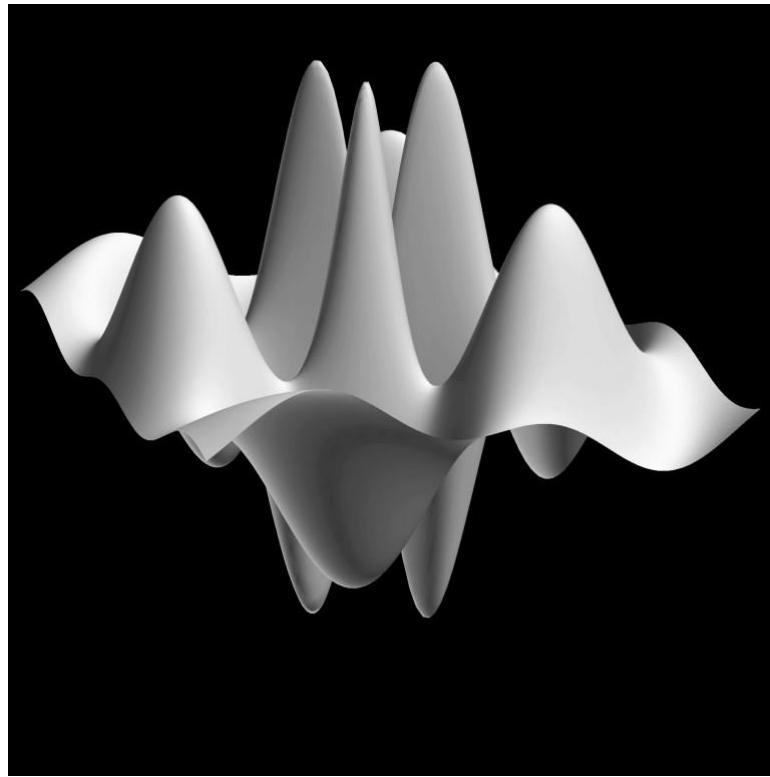
[https://www.sharelatex.com/learn/Pgfplots\\_package](https://www.sharelatex.com/learn/Pgfplots_package).



**Figure 2.1:** A pdf plot

## 2.3 Graphics using .NET Framework

The mpformulaPy toolbox offers a facility for producing 3D charts. The following pages give a few examples.



**Figure 2.2:** plot of a 2-dimensional function

The corresponding code is:

---

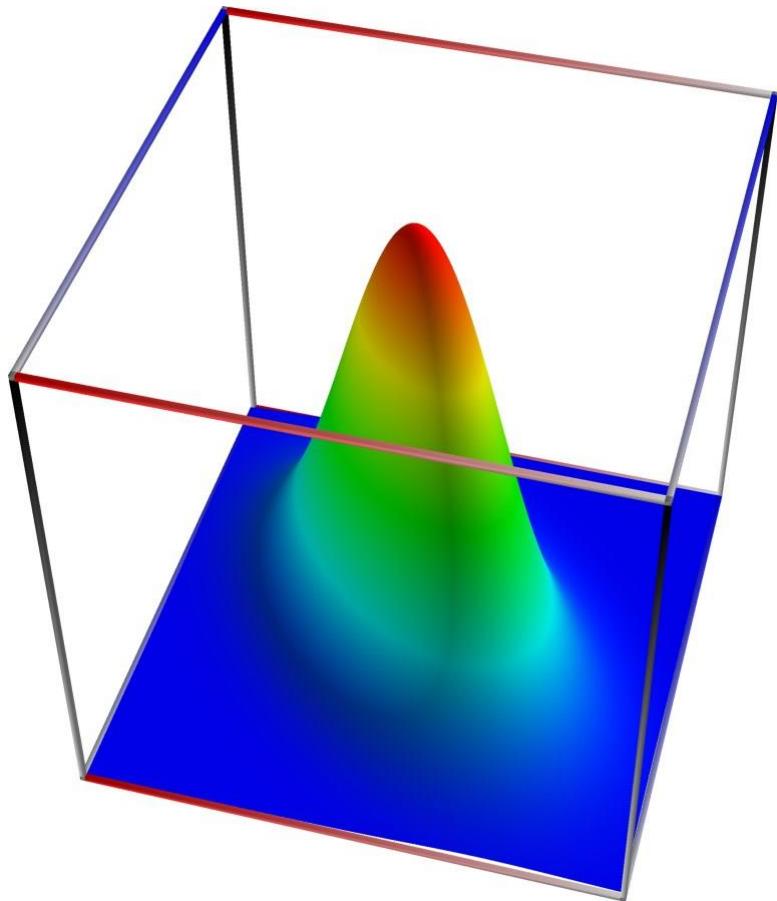
```
const double two_pi = 2 * Math.PI;
double r2 = x * x + z * z;
double r = Math.Sqrt(r2);
double theta = Math.Atan2(z, x);
result = Math.Exp(-r2) * Math.Sin(two_pi * r) * Math.Cos(3 * theta);
```

---

### 2.3.1 Surface plots for bivariate real functions

The bivariate normal distribution has the following density:

$$g(x, y; \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} e^{\frac{-(x^2-2\rho xy+y^2)}{2(1-\rho^2)}} \quad (2.3.1)$$



**Figure 2.3:** Surface plot of the probability density function of the bivariate normal distribution with  $\rho = -0.5$

The corresponding code is:

---

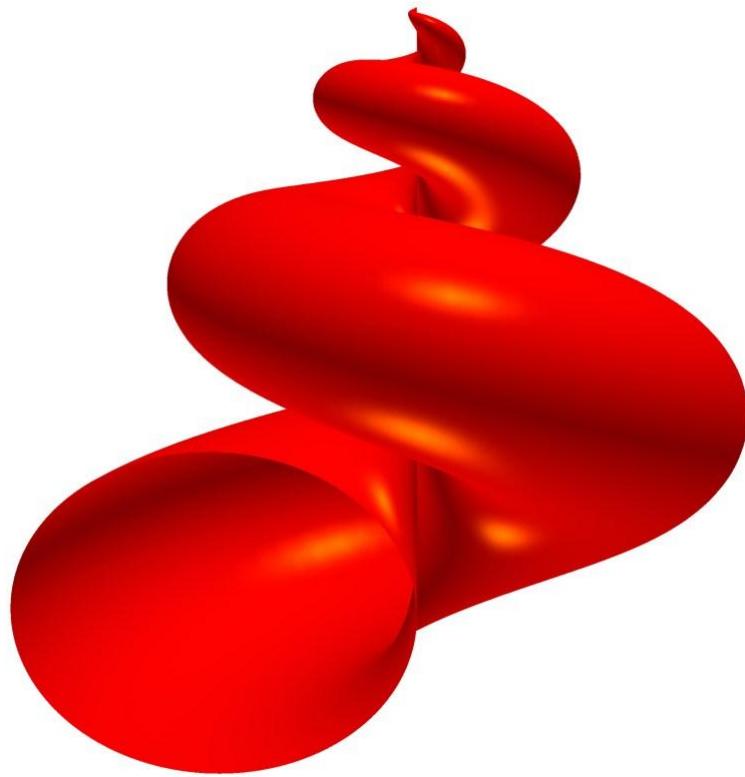
```
const double two_pi = 2 * Math.PI;
double rho = -0.5;
double r2 = 1.0 - rho*rho;
double f = 1 / (two_pi * Math.Sqrt(r2));
double e = -(x*x - 2*rho*x*z + z*z)/(2*r2);
result = f * Math.Exp(e);
```

---

## 2.3.2 3D Plots of parametric functions

### 2.3.2.1 3D Plot of a Seashell

This is a plot of a seashell.



**Figure 2.4:** 3D plot of a parametric function: Seashell.  $u_{\min} = 0$ ;  $u_{\max} = 6 * \text{Math.PI}$ ;  $v_{\min} = 0$ ;  $v_{\max} = 6 * \text{Math.PI}$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

The parametrization is:

---

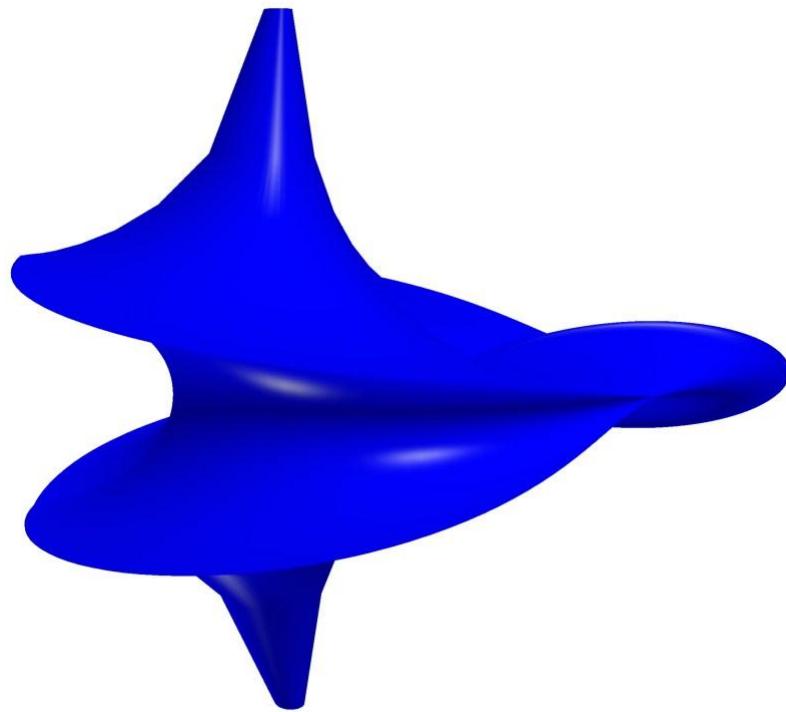
```
double a = Math.Exp(u / (6.0 * Math.PI));
double b = Math.Cos(v / 2.0);

x = 2.0 * (1.0 - a) * Math.Cos(u) * b * b;
z = 2.0 * (-1.0 + a) * Math.Sin(u) * b * b;
y = 1.0 - a * a - Math.Sin(v) * (1.0 - a);
```

---

### 2.3.2.2 3D Plot of Kuen's surface

This is a plot of Kuen's surface.



**Figure 2.5:** 3D plot of a parametric function: Kuen's surface.  $u_{\min} = -4.5$ ;  $u_{\max} = 4.5$ ;  $v_{\min} = 0.01$ ;  $v_{\max} = 3.14$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

The parametrization is:

---

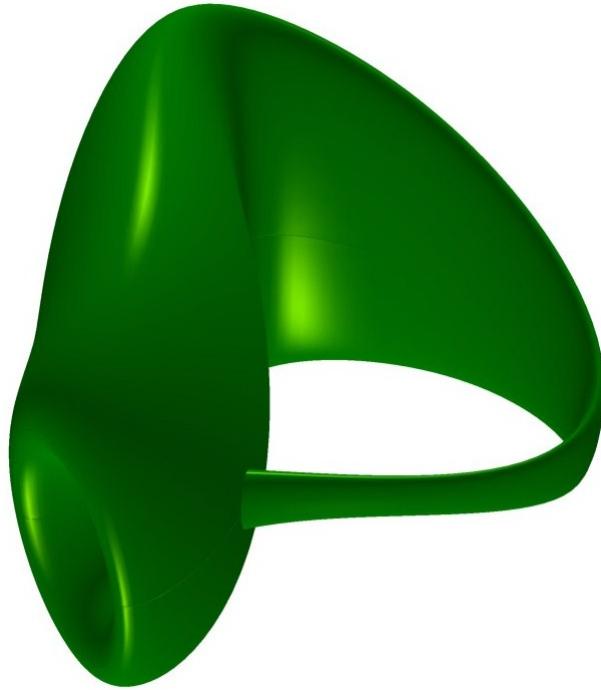
```
double a = 1.0 * Math.Sin(v);
double b = 1.0 + u * u * a * a;

x = 2.0 * a * (Math.Cos(u) + u * Math.Sin(u)) / b;
z = 2.0 * a * (Math.Sin(u) - u * Math.Cos(u)) / b;
y = Math.Log(Math.Tan(v/2.0)) + 2.0 * Math.Cos(v) / b;
```

---

### 2.3.2.3 3D Plot of Klein's Bottle

This is a plot of Klein's Bottle.



**Figure 2.6:** 3D plot of a parametric function: Klein's Bottle.  $u_{\min} = 0.0$ ;  $u_{\max} = 3.14$ ;  $v_{\min} = 0.0$ ;  $v_{\max} = 6.28$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

The following parametrization is due to Robert Israel (with some rearrangements):

---

```

double a = Math.Cos(u);
double b = Math.Sin(u);
double c = Math.Cos(v);
double a2 = a * a;
double a4 = a2 * a2;

x = -(2.0/15.0) * a * (3*c + b*(-30 + a4*(90 - 60*a2) + 5*a*c));
z = -(1.0/15.0) * b*b * (c*b* (3 - 48*a4 + 5*a*b*(1 - 16*a4)) - 60);
y = (2.0/15.0) * (3 + 5*a*b) * Math.Sin(v);

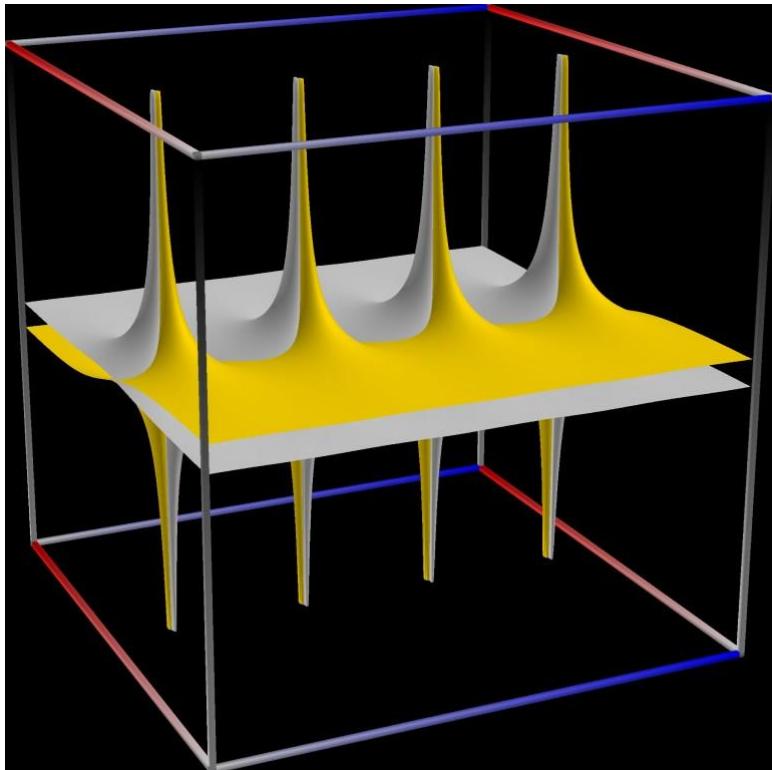
```

---

### 2.3.3 Surface plots of complex functions

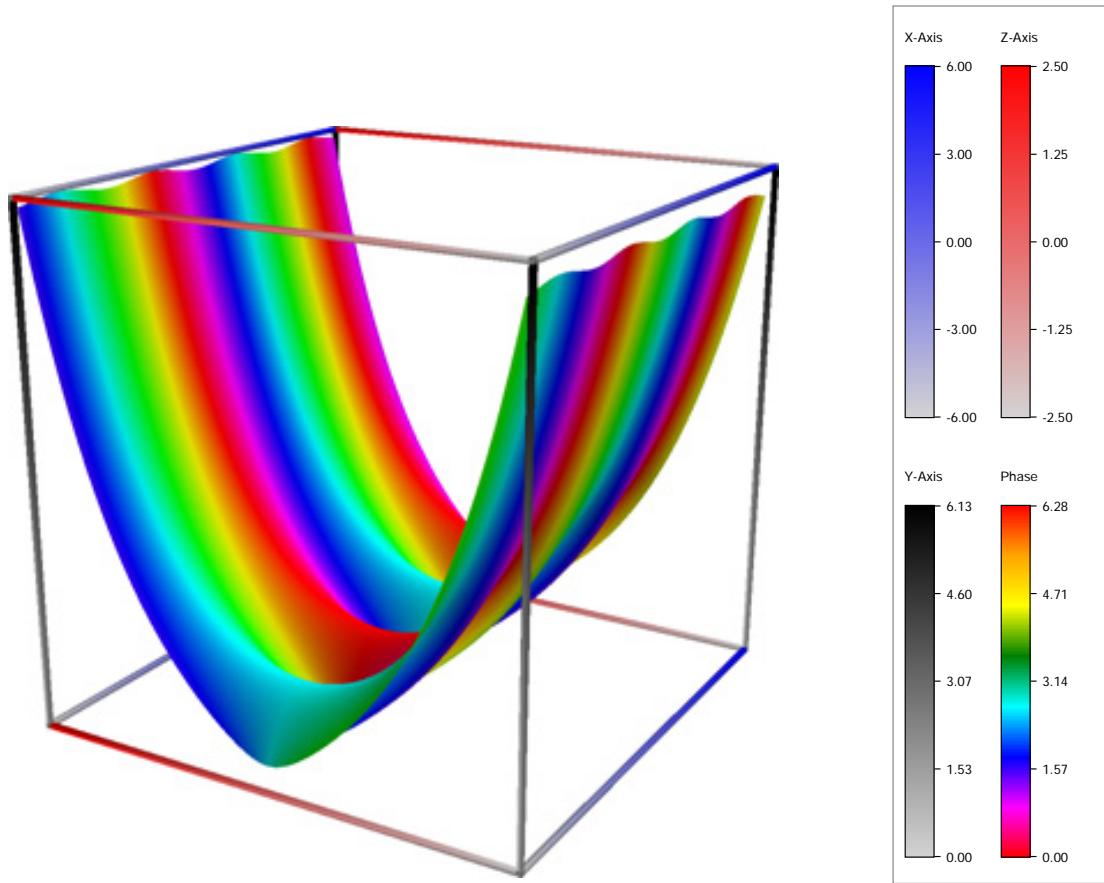
It is straight forward to produce surface plots of complex functions; these are available in two forms:

As plots of the real and imaginary component:



**Figure 2.7:** Surface plot of the real ("silver") and imaginary ("gold") component of  $z = \tan(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $-10 \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

As plots of the absolute value with the phase color-coded:



**Figure 2.8:** Surface plot of the magnitude and phase (color-coded) of  $z = \sin(x+iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $-10 \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

# Chapter 3

## Python: Built-in numerical types

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions. Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and do not return a specific item, never return the collection instance itself but `None`. Some operations are supported by several object types; in particular, practically all objects can be compared, tested for truth value, and converted to a string (with the `repr()` function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.

### 3.1 Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below. The following values are considered false:

`None`

`False`

zero of any numeric type, for example, `0`, `0.0`, `0j`.

any empty sequence, for example, `''`, `()`, `[]`.

any empty mapping, for example, `{}`.

instances of user-defined classes, if the class defines a `__bool__()` or `__len__()` method, when that method returns the integer zero or `bool` value `False`.

All other values are considered true – so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations `or` and `and` always return one of their operands.)

### 3.2 Boolean Operations: `and`, `or`, `not`

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
<code>x or y</code>	if <code>x</code> is false, then <code>y</code> , else <code>x</code>	(1)
<code>x and y</code>	if <code>x</code> is false, then <code>x</code> , else <code>y</code>	(2)
<code>not x</code>	if <code>x</code> is false, then <code>True</code> , else <code>False</code>	(3)

Notes: 1. This is a short-circuit operator, so it only evaluates the second argument if the first one is False. 2. This is a short-circuit operator, so it only evaluates the second argument if the first one is True. 3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

### 3.3 Comparisons

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, `x  $\mid$  y  $\mid=$  z` is equivalent to `x  $\mid$  y` and `y  $\mid=$  z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x  $\mid$  y` is found to be false).

This table summarizes the comparison operations:

Operation	Meaning
<code>&lt;</code>	strictly less than
<code>&lt;=</code>	less than or equal
<code>&gt;</code>	strictly greater than
<code>&gt;=</code>	greater than or equal
<code>==</code>	equal
<code>!=</code>	not equal
<code>is</code>	object identity
<code>is not</code>	negated object identity

Objects of different types, except different numeric types, never compare equal. Furthermore, some types (for example, function objects) support only a degenerate notion of comparison where any two objects of that type are unequal. The  `$\mid$` ,  `$\mid=$` ,  `$\mid$`  and  `$\mid=$`  operators will raise a `TypeError` exception when comparing a complex number with another built-in numeric type, when the objects are of different types that cannot be compared, or in other cases where there is no defined ordering.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types (below).

### 3.4 Numeric Types - `int`, `float`, `complex`

There are three distinct numeric types: integers, floating point numbers, and complex numbers. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number `z`, use `z.real` and `z.imag`.

(The standard library includes additional numeric types, fractions that hold rationals, and decimal that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending 'j' or 'J' to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `float()`, and `complex()` can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by ascending priority (operations in the same box have the same priority; all numeric operations have a higher priority than comparison operations):

Operation	Result	Notes
<code>x + y</code>	sum of x and y	
<code>x - y</code>	difference of x and y	
<code>x * y</code>	product of x and y	
<code>x / y</code>	quotient of x and y	(1)
<code>x // y</code>	floored quotient of x and y	
<code>x % y</code>	remainder of x / y	(2)
<code>-x</code>	x negated	
<code>+x</code>	x unchanged	
<code>abs(x)</code>	absolute value or magnitude of x	
<code>int(x)</code>	x converted to integer	(3)(6)
<code>float(x)</code>	x converted to floating point	(4)(6)
<code>complex(re, im)</code>	a complex number with real part re, imaginary part im. im defaults to zero.	(6)
<code>c.conjugate()</code>	conjugate of the complex number c	No
<code>divmod(x, y)</code>	the pair (x // y, x % y)	(2)
<code>pow(x, y)</code>	x to the power y	(5)
<code>x ** y</code>	x to the power y	(5)
<code>math.trunc(x)</code>	x truncated to Integral	(7)
<code>round(x[, n])</code>	x rounded to n digits, rounding half to even. If n is omitted, it defaults to 0.	(7)
<code>math.floor(x)</code>	the greatest integral float $\leq$ x	(7)
<code>math.ceil(x)</code>	the least integral float $\geq$ x	(7)

Notes:

1. Also referred to as integer division. The resultant value is a whole integer, though the result's type is not necessarily `int`. The result is always rounded towards minus infinity: `1//2` is 0, `(-1)//2` is -1, `1//(-2)` is -1, and `(-1)//(-2)` is 0.

2. Not for complex numbers. Instead convert to floats using `abs()` if appropriate.

3. Conversion from floating point to integer may round or truncate as in C; see functions `math.floor()` and `math.ceil()` for well-defined conversions.

4. `float` also accepts the strings "nan" and "inf" with an optional prefix "+" or "-" for Not a Number (NaN) and positive or negative infinity.

5. Python defines `pow(0, 0)` and `0 ** 0` to be 1, as is common for programming languages.

6. The numeric literals accepted include the digits 0 to 9 or any Unicode equivalent (code points with the Nd property).

7. Only real types (int and float).

See <http://www.unicode.org/Public/6.0.0/ucd/extracted/DerivedNumericType.txt> for a complete list of code points with the Nd property.

For additional numeric operations see the `math` and `cmath` modules.

## 3.5 Long integers

### 3.5.1 Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. Negative numbers are treated as their 2's complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (+ and -).

This table lists the bitwise operations sorted in ascending priority (operations in the same box have the same priority):

Operation	Result	Notes
<code>x   y</code>	bitwise or of x and y	
<code>x ^ y</code>	bitwise exclusive or of x and y	
<code>x &amp; y</code>	bitwise and of x and y	
<code>x &lt;&lt; n</code>	x shifted left by n bits	(1)(2)
<code>x &gt;&gt; n</code>	x shifted right by n bits	(1)(3)
<code>~x</code>	the bits of x inverted	

Notes: 1. Negative shift counts are illegal and cause a `ValueError` to be raised. 2. A left shift by n bits is equivalent to multiplication by `pow(2, n)` without overflow check. 3. A right shift by n bits is equivalent to division by `pow(2, n)` without overflow check.

### 3.5.2 Additional Methods on Integer Types

The `int` type implements the `numbers.Integral` abstract base class. In addition, it provides one more method:

#### 3.5.2.1 `int.bit_length()`

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

---

```
>>>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

---

More precisely, if  $x$  is nonzero, then  $x.bit\_length()$  is the unique positive integer  $k$  such that  $2^{k-1} \leq |x| < 2^k$ . Equivalently, when  $|x|$  is small enough to have a correctly rounded logarithm, then  $k = 1 + \text{int}(\log(|x|, 2))$ . If  $x$  is zero, then  $x.bit\_length()$  returns 0.

Equivalent to:

---

```
def bit_length(self):
    s = bin(self)      # binary representation: bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)      # len('100101') --> 6
```

---

### 3.5.2.2 int.to\_bytes

New in version 3.1.

`int.to_bytes(length, byteorder, *, signed=False)` Return an array of bytes representing an integer.

---

```
>>>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() // 8) + 1, byteorder='little')
b'\xe8\x03'
```

---

The integer is represented using  $\text{length}$  bytes. An `OverflowError` is raised if the integer is not representable with the given number of bytes.

The `byteorder` argument determines the byte order used to represent the integer. If `byteorder` is "big", the most significant byte is at the beginning of the byte array. If `byteorder` is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The `signed` argument determines whether two's complement is used to represent the integer. If `signed` is `False` and a negative integer is given, an `OverflowError` is raised. The default value for `signed` is `False`.

### 3.5.2.3 int.from\_bytes

New in version 3.2.

`classmethod int.from_bytes(bytes, byteorder, *, signed=False)` Return the integer represented by the given array of bytes.

---

```
>>>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
```

---

```
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

---

The argument bytes must either be a bytes-like object or an iterable producing bytes.

The byteorder argument determines the byte order used to represent the integer. If byteorder is "big", the most significant byte is at the beginning of the byte array. If byteorder is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use sys.byteorder as the byte order value.

The signed argument indicates whether twoâŽs complement is used to represent the integer.

### 3.5.3 Additional Methods on Float

The float type implements the numbers.Real abstract base class. float also has the following additional methods.

#### 3.5.3.1 float.as\_integer\_ratio()

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises OverflowError on infinities and a ValueError on NaNs.

#### 3.5.3.2 float.is\_integer()

Return True if the float instance is finite with integral value, and False otherwise:

---

```
>>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

---

Two methods support conversion to and from hexadecimal strings. Since Python's floats are stored internally as binary numbers, converting a float to or from a decimal string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

#### 3.5.3.3 float.hex()

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.

#### 3.5.3.4 float.fromhex(s)

Class method to return the float represented by a hexadecimal string s. The string s may have leading and trailing whitespace.

Note that float.hex() is an instance method, while float.fromhex() is a class method.

A hexadecimal string takes the form:

[sign] ['0x'] integer [': fraction] ['p' exponent]

where the optional sign may be either + or -, integer and fraction are strings of hexadecimal digits, and exponent is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of `float.hex()` is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C's `%a` format character or Java's `Double.toHexString` are accepted by `float.fromhex()`.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string `0x3.a7p10` represents the floating-point number  $(3 + 10./16 + 7./16^{**}2) * 2.0^{**}10$ , or 3740.0:

---

```
>>>> float.fromhex('0x3.a7p10')
3740.0
```

---

Applying the reverse conversion to 3740.0 gives a different hexadecimal string representing the same number:

---

```
>>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

---

## 3.6 Fractions

The fractions module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

```
class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)
```

The first version requires that numerator and denominator are instances of numbers.Rational and returns a new Fraction instance with value numerator/denominator. If denominator is 0, it raises a ZeroDivisionError.

The second version requires that other\_fraction is an instance of numbers.Rational and returns a Fraction instance with the same value.

The next two versions accept either a float or a decimal.Decimal instance, and return a Fraction instance with exactly the same value. Note that due to the usual issues with binary floating-point (see Floating Point Arithmetic: Issues and Limitations), the argument to Fraction(1.1) is not exactly equal to 11/10, and so Fraction(1.1) does not return Fraction(11, 10) as one might expect. (But see the documentation for the limit\_denominator() method below.)

The last version of the constructor expects a string or unicode instance. The usual form for this instance is:

[sign] numerator [’/] denominator

where the optional sign may be either '+' or '-' and numerator and denominator (if present) are strings of decimal digits. In addition, any string that represents a finite value and is accepted by the float constructor is also accepted by the Fraction constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

The corresponding code is:

---

```
>>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
```

---

```

Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)

```

---

The Fraction class inherits from the abstract base class numbers.Rational, and implements all of the methods and operations from that class. Fraction instances are hashable, and should be treated as immutable. In addition, Fraction has the following properties and methods:

Changed in version 3.2: The Fraction constructor now accepts float and decimal.Decimal instances.

### 3.6.1 Properties

#### 3.6.1.1 numerator

Numerator of the Fraction in lowest term.

#### 3.6.1.2 denominator

Denominator of the Fraction in lowest term.

### 3.6.2 Methods

#### 3.6.2.1 from\_float(flt)

This class method constructs a Fraction representing the exact value of flt, which must be a float. Beware that Fraction.from\_float(0.3) is not the same value as Fraction(3, 10)

Note: From Python 3.2 onwards, you can also construct a Fraction instance directly from a float.

#### 3.6.2.2 from\_decimal(dec)

This class method constructs a Fraction representing the exact value of dec, which must be a decimal.Decimal instance.

Note: From Python 3.2 onwards, you can also construct a Fraction instance directly from a decimal.Decimal instance.

#### 3.6.2.3 limit\_denominator()

limit\_denominator(max\_denominator=1000000) Finds and returns the closest Fraction to self that has denominator at most max\_denominator. This method is useful for finding rational approximations to a given floating-point number:

---

```

>>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)

```

---

or for recovering a rational number that's represented as a float:

---

```

>>>> from math import pi, cos
>>> Fraction(cos(pi/3))

```

---

---

```

Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)

```

---

### 3.6.2.4 `__floor__()`

Returns the greatest int  $\lfloor \text{self} \rfloor$ . This method can also be accessed through the `math.floor()` function:

---

```

>>>> from math import floor
>>> floor(Fraction(355, 113))
3

```

---

### 3.6.2.5 `__ceil__()`

Returns the least int  $\lceil \text{self} \rceil$ . This method can also be accessed through the `math.ceil()` function.

### 3.6.2.6 `__round__()`

`__round__()`  
`__round__(ndigits)`

The first version returns the nearest int to self, rounding half to even. The second version rounds self to the nearest multiple of `Fraction(1, 10**ndigits)` (logically, if `ndigits` is negative), again rounding half toward even. This method can also be accessed through the `round()` function.

### 3.6.2.7 `fractions.gcd(a, b)`

Return the greatest common divisor of the integers a and b. If either a or b is nonzero, then the absolute value of `gcd(a, b)` is the largest integer that divides both a and b. `gcd(a,b)` has the same sign as b if b is nonzero; otherwise it takes the sign of a. `gcd(0, 0)` returns 0.

## **Part II**

# **Functions With Error bounds**

# Chapter 4

## Basic Usage

In interactive code examples that follow, it will be assumed that all items in the mpFormulaPy namespace have been imported:

---

```
>>> from mpFormulaPy import *
```

---

Importing everything can be convenient, especially when using mpFormulaPy interactively, but be careful when mixing mpFormulaPy with other libraries! To avoid inadvertently overriding other functions or objects, explicitly import only the needed objects, or use the mpFormulaPy or mp.namespaces:

---

```
from mpFormulaPy import sin, cos
sin(1), cos(1)
import mpFormulaPy
mpFormulaPy.sin(1), mpFormulaPy.cos(1)
from mpFormulaPy import mp # mp context object -- to be explained
mp.sin(1), mp.cos(1)>>> from mpFormulaPy import *
```

---

### 4.1 Number types

Mpmath provides the following numerical types:

Class	Description
mpf	Real float
mpc	Complex float
matrix	Matrix

Currently missing: decimals. The MPD reference is [Krah \(2012\)](#)

The following section will provide a very short introduction to the types mpf and mpc. Intervals and matrices are described further in the documentation chapters on interval arithmetic and matrices / linear algebra.

The mpf type is analogous to Python's built-in float. It holds a real number or one of the special values inf (positive infinity), -inf (negative infinity) and nan (not-a-number, indicating an indeterminate result). You can create mpf instances from strings, integers, floats, and other mpf instances:

---

```
>>> mpf(4)
```

---

```
mpf('4.0')
>>> mpf(2.5)
mpf('2.5')
>>> mpf("1.25e6")
mpf('1250000.0')
>>> mpf(mpf(2))
mpf('2.0')
>>> mpf("inf")
mpf('+inf')
```

---

The mpc type represents a complex number in rectangular form as a pair of mpf instances. It can be constructed from a Python complex, a real number, or a pair of real numbers:

---

```
>>> mpc(2,3)
mpc(real='2.0', imag='3.0')
>>> mpc(complex(2,3)).imag
mpf('3.0')
```

---

You can mix mpf and mpc instances with each other and with Python numbers:

---

```
>>> mpf(3) + 2*mpf('2.5') + 1.0
mpf('9.0')
>>> mp.dps = 15 # Set precision (see below)
>>> mpc(1j)**0.5
mpc(real='0.70710678118654757', imag='0.70710678118654757')
```

---

### 4.1.1 Setting the precision

Mpmath uses a global working precision; it does not keep track of the precision or accuracy of individual numbers. Performing an arithmetic operation or calling mpf() rounds the result to the current working precision. The working precision is controlled by a context object called mp, which has the following default states:

---

```
>>> from mpFormulaPy import *
>>> mp.dps
25
>>> mp.prec
86
>>> mp.trap_complex
False
>>>
```

---

The term prec denotes the binary precision (measured in bits) while dps (short for decimal places) is the decimal precision. Binary and decimal precision are related roughly according to the formula prec = 3.33\*dps. For example, it takes a precision of roughly 333 bits to hold an approximation of pi that is accurate to 100 decimal places (actually slightly more than 333 bits is used).

Changing either precision property of the mp object automatically updates the other; usually you just want to change the dps value:

---

```
>>> mp.dps = 100
>>> mp.dps
```

```
100  
>>> mp.prec  
336
```

When the precision has been set, all mpf operations are carried out at that precision:

The precision of complex arithmetic is also controlled by the `mp` object:

```
>>> mp.dps = 10
>>> mpc(1,2) / 3
mpc(real='0.3333333333321', imag='0.6666666666642')
```

There is no restriction on the magnitude of numbers. An mpf can for example hold an approximation of a large Mersenne prime:

```
>>> mp.dps = 15
>>> print mpf(2)**32582657 - 1
1.24575026015369e+9808357
```

Or why not 1 googolplex:

The (binary) exponent is stored exactly and is independent of the precision.

#### 4.1.2 Temporarily changing the precision

It is often useful to change the precision during only part of a calculation. A way to temporarily increase the precision and then restore it is as follows:

```
>>> mp.prec += 2
>>> # do_something()
>>> mp.prec -= 2
```

As of Python 2.5, the `with` statement along with the `mpFormulaPy` functions `workprec`, `workdps`, `extraprec` and `extradps` can be used to temporarily change precision in a more safe manner:

```
>>> from __future__ import with_statement
>>> with workdps(20):
...     print mpf(1)/7
...     with extradps(10):
...         print mpf(1)/7
...
0.14285714285714285714
0.142857142857142857142857142857
>>> mp.dps
```

---

 15

The with statement ensures that the precision gets reset when exiting the block, even in the case that an exception is raised. (The effect of the with statement can be emulated in Python 2.4 by using a try/finally block.)

The workprec family of functions can also be used as function decorators:

---

```
>>> @workdps(6)
... def f():
...     return mpf(1)/3
...
>>> f()
mpf('0.3333331346511841')
```

---

Some functions accept the prec and dps keyword arguments and this will override the global working precision. Note that this will not affect the precision at which the result is printed, so to get all digits, you must either use increase precision afterward when printing or use nstr/nprint:

---

```
>>> mp.dps = 15
>>> print exp(1)
2.71828182845905
>>> print exp(1, dps=50) # Extra digits won't be printed
2.71828182845905
>>> nprint(exp(1, dps=50), 50)
2.7182818284590452353602874713526624977572470937
```

---

Finally, instead of using the global context object mp, you can create custom contexts and work with methods of those instances instead of global functions. The working precision will be local to each context object:

---

```
>>> mp2 = mp.clone()
>>> mp.dps = 10
>>> mp2.dps = 20
>>> print mp.mpf(1) / 3
0.3333333333
>>> print mp2.mpf(1) / 3
0.3333333333333333
```

---

Note: the ability to create multiple contexts is a new feature that is only partially implemented. Not all mpFormulaPy functions are yet available as context-local methods. In the present version, you are likely to encounter bugs if you try mixing different contexts.

### 4.1.3 Providing correct input

Note that when creating a new mpf, the value will at most be as accurate as the input. Be careful when mixing mpFormulaPy numbers with Python floats. When working at high precision, fractional mpf values should be created from strings or integers:

---

```
>>> mp.dps = 30
>>> mpf(10.9) # bad
mpf('10.9000000000000003552713678800501')
>>> mpf('10.9') # good
```

---

```
mpf('10.8999999999999999999999999999999999997')
>>> mpf(109) / mpf(10) # also good
mpf('10.8999999999999999999999999999999999997')
>>> mp.dps = 15
```

---

(Binary fractions such as 0.5, 1.5, 0.75, 0.125, etc, are generally safe as input, however, since those can be represented exactly by Python floats.)

#### 4.1.4 Printing

By default, the `repr()` of a number includes its type signature. This way `eval` can be used to recreate a number from its string representation:

---

```
>>> eval(repr(mpf(2.5)))
mpf('2.5')
```

---

Prettier output can be obtained by using `str()` or `print`, which hide the `mpf` and `mpc` signatures and also suppress rounding artifacts in the last few digits:

---

```
>>> mpf("3.14159")
mpf('3.141599999999999')
>>> print mpf("3.14159")
3.14159
>>> print mpc(1j)**0.5
(0.707106781186548 + 0.707106781186548j)
```

---

Setting the `mp.pretty` option will use the `str()`-style output for `repr()` as well:

---

```
>>> mp.pretty = True
>>> mpf(0.6)
0.6
>>> mp.pretty = False
>>> mpf(0.6)
mpf('0.5999999999999998')
```

---

The number of digits with which numbers are printed by default is determined by the working precision. To specify the number of digits to show without changing the working precision, use `mpFormulaPy.nstr()` and `mpFormulaPy.nprint()`:

---

```
>>> a = mpf(1) / 6
>>> a
mpf('0.1666666666666666')
>>> nstr(a, 8)
'0.16666667'
>>> nprint(a, 8)
0.16666667
>>> nstr(a, 50)
'0.1666666666666665741480812812369549646973609924316'
```

---

### 4.1.5 Contexts

High-level code in mpFormulaPy is implemented as methods on a 'context object'. The context implements arithmetic, type conversions and other fundamental operations. The context also holds settings such as precision, and stores cache data. A few different contexts (with a mostly compatible interface) are provided so that the high-level algorithms can be used with different implementations of the underlying arithmetic, allowing different features and speed/accuracy tradeoffs. Currently, mpFormulaPy provides the following contexts:

Arbitrary-precision arithmetic (mp)

A faster Cython-based version of mp (used by default in Sage, and currently only available there)

Arbitrary-precision interval arithmetic (iv)

Double-precision arithmetic using Python's builtin float and complex types (fp)

Most global functions in the global mpFormulaPy namespace are actually methods of the mp context. This fact is usually transparent to the user, but sometimes shows up in the form of an initial parameter called 'ctx' visible in the help for the function:

---

```
>>> import mpFormulaPy
>>> help(mpFormulaPy.fsum)
Help on method fsum in module mpFormulaPy.ctx_mp_python:
fsum(ctx, terms, absolute=False, squared=False) method of
    mpFormulaPy.ctx_mp.MPContext ins
Calculates a sum containing a finite number of terms (for infinite
series, see :func:`~mpFormulaPy.nsum`). The terms will be converted to
...
```

---

The following operations are equivalent:

---

```
>>> mpFormulaPy.mp.dps = 15; mpFormulaPy.mp.pretty = False
>>> mpFormulaPy.fsum([1,2,3])
mpf('6.0')
>>> mpFormulaPy.mp.fsum([1,2,3])
mpf('6.0')
```

---

The corresponding operation using the fp context:

---

```
>>> mpFormulaPy.fp.fsum([1,2,3])
6.0
```

---

### 4.1.6 Common interface

ctx.mpf creates a real number:

---

```
>>> from mpFormulaPy import mp, fp
>>> mp.mpf(3)
mpf('3.0')
>>> fp.mpf(3)
3.0
```

---

ctx.mpc creates a complex number:

---

```
>>> mp.mpc(2,3)
```

---

```
mpc(real='2.0', imag='3.0')
>>> fp.mpc(2,3)
(2+3j)
```

---

ctx.matrix creates a matrix:

---

```
>>> mp.matrix([[1,0],[0,1]])
matrix(
[[1.0, 0.0],
 [0.0, 1.0]])
>>> _[0,0]
mpf('1.0')
>>> fp.matrix([[1,0],[0,1]])
matrix(
[[1.0, 0.0],
 [0.0, 1.0]])
>>> _[0,0]
1.0
```

---

ctx.prec holds the current precision (in bits):

---

```
>>> mp.prec
53
>>> fp.prec
53
```

---

ctx.dps holds the current precision (in digits):

---

```
>>> mp.dps
15
>>> fp.dps
15
```

---

ctx.pretty controls whether objects should be pretty-printed automatically by repr(). Prettyprinting for mp numbers is disabled by default so that they can clearly be distinguished from Python numbers and so that eval(repr(x)) == x works:

---

```
>>> mp.mpf(3)
mpf('3.0')
>>> mpf = mp.mpf
>>> eval(repr(mp.mpf(3)))
mpf('3.0')
>>> mp.pretty = True
>>> mp.mpf(3)
3.0
>>> fp.matrix([[1,0],[0,1]])
matrix(
[[1.0, 0.0],
 [0.0, 1.0]])
>>> fp.pretty = True
>>> fp.matrix([[1,0],[0,1]])
[1.0 0.0]
[0.0 1.0]
```

---

```
>>> fp.pretty = False
>>> mp.pretty = False
```

---

### 4.1.7 Arbitrary-precision floating-point (mp)

The mp context is what most users probably want to use most of the time, as it supports the most functions, is most well-tested, and is implemented with a high level of optimization. Nearly all examples in this documentation use mp functions.

See Basic usage for a description of basic usage.

### 4.1.8 Arbitrary-precision interval arithmetic (iv)

The iv.mpf type represents a closed interval  $[a, b]$ ; that is, the set  $\{x : a \leq x \leq b\}$ , where  $a$  and  $b$  are arbitrary-precision floating-point values, possibly  $\pm\infty$ . The iv.mpc type represents a rectangular complex interval  $[a, b] + [c, d]i$ ; that is, the set  $\{z = x + iy : a \leq x \leq b \wedge c \leq y \leq d\}$ .

Interval arithmetic provides rigorous error tracking. If  $f$  is a mathematical function and  $\hat{f}$  is its interval arithmetic version, then the basic guarantee of interval arithmetic is that  $f(v) \subseteq \hat{f}(v)$  for any input interval  $v$ . Put differently, if an interval represents the known uncertainty for a fixed number, any sequence of interval operations will produce an interval that contains what would be the result of applying the same sequence of operations to the exact number. The principal drawbacks of interval arithmetic are speed (iv arithmetic is typically at least two times slower than mp arithmetic) and that it sometimes provides far too pessimistic bounds.

Note: The support for interval arithmetic in mpFormulaPy is still experimental, and many functions do not yet properly support intervals. Please use this feature with caution.

Intervals can be created from single numbers (treated as zero-width intervals) or pairs of endpoint numbers. Strings are treated as exact decimal numbers. Note that a Python float like 0.1 generally does not represent the same number as its literal; use '0.1' instead:

---

```
>>> from mpFormulaPy import iv
>>> iv.dps = 15; iv.pretty = False
>>> iv.mpf(3)
mpf('3.0', '3.0')
>>> print iv.mpf(3)
[3.0, 3.0]
>>> iv.pretty = True
>>> iv.mpf([2,3])
[2.0, 3.0]
>>> iv.mpf(0.1) # probably not intended
[0.100000000000000555, 0.100000000000000555]
>>> iv.mpf('0.1') # good, gives a containing interval
[0.0999999999999991673, 0.100000000000000555]
>>> iv.mpf(['0.1', '0.2'])
[0.0999999999999991673, 0.200000000000000111]
```

---

The fact that '0.1' results in an interval of nonzero width indicates that 1/10 cannot be represented using binary floating-point numbers at this precision level (in fact, it cannot be represented exactly at any precision).

Intervals may be infinite or half-infinite:

---

```
>>> print 1 / iv.mpf([2, 'inf'])
[0.0, 0.5]
```

---

The equality testing operators `==` and `!=` check whether their operands are identical as intervals; that is, have the same endpoints. The ordering operators `<`, `<=`, `>` and `>=` permit inequality testing using triple-valued logic: a guaranteed inequality returns `True` or `False` while an indeterminate inequality returns `None`:

---

```
>>> iv.mpf([1,2]) == iv.mpf([1,2])
True
>>> iv.mpf([1,2]) != iv.mpf([1,2])
False
>>> iv.mpf([1,2]) <= 2
True
>>> iv.mpf([1,2]) > 0
True
>>> iv.mpf([1,2]) < 1
False
>>> iv.mpf([1,2]) < 2 # returns None
>>> iv.mpf([2,2]) < 2
False
>>> iv.mpf([1,2]) <= iv.mpf([2,3])
True
>>> iv.mpf([1,2]) < iv.mpf([2,3]) # returns None
>>> iv.mpf([1,2]) < iv.mpf([-1,0])
False
```

---

The `in` operator tests whether a number or interval is contained in another interval:

---

```
>>> iv.mpf([0,2]) in iv.mpf([0,10])
True
>>> 3 in iv.mpf([-inf, 0])
False
```

---

Intervals have the properties `.a`, `.b` (endpoints), `.mid`, and `.delta` (width):

---

```
>>> x = iv.mpf([2, 5])
>>> x.a
[2.0, 2.0]
>>> x.b
[5.0, 5.0]
>>> x.mid
[3.5, 3.5]
>>> x.delta
[3.0, 3.0]
```

---

Some transcendental functions are supported:

---

```
>>> iv.dps = 15
>>> mp.dps = 15
>>> iv.mpf([0.5,1.5]) ** iv.mpf([0.5, 1.5])
```

```
[0.35355339059327373086, 1.837117307087383633]
>>> iv.exp(0)
[1.0, 1.0]
>>> iv.exp([-inf, 'inf'])
[0.0, +inf]
>>>
>>> iv.exp([-inf, 0])
[0.0, 1.0]
>>> iv.exp([0, 'inf'])
[1.0, +inf]
>>> iv.exp([0, 1])
[1.0, 2.7182818284590455349]
>>>
>>> iv.log(1)
[0.0, 0.0]
>>> iv.log([0, 1])
[-inf, 0.0]
>>> iv.log([0, 'inf'])
[-inf, +inf]
>>> iv.log(2)
[0.69314718055994528623, 0.69314718055994539725]
>>>
>>> iv.sin([100, 'inf'])
[-1.0, 1.0]
>>> iv.cos([-0.1, '0.1'])
[0.99500416527802570954, 1.0]
```

Interval arithmetic is useful for proving inequalities involving irrational numbers. Naive use of mp arithmetic may result in wrong conclusions, such as the following:

```
>>> mp.dps = 25
>>> x = mp.exp(mp.pi*mp.sqrt(163))
>>> y = mp.mpf(640320**3+744)
>>> print x
262537412640768744.0000001
>>> print y
262537412640768744.0
>>> x > y
True
```

But the correct result is  $e^{\pi\sqrt{163}} < 262537412640768744$ , as can be seen by increasing the precision:

```
>>> mp.dps = 50
>>> print mp.exp(mp.pi*mp.sqrt(163))
262537412640768743.99999999999925007259719818568888
```

With interval arithmetic, the comparison returns `None` until the precision is large enough for  $x - y$  to have a definite sign:

```
>>> iv.dps = 15
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
>>> iv.dps = 30
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
```

---

```
>>> iv.dps = 60
>>> iv.exp(iv.pi*iv.sqrt(163)) > (640320**3+744)
False
>>> iv.dps = 15
```

---

#### 4.1.9 Fast low-precision arithmetic (fp)

Although mpFormulaPy is generally designed for arbitrary-precision arithmetic, many of the high-level algorithms work perfectly well with ordinary Python float and complex numbers, which use hardware double precision (on most systems, this corresponds to 53 bits of precision).

Whereas the global functions (which are methods of the mp object) always convert inputs to mpFormulaPy numbers, the fp object instead converts them to float or complex, and in some cases employs basic functions optimized for double precision. When large amounts of function evaluations (numerical integration, plotting, etc) are required, and when fp arithmetic provides sufficient accuracy, this can give a significant speedup over mp arithmetic.

To take advantage of this feature, simply use the fp prefix, i.e. write fp.func instead of func or mp.func:

---

```
>>> u = fp.erfc(2.5)
>>> print u
0.000406952017445
>>> type(u)
<type 'float'>
>>> mp.dps = 15
>>> print mp.erfc(2.5)
0.000406952017444959
>>> fp.matrix([[1,2],[3,4]]) ** 2
matrix(
[[7.0, 10.0],
[15.0, 22.0]])
>>>
>>> type(_[0,0])
<type 'float'>
>>> print fp.quad(fp.sin, [0, fp.pi]) # numerical integration
2.0
```

---

The fp context wraps Python's math and cmath modules for elementary functions. It supports both real and complex numbers and automatically generates complex results for real inputs (math raises an exception):

---

```
>>> fp.sqrt(5)
2.23606797749979
>>> fp.sqrt(-5)
2.23606797749979j
>>> fp.sin(10)
-0.5440211108893698
>>> fp.power(-1, 0.25)
(0.7071067811865476+0.7071067811865475j)
>>> (-1) ** 0.25
Traceback (most recent call last):
```

```
...
ValueError: negative number cannot be raised to a fractional power
```

---

The prec and dps attributes can be changed (for interface compatibility with the mp context) but this has no effect:

---

```
>>> fp.prec
53
>>> fp.dps
15
>>> fp.prec = 80
>>> fp.prec
53
>>> fp.dps
15
```

---

Due to intermediate rounding and cancellation errors, results computed with fp arithmetic may be much less accurate than those computed with mp using an equivalent precision (mp.prec = 53), since the latter often uses increased internal precision. The accuracy is highly problem-dependent: for some functions, fp almost always gives 14-15 correct digits; for others, results can be accurate to only 2-3 digits or even completely wrong. The recommended use for fp is therefore to speed up large-scale computations where accuracy can be verified in advance on a subset of the input set, or where results can be verified afterwards.

## 4.2 Precision and representation issues

Most of the time, using mpFormulaPy is simply a matter of setting the desired precision and entering a formula. For verification purposes, a quite (but not always!) reliable technique is to calculate the same thing a second time at a higher precision and verifying that the results agree.

To perform more advanced calculations, it is important to have some understanding of how mpFormulaPy works internally and what the possible sources of error are. This section gives an overview of arbitrary-precision binary floating-point arithmetic and some concepts from numerical analysis.

The following concepts are important to understand:

The main sources of numerical errors are rounding and cancellation, which are due to the use of finite-precision arithmetic, and truncation or approximation errors, which are due to approximating infinite sequences or continuous functions by a finite number of samples.

Errors propagate between calculations. A small error in the input may result in a large error in the output.

Most numerical algorithms for complex problems (e.g. integrals, derivatives) give wrong answers for sufficiently ill-behaved input. Sometimes virtually the only way to get a wrong answer is to design some very contrived input, but at other times the chance of accidentally obtaining a wrong result even for reasonable-looking input is quite high.

Like any complex numerical software, mpFormulaPy has implementation bugs. You should be reasonably suspicious about any results computed by mpFormulaPy, even those it claims to be able to compute correctly! If possible, verify results analytically, try different algorithms, and cross-compare with other software.

### 4.2.1 Precision, error and tolerance

The following terms are common in this documentation:

Precision (or working precision) is the precision at which floating-point arithmetic operations are performed.

Error is the difference between a computed approximation and the exact result.

Accuracy is the inverse of error.

Tolerance is the maximum error (or minimum accuracy) desired in a result.

Error and accuracy can be measured either directly, or logarithmically in bits or digits. Specifically, if a  $\hat{y}$  is an approximation for  $y$ , then

(Direct) absolute error =  $|\hat{y} - y|$

(Direct) relative error =  $|\hat{y} - y||y|^{-1}$

(Direct) absolute accuracy =  $|\hat{y} - y|^{-1}$

(Direct) relative accuracy =  $|\hat{y} - y|^{-1}|y|$

(Logarithmic) absolute error =  $\log_b |\hat{y} - y|$

(Logarithmic) relative error =  $\log_b |\hat{y} - y| - \log_b |y|$

(Logarithmic) absolute accuracy =  $-\log_b |\hat{y} - y|$

(Logarithmic) relative accuracy =  $-\log_b |\hat{y} - y| - \log_b |y|$

where  $b = 2$  and  $b = 10$  for bits and digits respectively. Note that:

The logarithmic error roughly equals the position of the first incorrect bit or digit.

The logarithmic accuracy roughly equals the number of correct bits or digits in the result.

These definitions also hold for complex numbers, using  $|a + bi| = \sqrt{a^2 + b^2}$ .

Full accuracy means that the accuracy of a result at least equals prec-1, i.e. it is correct except possibly for the last bit.

### 4.2.2 Representation of numbers

Mpmath uses binary arithmetic. A binary floating-point number is a number of the form  $man \times 2^{exp}$  where both man (the mantissa) and exp (the exponent) are integers. Some examples of floating-point numbers are given in the following table.

Number	Mantissa	Exponent
--------	----------	----------

3	3	0
10	5	1
-16	-1	4
1.25	5	-2

The representation as defined so far is not unique; one can always multiply the mantissa by 2 and subtract 1 from the exponent with no change in the numerical value. In mpFormulaPy, numbers are always normalized so that man is an odd number, with the exception of zero which is always taken to have man = exp = 0. With these conventions, every representable number has a unique representation. (Mpmath does not currently distinguish between positive and negative zero.)

Simple mathematical operations are now easy to define. Due to uniqueness, equality testing of two numbers simply amounts to separately checking equality of the mantissas and the exponents. Multiplication of nonzero numbers is straightforward:  $(m2^e) \times (n2^f) = (mn) \times 2^{e+f}$ . Addition is a bit more involved: we first need to multiply the mantissa of one of the operands by a suitable power of 2 to obtain equal exponents.

More technically, mpFormulaPy represents a floating-point number as a 4-tuple (sign, man, exp, bc) where sign is 0 or 1 (indicating positive vs negative) and the mantissa is nonnegative; bc (bitcount) is the size of the absolute value of the mantissa as measured in bits. Though redundant, keeping a separate sign field and explicitly keeping track of the bitcount significantly speeds up arithmetic (the bitcount, especially, is frequently needed but slow to compute from scratch due to the lack of a Python built-in function for the purpose).

Contrary to popular belief, floating-point numbers do not come with an inherent 'small uncertainty', although floating-point arithmetic generally is inexact. Every binary floating-point number is an exact rational number. With arbitrary-precision integers used for the mantissa and exponent, floating-point numbers can be added, subtracted and multiplied exactly. In particular, integers and integer multiples of  $1/2$ ,  $1/4$ ,  $1/8$ ,  $1/16$ , etc. can be represented, added and multiplied exactly in binary floating-point arithmetic.

Floating-point arithmetic is generally approximate because the size of the mantissa must be limited for efficiency reasons. The maximum allowed width (bitcount) of the mantissa is called the precision or prec for short. Sums and products of floating-point numbers are exact as long as

the absolute value of the mantissa is smaller than  $2^{prec}$ . As soon as the mantissa becomes larger than this, it is truncated to contain at most prec bits (the exponent is incremented accordingly to preserve the magnitude of the number), and this operation introduces a rounding error. Division is also generally inexact; although we can add and multiply exactly by setting the precision high enough, no precision is high enough to represent for example  $1/3$  exactly (the same obviously applies for roots, trigonometric functions, etc).

The special numbers `+inf`, `-inf` and `nan` are represented internally by a zero mantissa and a nonzero exponent.

Mpmath uses arbitrary precision integers for both the mantissa and the exponent, so numbers can be as large in magnitude as permitted by the computer's memory. Some care may be necessary when working with extremely large numbers. Although standard arithmetic operators are safe, it is for example futile to attempt to compute the exponential function of  $10^{100000}$ . Mpmath does not complain when asked to perform such a calculation, but instead chugs away on the problem to the best of its ability, assuming that computer resources are infinite. In the worst case, this will be slow and allocate a huge amount of memory; if entirely impossible Python will at some point raise `OverflowError`: long int too large to convert to int.

For further details on how the arithmetic is implemented, refer to the `mpFormulaPy` source code. The basic arithmetic operations are found in the `libmp` directory; many functions there are commented extensively.

### 4.2.3 Decimal issues

Mpmath uses binary arithmetic internally, while most interaction with the user is done via the decimal number system. Translating between binary and decimal numbers is a somewhat subtle matter; many Python novices run into the following 'bug' (addressed in the General Python FAQ):

---

```
>>> 1.2 - 1.0
0.1999999999999996
```

---

Decimal fractions fall into the category of numbers that generally cannot be represented exactly in binary floating-point form. For example, none of the numbers  $0.1$ ,  $0.01$ ,  $0.001$  has an exact representation as a binary floating-point number. Although `mpFormulaPy` can approximate decimal fractions with any accuracy, it does not solve this problem for all uses; users who need exact decimal fractions should look at the `decimal` module in Python's standard library (or perhaps use `fractions`, which are much faster).

With `prec` bits of precision, an arbitrary number can be approximated relatively to within  $2^{-prec}$ , or within  $10^{-dps}$  for `dps` decimal digits. The equivalent values for `prec` and `dps` are therefore related proportionally via the factor  $C = \log(1)/\log(2)$ , or roughly 3.32. For example, the standard (binary) precision in `mpFormulaPy` is 53 bits, which corresponds to a decimal precision of 15.95 digits.

More precisely, `mpFormulaPy` uses the following formulas to translate between `prec` and `dps`:

---

```
dps(prec) = max(1, int(round(int(prec) / C - 1)))
prec(dps) = max(1, int(round((int(dps) + 1) * C)))
```

---

Note that the `dps` is set 1 decimal digit lower than the corresponding binary precision. This is done to hide minor rounding errors and artifacts resulting from binary-decimal conversion. As a result, `mpFormulaPy` interprets 53 bits as giving 15 digits of decimal precision, not 16.

The `dps` value controls the number of digits to display when printing numbers with `str()`, while the decimal precision used by `repr()` is set two or three digits higher. For example, with 15 `dps` we have:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15
>>> str(pi)
'3.14159265358979'
>>> repr(+pi)
"mpf('3.1415926535897931')"
```

---

The extra digits in the output from `repr` ensure that `x == eval(repr(x))` holds, i.e. that numbers can be converted to strings and back losslessly.

It should be noted that precision and accuracy do not always correlate when translating between binary and decimal. As a simple example, the number 0.1 has a decimal precision of 1 digit but is an infinitely accurate representation of 1/10. Conversely, the number  $2^{-50}$  has a binary representation with 1 bit of precision that is infinitely accurate; the same number can actually be represented exactly as a decimal, but doing so requires 35 significant digits:

---

```
0.00000000000000088817841970012523233890533447265625
```

---

All binary floating-point numbers can be represented exactly as decimals (possibly requiring many digits), but the converse is false.

#### 4.2.4 Correctness guarantees

Basic arithmetic operations (with the `mp` context) are always performed with correct rounding. Results that can be represented exactly are guaranteed to be exact, and results from single inexact operations are guaranteed to be the best possible rounded values. For higher-level operations, `mpFormulaPy` does not generally guarantee correct rounding. In general, `mpFormulaPy` only guarantees that it will use at least the user-set precision to perform a given calculation. The user may have to manually set the working precision higher than the desired accuracy for the result, possibly much higher.

Functions for evaluation of transcendental functions, linear algebra operations, numerical integration, etc., usually automatically increase the working precision and use a stricter tolerance to give a correctly rounded result with high probability: for example, at 50 bits the temporary precision might be set to 70 bits and the tolerance might be set to 60 bits. It can often be assumed that such functions return values that have full accuracy, given inputs that are exact (or sufficiently precise approximations of exact values), but the user must exercise judgement about whether to trust `mpFormulaPy`.

The level of rigor in `mpFormulaPy` covers the entire spectrum from 'always correct by design' through 'nearly always correct' and 'handling the most common errors' to 'just computing blindly and hoping for the best'. Of course, a long-term development goal is to successively increase the rigor where possible. The following list might give an idea of the current state.

Operations that are correctly rounded:

Addition, subtraction and multiplication of real and complex numbers. Division and square roots of real numbers.

Powers of real numbers, assuming sufficiently small integer exponents (huge powers are rounded in the right direction, but possibly farther than necessary).

Conversion from decimal to binary, for reasonably sized numbers (roughly  $10^{-100}$  between and  $10^{100}$ ).

Typically, transcendental functions for exact input-output pairs.

Operations that should be fully accurate (however, the current implementation may be based on a heuristic error analysis):

Radix conversion (large or small numbers).

Mathematical constants like  $\pi$ .

Both real and imaginary parts of  $\exp$ ,  $\cos$ ,  $\sin$ ,  $\cosh$ ,  $\sinh$ ,  $\log$ .

Other elementary functions (the largest of the real and imaginary part).

The gamma and log-gamma functions (the largest of the real and the imaginary part; both, when close to real axis).

Some functions based on hypergeometric series (the largest of the real and imaginary part).

Correctness of root-finding, numerical integration, etc. largely depends on the well-behavedness of the input functions. Specific limitations are sometimes noted in the respective sections of the documentation.

#### 4.2.5 Double precision emulation

On most systems, Python's float type represents an IEEE 754 double precision number, with a precision of 53 bits and rounding-to-nearest. With default precision (`mp.prec = 53`), the `mpf` type roughly emulates the behavior of the float type. Sources of incompatibility include the following:

In hardware floating-point arithmetic, the size of the exponent is restricted to a fixed range: regular Python floats have a range between roughly  $10^{-300}$  and  $10^{300}$ ). Mpmath does not emulate overflow or underflow when exponents fall outside this range.

On some systems, Python uses 80-bit (extended double) registers for floating-point operations. Due to double rounding, this makes the float type less accurate. This problem is only known to occur with Python versions compiled with GCC on 32-bit systems.

Machine floats very close to the exponent limit round subnormally, meaning that they lose accuracy (Python may raise an exception instead of rounding a float subnormally).

Mpmath is able to produce more accurate results for transcendental functions.

## 4.3 Conversion and printing

### 4.3.1 convert()

mpFormulaPy.mpFormulaify(x, strings=True)

Converts x to an mpf or mpc. If x is of type mpf, mpc, int, float, complex, the conversion will be performed losslessly.

If x is a string, the result will be rounded to the present working precision. Strings representing fractions or complex numbers are permitted.

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> convert(3.5)
mpf('3.5')
>>> convert('2.1')
mpf('2.100000000000001')
>>> convert('3/4')
mpf('0.75')
>>> convert('2+3j')
mpc(real='2.0', imag='3.0')
```

---

### 4.3.2 nstr()

mpFormulaPy.nstr(x, n=6, \*\*kwargs)

Convert an mpf or mpc to a decimal string literal with n significant digits. The small default value for n is chosen to make this function useful for printing collections of numbers (lists, matrices, etc.).

If x is a list or tuple, nstr() is applied recursively to each element. For unrecognized classes, nstr() simply returns str(x).

The companion function nprint() prints the result instead of returning it.

---

```
>>> from mpFormulaPy import *
>>> nstr([+pi, ldexp(1,-500)])
[3.14159, 3.05494e-151]
>>> nprint([+pi, ldexp(1,-500)])
[3.14159, 3.05494e-151]
```

---

## 4.4 Rounding

Function **ceil**(*x* As *mpNum*) As *mpNum*

The function `ceil` returns a number down to the nearest integer.

## Parameter:

$x$ : A real number.

Computes the ceiling of  $x$ ,  $\lceil x \rceil$ , defined as the smallest integer greater than or equal to  $x$ :

```
>>> from mpFormulaPy import *
>>> mp.pretty = False
>>> ceil(3.5)
mpf('4.0')
```

The ceiling function is defined for complex numbers and acts on the real and imaginary parts separately:

```
>>> ceil(3.25+4.75j)
mpc(real='4.0', imag='5.0')
```

See notes about rounding for `floor()`.

#### 4.4.1 Next lower or equal integer: Floor( $x$ )

Function **floor**(*x* As *mpNum*) As *mpNum*

The function `floor` returns a number down to the nearest integer.

## Parameter:

$x$ : A real number.

Computes the floor of  $x$ ,  $\lfloor x \rfloor$ , defined as the largest integer less than or equal to  $x$ :

```
>>> from mpFormulaPy import *
>>> mp.pretty = False
>>> floor(3.5)
mpf('3.0')
```

Note: `floor()`, `ceil()` and `nint()` return a floating-point number, not a Python `int`. If `is` is too large to be represented exactly at the present working precision, the result will be rounded, not necessarily in the direction implied by the mathematical definition of the function.

To avoid rounding, use `prec=0`:

```
>>> mp.dps = 15
>>> print(int(floor(10**30+1)))
10000000000000000000019884624838656
>>> print(int(floor(10**30+1, prec=0)))
10000000000000000000000000000000000001
```

The floor function is defined for complex numbers and acts on the real and imaginary parts separately:

---

```
>>> floor(3.25+4.75j)
mpc(real='3.0', imag='4.0')
```

---

#### 4.4.2 Next integer, rounded toward zero: $\text{Trunc}(x)$

---

Function **nint**(*x* As *mpNum*) As *mpNum*

---

The function **nint** returns a number down to the nearest integer.

**Parameter:**

*x*: A real number.

Evaluates the nearest integer function,  $\text{nint}(x)$ . This gives the nearest integer to *x*; on a tie, it gives the nearest even integer:

---

```
>>> from mpFormulaPy import *
>>> mp.pretty = False
>>> nint(3.2)
mpf('3.0')
>>> nint(3.8)
mpf('4.0')
>>> nint(3.5)
mpf('4.0')
>>> nint(4.5)
mpf('4.0')
```

---

The nearest integer function is defined for complex numbers and acts on the real and imaginary parts separately:

---

```
>>> nint(3.25+4.75j)
mpc(real='3.0', imag='5.0')
```

---

See notes about rounding for  $\text{floor}()$ .

#### 4.4.3 Fractional Part

---

Function **frac**(*x* As *mpNum*) As *mpNum*

---

The function **frac** returns the fractional part of *x*.

**Parameter:**

*x*: A colpmex or real number.

Gives the fractional part of *x*, defined as  $\text{frac}(x) = x - \lfloor x \rfloor$  (see  $\text{floor}()$ ). In effect, this computes *x* modulo 1, or  $x + n$  where  $n \in \mathbb{Z}$  is such that  $x + n \in [0, 1)$ :

---

```
>>> from mpFormulaPy import *
>>> mp.pretty = False
>>> frac(1.25)
mpf('0.25')
>>> frac(3)
mpf('0.0')
```

---

---

```
>>> frac(-1.25)
mpf('0.75')
```

---

For a complex number, the fractional part function applies to the real and imaginary parts separately:

---

```
>>> frac(2.25+3.75j)
mpc(real='0.25', imag='0.75')
```

---

Plotted, the fractional part function gives a sawtooth wave. The Fourier series coefficients have a simple form:

---

```
>>> mp.dps = 15
>>> nprint(fourier(lambda x: frac(x)-0.5, [0,1], 4))
([0.0, 0.0, 0.0, 0.0, 0.0], [0.0, -0.31831, -0.159155, -0.106103, -0.0795775])
>>> nprint([-1/(pi*k) for k in range(1,5)])
[-0.31831, -0.159155, -0.106103, -0.0795775]
```

---

Note: The fractional part is sometimes defined as a symmetric function, i.e. returning  $-\text{frac}(-x)$  if  $x < 0$ . This convention is used, for instance, by Mathematica's FractionalPart.

## 4.5 Components of Real and Complex Numbers

### 4.5.1 Number generated from Significand and Exponent: Ldexp( $x, y$ )

---

Function **ldexp**( $x$  As *mpNum*,  $y$  As *mpNum*) As *mpNum*

---

The function **ldexp** returns  $x \cdot 2^y$

**Parameters:**

$x$ : A real number.

$y$ : A real number.

Returns the result of multiplying  $x$  (the significand) by 2 raised to the power of  $y$  (the exponent):

$$\text{Ldexp}(x, y) = x \cdot 2^y.$$

`mpFormulaPy.ldexp(x, n)`

Computes  $x2^n$  efficiently. No rounding is performed. The argument  $x$  must be a real floating-point number (or possible to convert into one) and  $n$  must be a Python int.

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> ldexp(1, 10)
mpf('1024.0')
>>> ldexp(1, -3)
mpf('0.125')
```

---

### 4.5.2 Significand and Exponent: Frexp( $x$ )

---

Function **frexp**( $x$  As *mpNum*) As *mpNumList*

---

The function **frexp** returns returns simultaneously significand and exponent of  $x$

**Parameter:**

$x$ : A real number.

Set  $\text{exp}$  (formally, the value pointed to by  $\text{exp}$ ) and  $y$  such that  $0.5 \leq |y| < 1$  and  $y \times 2^{\text{exp}}$  equals  $x$  rounded to the precision of  $y$ , using the given rounding mode. If  $x$  is zero, then  $y$  is set to a zero of the same sign and  $\text{exp}$  is set to 0. If  $x$  is NaN or an infinity, then  $y$  is set to the same value and  $\text{exp}$  is undefined.

`mpFormulaPy.frexp(x, n)`

Given a real number  $x$ , returns  $(y, n)$  with  $y \in [0.5, 1)$ ,  $n$  a Python integer, and such that  $x = y2^n$ . No rounding is performed.

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> frexp(7.5)
(mpf('0.9375'), 3)
```

---

### 4.5.3 Building a Complex Number from Real Components

---

#### Function **mpc**(*x* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function **mpc** returns a complex number  $z$  build from the real components  $x$  and  $y$  as  $z = x + iy$ .

**Parameters:**

*x*: A real number.

*y*: A real number.

### 4.5.4 Representations of Complex Numbers

---

#### Function **polar**(*z* As *mpNum*) As *mpNum*

---

The function **polar** returns Returns the polar representation of the complex number  $z$ .

**Parameter:**

*z*: A complex or real number.

Returns the polar representation of the complex number  $z$  as a pair  $(r, \phi)$  such that  $z = re^{i\phi}$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> polar(-2)
(2.0, 3.14159265358979)
>>> polar(3-4j)
(5.0, -0.927295218001612)
```

---



---

#### Function **rect**(*x* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function **rect** returns the complex number represented by polar coordinates  $(r, \phi)$ .

**Parameters:**

*x*: A real number.

*y*: A real number.

Returns the complex number represented by polar coordinates  $(r, \phi)$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> chop(rect(2, pi))
-2.0
>>> rect(sqrt(2), -pi/4)
(1.0 - 1.0j)
```

---

### 4.5.5 Real Component

---

#### Function **re**(*z* As *mpNum*) As *mpNum*

---

The function **re** returns the real part of  $x$ ,  $\Re(x)$ .

**Parameter:**

$z$ : A complex number.

Unlike  $x.\text{real}$ ,  $\text{re}()$  converts  $x$  to a mpFormulaPy number:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> re(3)
mpf('3.0')
>>> re(-1+4j)
mpf('-1.0')
```

---

## 4.5.6 Imaginary Component

---

### Function **im**( $z$ As mpNum) As mpNum

---

The function  $\text{im}$  returns the imaginary part of  $x$ ,  $\Im(x)$ .

**Parameter:**

$z$ : A complex number.

Unlike  $x.\text{imag}$ ,  $\text{im}()$  converts  $x$  to a mpFormulaPy number:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> im(3)
mpf('0.0')
>>> im(-1+4j)
mpf('4.0')
```

---

## 4.5.7 Absolute Value

---

### Function **abs**( $z$ As mpNum) As mpNum

---

The function  $\text{abs}$  returns the absolute value of  $z = x + iy$

**Parameter:**

$z$ : A real or complex number.

---

### Function **fabs**( $z$ As mpNum) As mpNum

---

The function  $\text{fabs}$  returns the absolute value of  $z = x + iy$

**Parameter:**

$z$ : A real or complex number.

Returns the absolute value of  $x$ ,  $|x|$ . Unlike  $\text{abs}()$ ,  $\text{fabs}()$  converts non-mpFormulaPy numbers (such as int) into mpFormulaPy numbers:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fabs(3)
```

---

```
mpf('3.0')
>>> fabs(-3)
mpf('3.0')
>>> fabs(3+4j)
mpf('5.0')
```

---

## 4.5.8 Argument

---

### Function **arg(z As mpNum) As mpNum**

---

The function **arg** returns the argument of  $z = x + iy$

**Parameter:**

$z$ : A complex number.

---

### Function **phase(z As mpNum) As mpNum**

---

The function **phase** returns the argument of  $z = x + iy$

**Parameter:**

$z$ : A complex number.

Computes the complex argument (phase) of  $x$ , defined as the signed angle between the positive real axis and in the complex plane:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> arg(3)
0.0
>>> arg(3+3j)
0.785398163397448
>>> arg(3j)
1.5707963267949
>>> arg(-3)
3.14159265358979
>>> arg(-3j)
-1.5707963267949
```

---

The angle is defined to satisfy  $-\pi < \arg(x) \leq \pi$  and with the sign convention that a nonnegative imaginary part results in a nonnegative argument.

The value returned by **arg()** is an **mpf** instance.

## 4.5.9 Sign

---

### Function **sign(x As mpNum) As mpNum**

---

The function **sign** returns the value of the sign of  $x$ ,  $\text{sign}(x)$ .

**Parameter:**

$x$ : A real or complex number.

The sign of  $x$  is defined as  $\text{sign}(x) = x/|x|$  (with the special case  $\text{sign}(0) = 0$ ):

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> sign(10)
mpf('1.0')
>>> sign(-10)
mpf('-1.0')
>>> sign(0)
mpf('0.0')
```

---

Note that the sign function is also defined for complex numbers, for which it gives the projection onto the unit circle:

---

```
>>> mp.dps = 15; mp.pretty = True
>>> sign(1+j)
(0.707106781186547 + 0.707106781186547j)
```

---

#### 4.5.10 Conjugate

---

Function **conj**(*z* As *mpNum*) As *mpNum*

---

The function **conj** returns the complex conjugate of  $z$ ,  $\bar{z}$

**Parameter:**

*z*: A complex number.

Unlike *x.conjugate()*, *im()* converts *x* to a *mpFormulaPy* number:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> conj(3)
mpf('3.0')
>>> conj(-1+4j)
mpc(real='-1.0', imag='-4.0')
```

---

## 4.6 Arithmetic operations

See also `mpFormulaPy.sqrt()`, `mpFormulaPy.exp()` etc., listed in Powers and logarithms

### 4.6.1 Addition and Sum

---

Operator `+`

---

Number.Function **.Plus**(`a` As `mpNum`, `b` As `mpNum`) As `mpNum`

---

The binary operator `+` is used to return the sum of the 2 operands  $a$  and  $b$ , and assign the result to  $c$ :  $c = a + b$ .

For languages not supporting operator overloading, the function `.Plus` can be used to achieve the same:  $c = a$ .`Plus`( $b$ )

The operator `+` returns the sum of  $z1$  and  $z2$ .

---

Function **fadd**(`x` As `mpNum`, `y` As `mpNum`, **Keywords** As `String`) As `mpNum`

---

The function `fadd` returns the sum of the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode..

**Parameters:**

`x`: A complex number.

`y`: A complex number.

**Keywords:** `prec`, `dps`, `exact`, `rounding`.

`mpFormulaPy.fadd(ctx, x, y, **kwargs)`

Adds the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode.

The default precision is the working precision of the context. You can specify a custom precision in bits by passing the `prec` keyword argument, or by providing an equivalent decimal precision with the `dps` keyword argument. If the precision is set to `+inf`, or if the flag `exact=True` is passed, an exact addition with no rounding is performed.

When the precision is finite, the optional rounding keyword argument specifies the direction of rounding. Valid options are `'n'` for nearest (default), `'f'` for floor, `'c'` for ceiling, `'d'` for down, `'u'` for up.

**Examples**

Using `fadd()` with precision and rounding control:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fadd(2, 1e-20)
mpf('2.0')
>>> fadd(2, 1e-20, rounding='u')
mpf('2.000000000000004')
>>> nprint(fadd(2, 1e-20, prec=100), 25)
2.00000000000000000000000000001
>>> nprint(fadd(2, 1e-20, dps=15), 25)
2.0
```

Exact addition avoids cancellation errors, enforcing familiar laws of numbers such as  $x + y - x = y$ , which do not hold in floating-point arithmetic with finite precision:

```
>>> x, y = mpf(2), mpf('1e-1000')
>>> print(x + y - x)
0.0
>>> print(fadd(x, y, prec=inf) - x)
1.0e-1000
>>> print(fadd(x, y, exact=True) - x)
1.0e-1000
```

Exact addition can be inefficient and may be impossible to perform with large magnitude differences:

### 4.6.2 Sums and Series

Function **fsum**(*terms* As *mpNum*, *Keywords* As *String*) As *mpNum*

The function `fsum` returns the sum of the numbers `x` and `y`, giving a floating-point result, optionally using a custom precision and rounding mode..

## Parameters:

*terms*: a finite number of terms.

*Keywords:* absolute=False, squared=False.

```
mpFormulaPy.fsum(terms, absolute=False, squared=False)
```

Calculates a sum containing a finite number of terms (for infinite series, see `nsum()`). The terms will be converted to `mpFormulaPy` numbers. For `len(terms) < 2`, this function is generally faster and produces more accurate results than the builtin Python function `sum()`.

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fsum([1, 2, 0.5, 7])
mpf('10.5')
```

With `squared=True` each term is squared, and with `absolute=True` the absolute value of each term is used.

### 4.6.3 Subtraction

---

Operator –

---

Function **.Minus(a As mpNum, b As mpNum) As mpNum**

---

The binary operator – is used to return the difference of the 2 operands  $a$  and  $b$ , and assign the result to  $c$ :  $c = a - b$ .

For languages not supporting operator overloading, the function **.Minus** can be used to achieve the same:  $c = a$ .**Minus**( $b$ )

---

Function **fsub(x As mpNum, y As mpNum, Keywords As String) As mpNum**

---

The function **fsub** returns the sum of the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode..

**Parameters:**

$x$ : A complex number.

$y$ : A complex number.

**Keywords:** prec, dps, exact, rounding.

`mpFormulaPy.fsub(ctx, x, y, **kwargs)`

Subtracts the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of `fadd()` for a detailed description of how to specify precision and rounding.

Examples Using `fsub()` with precision and rounding control:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fsub(2, 1e-20)
mpf('2.0')
>>> fsub(2, 1e-20, rounding='d')
mpf('1.999999999999998')
>>> nprint(fsub(2, 1e-20, prec=100), 25)
1.99999999999999999999999
>>> nprint(fsub(2, 1e-20, dps=15), 25)
2.0
>>> nprint(fsub(2, 1e-20, dps=25), 25)
1.99999999999999999999999
>>> nprint(fsub(2, 1e-20, exact=True), 25)
1.99999999999999999999999
```

---

Exact subtraction avoids cancellation errors, enforcing familiar laws of numbers such as  $x+y-x = y$ , which don't hold in floating-point arithmetic with finite precision:

---

```
>>> x, y = mpf(2), mpf('1e1000')
>>> print(x - y + y)
0.0
>>> print(fsub(x, y, prec=inf) + y)
2.0
>>> print(fsub(x, y, exact=True) + y)
```

2.0

Exact subtraction can be inefficient and may be impossible to perform with large magnitude differences:

#### 4.6.4 Negation

Function **fneg**(*x* As *mpNum*, **Keywords** As *String*) As *mpNum*

The function **fneg** returns the sum of the numbers *x* and *y*, giving a floating-point result, optionally using a custom precision and rounding mode..

## Parameters:

$x$ : A complex number.

**Keywords:** prec, dps, exact, rounding.

```
mpFormulaPy.fneg(ctx, x, **kwargs)
```

Negates the number  $x$ , giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of `fadd()` for a detailed description of how to specify precision and rounding.

## Examples

An mpFormulaPy number is returned:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fneg(2.5)
mpf('-2.5')
>>> fneg(-5+2j)
mpc(real='5.0', imag='-2.0')
```

Precise control over rounding is possible:

```
>>> x = fadd(2, 1e-100, exact=True)
>>> fneg(x)
mpf('-2.0')
>>> fneg(x, rounding='f')
mpf('-2.0000000000000004')
```

Negating with and without roundoff:

## 4.6.5 Multiplication

## Operator \*

```
Function .Times(a As mpNum, b As mpNum) As mpNum
Function .TimesMat(a As mpNum, b As Integer) As mpNum
Function .DotProd(a As mpNum, b As Integer) As mpNum
Function .LSH(a As mpNum, b As Integer) As mpNum
```

The binary operator `*` is used to return the product of the 2 operands `a` and `b`, and assign the result to `c`: `c = a * b`.

For languages not supporting operator overloading, the function `.Times` can be used to achieve the same: `c = a.Times(b)`

Function **fmul**(*x* As *mpNum*, *y* As *mpNum*, **Keywords** As String) As *mpNum*

The function `fmul` returns the sum of the numbers `x` and `y`, giving a floating-point result, optionally using a custom precision and rounding mode..

## Parameters:

$x$ : A complex number.

*y*: A complex number.

*Keywords:* prec, dps, exact, rounding.

```
mpFormulaPy.fmul(ctx, x, y, **kwargs)
```

Multiplies the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of `fadd()` for a detailed description of how to specify precision and rounding.

## Examples

The result is an mpFormulaPy number:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fmul(2, 5.0)
mpf('10.0')
>>> fmul(0.5j, 0.5)
mpc(real='0.0', imag='0.25')
```

## Avoiding roundoff:

---

```
>>> x, y = 10**10+1, 10**15+1
>>> print(x*y)
100000000010000100000000001
>>> print(mpf(x) * mpf(y))
1.0000000001e+25
>>> print(int(mpf(x) * mpf(y)))
10000000001000011026399232
>>> print(int(fmul(x, y)))
10000000001000011026399232
>>> print(int(fmul(x, y, dps=25)))
100000000010000100000000001
>>> print(int(fmul(x, y, exact=True)))
100000000010000100000000001
```

---

Exact multiplication with complex numbers can be inefficient and may be impossible to perform with large magnitude differences between real and imaginary parts:

---

```
>>> x = 1+2j
>>> y = mpc(2, '1e-10000000000000000000000000000000')
>>> fmul(x, y)
mpc(real='2.0', imag='4.0')
>>> fmul(x, y, rounding='u')
mpc(real='2.0', imag='4.0000000000000009')
>>> fmul(x, y, exact=True)
Traceback (most recent call last):
...
OverflowError: the exact result does not fit in memory
```

---

## 4.6.6 Products

---

### Function **fprod**(*factors* As mpNum, *Keywords* As String) As mpNum

---

The function fprod returns a product containing a finite number of factors

**Parameters:**

*factors*: a finite number of factors

*Keywords*: prec, dps, exact, rounding.

mpFormulaPy.fprod(factors)

Calculates a product containing a finite number of factors (for infinite products, see nprod()). The factors will be converted to mpFormulaPy numbers.

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fprod([1, 2, 0.5, 7])
mpf('7.0')
```

---



---

### Function **fdot**(*factors* As mpNum, *Keywords* As String) As mpNum

---

The function fdot returns a product containing a finite number of factors

**Parameters:***factors*: a finite number of factors*Keywords*: prec, dps, exact, rounding.

mpFormulaPy.fdot(A, B=None, conjugate=False)

Computes the dot product of the iterables  $A$  and  $B$ ,

$$\sum_{k=0} A_k B_k \quad (4.6.1)$$

Alternatively, fdot() accepts a single iterable of pairs. In other words, fdot(A,B) and fdot(zip(A,B)) are equivalent. The elements are automatically converted to mpFormulaPy numbers.

With conjugate=True, the elements in the second vector will be conjugated:

$$\sum_{k=0} A_k \overline{B_k} \quad (4.6.2)$$

**Examples**


---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> A = [2, 1.5, 3]
>>> B = [1, -1, 2]
>>> fdot(A, B)
mpf('6.5')
>>> list(zip(A, B))
[(2, 1), (1.5, -1), (3, 2)]
>>> fdot(_)
mpf('6.5')
>>> A = [2, 1.5, 3j]
>>> B = [1+j, 3, -1-j]
>>> fdot(A, B)
mpc(real='9.5', imag='-1.0')
>>> fdot(A, B, conjugate=True)
mpc(real='3.5', imag='-5.0')
```

---

**4.6.7 Multiplication by multiples of 2 (LSH)**The function BITLSHIFT( $n, k$ ) returns the product of  $n$  and  $2^k$ :

$$\text{intLSH}(n, k) = n \times 2^k. \quad (4.6.3)$$

This operation can also be defined as a left shift by  $k$  bits.**4.6.8 Division**

---

Operator /

Function .Div(a As mpNum, b As mpNum) As mpNum

Function .RSH(a As mpNum, b As Integer) As mpNum

The binary operator `/` is used to return the quotient of the 2 operands  $a$  and  $b$ , and assign the result to  $c$ :  $c = a / b$ .

For languages not supporting operator overloading, the function `.Div` can be used to achieve the same:  $c = a.Div(b)$

The function `.DivInt` can be used if the second operand is an integer:  $c = a.DivInt(b)$

---

Function **`fdiv(x As mpNum, y As mpNum, Keywords As String)`** As mpNum

---

The function `fdiv` returns the sum of the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode..

**Parameters:**

$x$ : A complex number.

$y$ : A complex number.

**Keywords:** `prec`, `dps`, `exact`, `rounding`.

`mpFormulaPy.fdiv(ctx, x, y, **kwargs)`

Divides the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of `fadd()` for a detailed description of how to specify precision and rounding.

**Examples**

The result is an `mpFormulaPy` number:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> fdiv(3, 2)
mpf('1.5')
>>> fdiv(2, 3)
mpf('0.6666666666666663')
>>> fdiv(2+4j, 0.5)
mpc(real='4.0', imag='8.0')
```

---

The rounding direction and precision can be controlled:

---

```
>>> fdiv(2, 3, dps=3) # Should be accurate to at least 3 digits
mpf('0.6666259765625')
>>> fdiv(2, 3, rounding='d')
mpf('0.6666666666666663')
>>> fdiv(2, 3, prec=60)
mpf('0.6666666666666667')
>>> fdiv(2, 3, rounding='u')
mpf('0.6666666666666674')
```

---

Checking the error of a division by performing it at higher precision:

---

```
>>> fdiv(2, 3) - fdiv(2, 3, prec=100)
mpf('-3.7007434154172148e-17')
```

---

Unlike `fadd()`, `fmul()`, etc., exact division is not allowed since the quotient of two floating-point numbers generally does not have an exact floating-point representation. (In the future this might be changed to allow the case where the division is actually exact.)

---

```
>>> fdiv(2, 3, exact=True)
Traceback (most recent call last):
...
ValueError: division is not an exact operation
```

---

### 4.6.9 Division by multiples of 2 (RSH)

The function `BITRSHIFT`( $n, k$ ) returns the quotient of  $n$  and  $2^k$ :

$$\text{intRSH}(n, k) = n \div 2^k. \quad (4.6.4)$$

This operation can also be defined as a right shift by  $k$  bits.

### 4.6.10 Modulo

---

Operator **Mod** (VB.NET)

Operator **%** (Python)

---

Function **.Mod(a As mpNum, b As mpNum) As mpNum**

---

The binary operator `mod` is used to return the modulo of the 2 operands  $a$  and  $b$ , and assign the result to  $c$ :  $c = a \bmod b$ .

For languages not supporting operator overloading, the function `.Mod` can be used to achieve the same:  $c = a.\text{Mod}(b)$

Returns the value of  $x - ny$ ,  $n = \lfloor x/y \rfloor$ , i.e. rounded according to the direction `rnd`, where  $n$  is the integer quotient of  $x$  divided by  $y$ , rounded toward zero.

---

Function **fmod(x As mpReal, y As mpReal) As mpReal**

---

The function `fmod` returns the remainder of  $x/y$

**Parameters:**

`x`: A real number.

`y`: A real number.

Converts  $x$  and  $y$  to `mpFormulaPy` numbers and returns  $x \bmod y$ . For `mpFormulaPy` numbers, this is equivalent to `x % y`.

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> fmod(100, pi)
2.61062773871641
```

---

You can use `fmod()` to compute fractional parts of numbers:

---

```
>>> fmod(10.25, 1)
0.25
```

---

### 4.6.11 Power

---

Operator `^` (VB.NET)

Operator `**` (Python)

---

Function `.Pow(a As mpNum, b As mpNum) As mpNum`

---

The binary operator `^` is used to return  $a$  raised to the power of  $b$ , and assign the result to  $c$ :  
 $c = a ^ b$ .

For languages not supporting operator overloading, the function `.Pow` can be used to achieve the same:  $c = a.Pow(b)$

## 4.7 Logical Operators

### 4.7.1 Bitwise AND

### 4.7.2 Bitwise Inclusive OR

### 4.7.3 Bitwise Exclusive OR

## 4.8 Comparison Operators and Sorting

### 4.8.1 Equal

---

Operator `=` (VB.NET)

Operator `==` (C#)

---

Function `.EQ(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `=` returns TRUE if  $a = b$  and FALSE otherwise, e.g.:

`if (a = b) then`

For languages not supporting operator overloading, the function `.EQ` can be used to achieve the same, e.g.:

`if a.EQ(b) then`

### 4.8.2 Greater or equal

---

Operator `>=`

---

Function `.GE(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `>=` returns TRUE if  $a \geq b$  and FALSE otherwise, e.g.:

`if (a >= b) then`

For languages not supporting operator overloading, the function `.GE` can be used to achieve the same, e.g.:

`if a.GE(b) then`

### 4.8.3 Greater than

---

Operator `>`

---

Function `.GT(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `>` returns TRUE if  $a > b$  and FALSE otherwise, e.g.:

`if (a > b) then`

For languages not supporting operator overloading, the function `.GT` can be used to achieve the same, e.g.:

`if a.GT(b) then`

### 4.8.4 Less or equal

---

Operator `<=`

---

Function `.LE(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `<=` returns TRUE if  $a \leq b$  and FALSE otherwise, e.g.:

`if (a <= b) then`

For languages not supporting operator overloading, the function `.LE` can be used to achieve the same, e.g.:

`if a.LE(b) then`

### 4.8.5 Less than

---

Operator <

---

Function **.LT**(**a** As mpNum, **b** As mpNum) As Boolean

The binary logical operator > returns TRUE if  $a < b$  and FALSE otherwise, e.g.:

if (**a** < **b**) then

For languages not supporting operator overloading, the function **.LT** can be used to achieve the same, e.g.:

if **a.LT(b)** then

### 4.8.6 Not equal

---

Operator <> (VB.NET)

Operator != (C#)

---

Function **.NE**(**a** As mpNum, **b** As mpNum) As Boolean

The binary logical operator <> returns TRUE if  $a \neq b$  and FALSE otherwise, e.g.:

if (**a** <> **b**) then

For languages not supporting operator overloading, the function **.NE** can be used to achieve the same, e.g.:

if **a.NE(b)** then

### 4.8.7 Tolerances and approximate comparisons

---

Function **chop**(**x** As mpNum, **Keywords** As String) As mpNum

The function **chop** returns Chops off small real or imaginary parts, or converts numbers close to zero to exact zeros

**Parameters:**

**x**: A real or complex number.

**Keywords**: tol=None

mpFormulaPy.chop(**x**, tol=None)

Chops off small real or imaginary parts, or converts numbers close to zero to exact zeros. The input can be a single number or an iterable:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> chop(5+1e-10j, tol=1e-9)
mpf('5.0')
>>> nprint(chop([1.0, 1e-20, 3+1e-18j, -4, 2]))
[1.0, 0.0, 3.0, -4.0, 2.0]
```

The tolerance defaults to 100\*eps.

---

Function **almosteq**(**s** As mpNum, **t** As mpNum, **Keywords** As String) As mpNum

The function `almosteq` returns Determine whether the difference between  $s$  and  $t$  is smaller than a given epsilon, either relatively or absolutely.

**Parameters:**

$s$ : A real or complex number.

$t$ : A real or complex number.

*Keywords*: `rel_eps=None`, `abs_eps=None`

```
mpFormulaPy.almosteq(s, t, rel_eps=None, abs_eps=None)
```

Determine whether the difference between  $s$  and  $t$  is smaller than a given epsilon, either relatively or absolutely.

Both a maximum relative difference and a maximum difference ('epsilons') may be specified. The absolute difference is defined as  $|s - t|$  and the relative difference is defined as  $|s - t|/\max(|s|, |t|)$ .

If only one epsilon is given, both are set to the same value. If none is given, both epsilons are set to  $2^{-p+m}$  where  $p$  is the current working precision and  $m$  is a small integer. The default setting typically allows `almosteq()` to be used to check for mathematical equality in the presence of small rounding errors.

**Examples**

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15
>>> almosteq(3.141592653589793, 3.141592653589790)
True
>>> almosteq(3.141592653589793, 3.141592653589700)
False
>>> almosteq(3.141592653589793, 3.141592653589700, 1e-10)
True
>>> almosteq(1e-20, 2e-20)
True
>>> almosteq(1e-20, 2e-20, rel_eps=0, abs_eps=0)
False
```

---

## 4.9 Properties of numbers

### 4.9.1 Testing for special values

---

#### Function **isnormal**(*Number1* As *mpNum*) As Boolean

---

The function `isnormal` returns Determine whether x is 'normal' in the sense of floating-point representation; that is, return False if x is zero, an infinity or NaN; otherwise return True. By extension, a complex number x is considered 'normal' if its magnitude is normal

**Parameter:**

*Number1*: A real or complex number.

`mpFormulaPy.isnormal(x)`

Determine whether x is 'normal' in the sense of floating-point representation; that is, return False if x is zero, an infinity or NaN; otherwise return True. By extension, a complex number x is considered 'normal' if its magnitude is normal:

---

```
>>> from mpFormulaPy import *
>>> isnormal(3)
True
>>> isnormal(0)
False
>>> isnormal(inf); isnormal(-inf); isnormal(nan)
False
False
False
>>> isnormal(0+0j)
False
>>> isnormal(0+3j)
True
>>> isnormal(mpc(2,nan))
False
```

---



---

#### Function **isfinite**(*Number1* As *mpNum*) As Boolean

---

The function `isfinite` returns Return True if x is a finite number, i.e. neither an infinity or a NaN

**Parameter:**

*Number1*: A real or complex number.

`mpFormulaPy.isfinite(x)`

Return True if x is a finite number, i.e. neither an infinity or a NaN.

---

```
>>> from mpFormulaPy import *
>>> isfinite(inf)
False
>>> isfinite(-inf)
False
>>> isfinite(3)
True
>>> isfinite(nan)
False
```

---

```
>>> isfinite(3+4j)
True
>>> isfinite(mpc(3,inf))
False
>>> isfinite(mpc(nan,3))
False
```

---

### Function **isinf**(*Number1* As *mpNum*) As Boolean

---

The function **isinf** returns True if the absolute value of x is infinite; otherwise return False

#### Parameter:

*Number1*: A real or complex number.  
`mpFormulaPy.isinf(x)`

Return True if the absolute value of x is infinite; otherwise return False:

---

```
>>> from mpFormulaPy import *
>>> isinf(inf)
True
>>> isinf(-inf)
True
>>> isinf(3)
False
>>> isinf(3+4j)
False
>>> isinf(mpc(3,inf))
True
>>> isinf(mpc(inf,3))
True
```

---

### Function **isnan**(*Number1* As *mpNum*) As Boolean

---

The function **isnan** returns Return True if x is a NaN (not-a-number), or for a complex number, whether either the real or complex part is NaN; otherwise return False

#### Parameter:

*Number1*: A real or complex number.  
`mpFormulaPy.isnan(x)`

Return True if x is a NaN (not-a-number), or for a complex number, whether either the real or complex part is NaN; otherwise return False:

---

```
>>> from mpFormulaPy import *
>>> isnan(3.14)
False
>>> isnan(nan)
True
>>> isnan(mpc(3.14,2.72))
False
>>> isnan(mpc(3.14,nan))
True
```

---

## 4.9.2 Testing for integers

---

Function **isint**(*x* As *mpNum*, **Kewords** As *String*) As Boolean

---

The function **isint** returns Return True if *x* is integer-valued; otherwise return False.

**Parameters:**

*x*: A real number.

**Kewords**: gaussian=False.

mpFormulaPy.isint(*x*, gaussian=False)

Return True if *x* is integer-valued; otherwise return False:

---

```
>>> from mpFormulaPy import *
>>> isint(3)
True
>>> isint(mpf(3))
True
>>> isint(3.2)
False
>>> isint(inf)
False
```

---

Optionally, Gaussian integers can be checked for:

---

```
>>> isint(3+0j)
True
>>> isint(3+2j)
False
>>> isint(3+2j, gaussian=True)
True
```

---

## 4.9.3 Approximating magnitude and precision

---

Function **mag**(*x* As *mpNum*) As *mpNum*

---

The function **mag** returns Quick logarithmic magnitude estimate of a number.

**Parameter:**

*x*: A real number.

mpFormulaPy.mag(*x*)

Quick logarithmic magnitude estimate of a number. Returns an integer or infinity *m* such that  $|x| \leq 2^m$ . It is not guaranteed that *m* is an optimal bound, but it will never be too large by more than 2 (and probably not more than 1).

**Examples**

---

```
>>> from mpFormulaPy import *
>>> mp.pretty = True
>>> mag(10), mag(10.0), mag(mpf(10)), int(ceil(log(10,2)))
(4, 4, 4, 4)
>>> mag(10j), mag(10+10j)
```

---

```
(4, 5)
>>> mag(0.01), int(ceil(log(0.01,2)))
(-6, -6)
>>> mag(0), mag(inf), mag(-inf), mag(nan)
(-inf, +inf, +inf, nan)
```

---



---

### Function `.nint_distance(x As mpNum) As mpNum`

---

The function `.nint_distance` returns  $(n, d)$  where  $n$  is the nearest integer to  $x$  and  $d$  is an estimate of  $\log_2(|x - n|)$ .

**Parameter:**

$x$ : A real number.

`mpFormulaPy.nint_distance(x)`

Return  $(n, d)$  where  $n$  is the nearest integer to  $x$  and  $d$  is an estimate of  $\log_2(|x - n|)$ . If  $d < 0$ ,  $-d$  gives the precision (measured in bits) lost to cancellation when computing  $x - n$ .

---

```
>>> from mpFormulaPy import *
>>> n, d = nint_distance(5)
>>> print(n); print(d)
5
-inf
>>> n, d = nint_distance(mpf(5))
>>> print(n); print(d)
5
-inf
>>> n, d = nint_distance(mpf(5.00000001))
>>> print(n); print(d)
5
-26
>>> n, d = nint_distance(mpf(4.99999999))
>>> print(n); print(d)
5
-26
>>> n, d = nint_distance(mpc(5,10))
>>> print(n); print(d)
5
4
>>> n, d = nint_distance(mpc(5,0.000001))
>>> print(n); print(d)
5
-19
```

---

## 4.10 Number generation

### 4.10.1 Random numbers

---

#### Function **rand()** As mpNum

---

The function **rand** returns Returns an mpf with value chosen randomly from  $[0, 1)$ . The number of randomly generated bits in the mantissa is equal to the working precision.

mpFormulaPy.rand()

Returns an mpf with value chosen randomly from  $[0, 1)$ . The number of randomly generated bits in the mantissa is equal to the working precision.

### 4.10.2 Fractions

---

#### Function **fraction(*p* As mpNum, *q* As mpNum)** As mpNum

---

The function **fraction** returns Given Python integers  $(p, q)$ , returns a lazy mpf representing the fraction  $p/q$ . The value is updated with the precision.

**Parameters:**

*p*: an integer.

*q*: an integer.

mpFormulaPy.fraction(*p*, *q*)

Given Python integers  $(p, q)$ , returns a lazy mpf representing the fraction  $p/q$ . The value is updated with the precision.

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15
>>> a = fraction(1,100)
>>> b = mpf(1)/100
>>> print(a); print(b)
0.01
0.01
>>> mp.dps = 30
>>> print(a); print(b) # a will be accurate
0.01
0.010000000000000002081668171172
>>> mp.dps = 15
```

---

### 4.10.3 Ranges

---

#### Function **arange(*a* As mpNum, *b* As mpNum, *h* As mpNum)** As mpNum

---

The function **arange** returns This is a generalized version of Python's range() function that accepts fractional endpoints and step sizes and returns a list of mpf instance.

**Parameters:**

*a*: a real number.

*b*: a real number.

*h*: a real number.

mpFormulaPy.arange(\*args)

This is a generalized version of Python's range() function that accepts fractional endpoints and step sizes and returns a list of mpf instances. Like range(), arange() can be called with 1, 2 or 3 arguments:

arange(b):  $[0, 1, 2, \dots, x]$ .

arange(a, b):  $[a, a + 1, a + 2, \dots, x]$

arange(a, b, h):  $[a, a + h, a + 2h, \dots, x]$

where  $b - 1 \leq x < b$  (in the third case,  $b - h \leq x < b$ ).

Like Python's range(), the endpoint is not included. To produce ranges where the endpoint is included, linspace() is more convenient.

## Examples

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> arange(4)
[mpf('0.0'), mpf('1.0'), mpf('2.0'), mpf('3.0')]
>>> arange(1, 2, 0.25)
[mpf('1.0'), mpf('1.25'), mpf('1.5'), mpf('1.75')]
>>> arange(1, -1, -0.75)
[mpf('1.0'), mpf('0.25'), mpf('−0.5')]
```

---

## Function **linspace(a As mpNum, b As mpNum, h As mpNum, Keywords As String) As mpNum**

The function linspace returns This is a generalized version of Python's range() function that accepts fractional endpoints and step sizes and returns a list of mpf instance.

### Parameters:

*a*: a real number.

*b*: a real number.

*h*: a real number.

*Keywords*: endpoint=True.

mpFormulaPy.linspace(\*args, \*\*kwargs)

linspace(a, b, n) returns a list of *n* evenly spaced samples from *a* to *b*. The syntax linspace(mpi(a,b), n) is also valid.

This function is often more convenient than arange() for partitioning an interval into subintervals, since the endpoint is included:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> linspace(1, 4, 4)
[mpf('1.0'), mpf('2.0'), mpf('3.0'), mpf('4.0')]
```

---

You may also provide the keyword argument endpoint=False:

---

```
>>> linspace(1, 4, 4, endpoint=False)
[mpf('1.0'), mpf('1.75'), mpf('2.5'), mpf('3.25')]
```

---

## 4.11 Matrices

### 4.11.1 Basic methods

---

#### Function **matrix(*data* As Object, *Keywords* As String) As mpNum**

---

The function `matrix` returns This is a generalized version of Python's `range()` function that accepts fractional endpoints and step sizes and returns a list of `mpf` instance.

#### Parameters:

*data*: an object specifying the matrix.

*Keywords*: random, random-symmetric, random-complex, random-hermitian, zeros, ones, eye, row-vector, col-vector, diagonal.

Matrices in `mpFormulaPy` are implemented using dictionaries. Only non-zero values are stored, so it is cheap to represent sparse matrices.

The most basic way to create one is to use the `matrix` class directly. You can create an empty matrix specifying the dimensions:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> matrix(2)
matrix(
[[0.0, 0.0],
[0.0, 0.0]])
>>> matrix(2, 3)
matrix(
[[0.0, 0.0, 0.0],
[0.0, 0.0, 0.0]])
```

---

Calling `matrix` with one dimension will create a square matrix.

To access the dimensions of a matrix, use the `rows` or `cols` keyword:

---

```
>>> A = matrix(3, 2)
>>> A
matrix(
[[0.0, 0.0],
[0.0, 0.0],
[0.0, 0.0]])
>>> A.rows
3
>>> A.cols
2
```

---

You can also change the dimension of an existing matrix. This will set the new elements to 0. If the new dimension is smaller than before, the concerning elements are discarded:

---

```
>>> A.rows = 2
>>> A
matrix(
[[0.0, 0.0],
[0.0, 0.0]])
```

---

Internally convert is applied every time an element is set. This is done using the syntax A [row,column], counting from 0:

---

```
>>> A = matrix(2)
>>> A[1,1] = 1 + 1j
>>> print A
[0.0 0.0]
[0.0 (1.0 + 1.0j)]
```

---

A more comfortable way to create a matrix lets you use nested lists:

---

```
>>> matrix([[1, 2], [3, 4]])
matrix(
[[1.0, 2.0],
 [3.0, 4.0]])
```

---

Interval matrices can be used to perform linear algebra operations with rigorous error tracking:

---

```
>>> a = iv.matrix([[0.1, 0.3, 1.0],
... [7.1, 5.5, 4.8],
... [3.2, 4.4, 5.6]])
>>>
>>> b = iv.matrix([4, 0.6, 0.5])
>>> c = iv.lu_solve(a, b)
>>> print c
[ [5.2582327113062393041, 5.2582327113062749951]
[[ -13.155049396267856583, -13.155049396267821167]
[ [7.4206915477497212555, 7.4206915477497310922]
>>> print a*c
[ [3.999999999999866773, 4.000000000000133227]
[[0.5999999999972430942, 0.60000000000027142733]
[[0.4999999999982236432, 0.50000000000018474111]]
```

---

Convenient functions are available for creating various standard matrices:

---

```
>>> zeros(2)
matrix(
[[0.0, 0.0],
[0.0, 0.0]])
>>> ones(2)
matrix(
[[1.0, 1.0],
[1.0, 1.0]])
>>> diag([1, 2, 3]) # diagonal matrix
matrix(
[[1.0, 0.0, 0.0],
[0.0, 2.0, 0.0],
[0.0, 0.0, 3.0]])
>>> eye(2) # identity matrix
matrix(
[[1.0, 0.0],
[0.0, 1.0]])
```

---

You can even create random matrices:

---

```
>>> randmatrix(2)
matrix(
[[0.53491598236191806, 0.57195669543302752],
[0.85589992269513615, 0.82444367501382143])
```

---

## 4.11.2 Vectors

Vectors may also be represented by the matrix class (with rows = 1 or cols = 1). For vectors there are some things which make life easier. A column vector can be created using a flat list, a row vectors using an almost flat nested list:

---

```
>>> matrix([1, 2, 3])
matrix(
[[1.0],
[2.0],
[3.0]])
>>> matrix([[1, 2, 3]])
matrix(
[[1.0, 2.0, 3.0]])
```

---

Optionally vectors can be accessed like lists, using only a single index:

---

```
>>> x = matrix([1, 2, 3])
>>> x[1]
mpf('2.0')
>>> x[1,0]
mpf('2.0')
```

---

## 4.11.3 Other

Like you probably expected, matrices can be printed:

---

```
>>> print randmatrix(3)
[ 0.782963853573023 0.802057689719883 0.427895717335467]
[0.0541876859348597 0.708243266653103 0.615134039977379]
[ 0.856151514955773 0.544759264818486 0.686210904770947]
```

---

Use nstr or nprint to specify the number of digits to print:

---

```
>>> nprint(randmatrix(5), 3)
[2.07e-1 1.66e-1 5.06e-1 1.89e-1 8.29e-1]
[6.62e-1 6.55e-1 4.47e-1 4.82e-1 2.06e-2]
[4.33e-1 7.75e-1 6.93e-2 2.86e-1 5.71e-1]
[1.01e-1 2.53e-1 6.13e-1 3.32e-1 2.59e-1]
[1.56e-1 7.27e-2 6.05e-1 6.67e-2 2.79e-1]
```

---

As matrices are mutable, you will need to copy them sometimes:

---

```
>>> A = matrix(2)
>>> A
```

---

---

```

matrix(
[[0.0, 0.0],
[0.0, 0.0]])
>>> B = A.copy()
>>> B[0,0] = 1
>>> B
matrix(
[[1.0, 0.0],
[0.0, 0.0]])
>>> A
matrix(
[[0.0, 0.0],
[0.0, 0.0]])

```

---

Finally, it is possible to convert a matrix to a nested list. This is very useful, as most Python libraries involving matrices or arrays (namely NumPy or SymPy) support this format:

---

```

>>> B.tolist()
[[mpf('1.0'), mpf('0.0')], [mpf('0.0'), mpf('0.0')]]

```

---

#### 4.11.4 Transposition

Matrix transposition is straightforward:

---

```

>>> A = ones(2, 3)
>>> A
matrix(
[[1.0, 1.0, 1.0],
[1.0, 1.0, 1.0]])
>>> A.T
matrix(
[[1.0, 1.0],
[1.0, 1.0],
[1.0, 1.0]])

```

---

#### 4.11.5 Matrix Properties

Add a note on matrix properties:

Rows, Cols, T etc.

#### 4.11.6 Addition

---

Operator +

---



---

*matrix*.**Plus**(**a** As mpNum, **b** As mpNum) As mpNum

---

The binary operator + is used to return the sum of the 2 operands *a* and *b*, and assign the result to *c*: *c* = *a* + *b*.

For languages not supporting operator overloading, the function .Plus can be used to achieve the same: *c* = *a*.Plus(*b*)

The operator `+` returns the sum of `z1` and `z2`.

---

**Function `MatrixAdd(x As mpNum, y As mpNum, Keywords As String) As mpNum`**

---

The function `MatrixAdd` returns the sum of the numbers `x` and `y`, giving a floating-point result, optionally using a custom precision and rounding mode..

**Parameters:**

`x`: A complex number.

`y`: A complex number.

`Keywords`: `prec`, `dps`, `exact`, `rounding`.

`mpFormulaPy.fadd(ctx, x, y, **kwargs)`

Adds the matrices `x` and `y`, giving a floating-point result, optionally using a custom precision and rounding mode.

You can add and subtract matrices of compatible dimensions:

---

```
>>> A = matrix([[1, 2], [3, 4]])
>>> B = matrix([[-2, 4], [5, 9]])
>>> A + B
matrix(
[[-1.0, '6.0'],
['8.0', '13.0']])
>>> A - B
matrix(
[['3.0', '-2.0'],
['-2.0', '-5.0']])
>>> A + ones(3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "...", line 238, in __add__
raise ValueError('incompatible dimensions for addition')
ValueError: incompatible dimensions for addition
```

---

It is possible to multiply or add matrices and scalars. In the latter case the operation will be done element-wise:

---

```
>>> A * 2
matrix(
[['2.0', '4.0'],
['6.0', '8.0']])
>>> A / 4
matrix(
[['0.25', '0.5'],
['0.75', '1.0']])
>>> A - 1
matrix(
[['0.0', '1.0'],
['2.0', '3.0']])
```

---

### 4.11.7 Multiplication

---

Operator \*

---

`matrix.TimesMat(a As mpNum, b As Integer) As mpNum`

---

The binary operator `*` is used to return the product of the 2 operands  $a$  and  $b$ , and assign the result to  $c$ :  $c = a * b$ .

For languages not supporting operator overloading, the function `.Times` can be used to achieve the same:  $c = a.Times(b)$

`mpFormulaPy.fmul(ctx, x, y, **kwargs)`

Multiplies the numbers  $x$  and  $y$ , giving a floating-point result, optionally using a custom precision and rounding mode.

See the documentation of `fadd()` for a detailed description of how to specify precision and rounding. You can perform matrix multiplication, if the dimensions are compatible:

---

```
>>> A * B
matrix(
[[8.0, 22.0],
[14.0, 48.0]])
>>> matrix([[1, 2, 3]]) * matrix([-6, 7, -2])
matrix(
[[2.0]])
```

---

You can raise powers of square matrices:

---

```
>>> A**2
matrix(
[[7.0, 10.0],
[15.0, 22.0]])
```

---

### 4.11.8 Python-specific convenience functions

#### 4.11.8.1 `autoprec()`

`mpmath.autoprec(ctx, f, maxprec=None, catch=(), verbose=False)`

Return a wrapped copy of  $f$  that repeatedly evaluates  $f$  with increasing precision until the result converges to the full precision used at the point of the call.

This heuristically protects against rounding errors, at the cost of roughly a 2x slowdown compared to manually setting the optimal precision. This method can, however, easily be fooled if the results from  $f$  depend 'discontinuously' on the precision, for instance if catastrophic cancellation can occur. Therefore, `autoprec()` should be used judiciously.

#### Examples

Many functions are sensitive to perturbations of the input arguments. If the arguments are decimal numbers, they may have to be converted to binary at a much higher precision. If the amount of required extra precision is unknown, `autoprec()` is convenient:

---

```
>>> from mpmath import *
```

---

```
>>> mp.dps = 15
>>> mp.pretty = True
>>> besselj(5, 125 * 10**28) # Exact input
-8.03284785591801e-17
>>> besselj(5, '1.25e30') # Bad
7.12954868316652e-16
>>> autoprec(besselj)(5, '1.25e30') # Good
-8.03284785591801e-17
```

---

The following fails to converge because  $\sin(\pi) = 0$  whereas all finite-precision approximations of  $\pi$  give nonzero values:

---

```
>>> autoprec(sin)(pi)
Traceback (most recent call last):
...
NoConvergence: autoprec: prec increased to 2910 without convergence
```

---

As the following example shows, autoprec() can protect against cancellation, but is fooled by too severe cancellation:

---

```
>>> x = 1e-10
>>> exp(x)-1; expm1(x); autoprec(lambda t: exp(t)-1)(x)
1.00000008274037e-10
1.00000000005e-10
1.00000000005e-10
>>> x = 1e-50
>>> exp(x)-1; expm1(x); autoprec(lambda t: exp(t)-1)(x)
0.0
1.0e-50
0.0
```

---

With catch, an exception or list of exceptions to intercept may be specified. The raised exception is interpreted as signaling insufficient precision. This permits, for example, evaluating a function where a too low precision results in a division by zero:

---

```
>>> f = lambda x: 1/(exp(x)-1)
>>> f(1e-30)
Traceback (most recent call last):
...
ZeroDivisionError
>>> autoprec(f, catch=ZeroDivisionError)(1e-30)
1.0e+30
```

---

#### 4.11.8.2 workprec()

mpmath.workprec(ctx, n, normalize\_output=False)  
The block

---

```
with workprec(n):
<code>
```

---

sets the precision to n bits, executes `<code>`, and then restores the precision.

workprec(n)(f) returns a decorated version of the function f that sets the precision to n bits before execution, and restores the precision afterwards. With normalize\_output=True, it rounds the return value to the parent precision.

#### 4.11.8.3 workdps()

mpmath.workdps(ctx, n, normalize\_output=False)

This function is analogous to workprec (see documentation) but changes the decimal precision instead of the number of bits.

#### 4.11.8.4 extraprec()

mpmath.extraprec(ctx, n, normalize\_output=False)

The block

---

```
with extraprec(n):
    <code>
```

---

increases the precision n bits, executes `code`, and then restores the precision.

extraprec(n)(f) returns a decorated version of the function f that increases the working precision by n bits before execution, and restores the parent precision afterwards. With normalize\_output=True, it rounds the return value to the parent precision.

#### 4.11.8.5 extradps()

mpmath.extradps(ctx, n, normalize\_output=False)

This function is analogous to extraprec (see documentation) but changes the decimal precision instead of the number of bits.

#### 4.11.8.6 memoize()

mpmath.memoize(ctx, f)

Return a wrapped copy of f that caches computed values, i.e. a memoized copy of f. Values are only reused if the cached precision is equal to or higher than the working precision:

---

```
>>> from mpmath import *
>>> mp.dps = 15; mp.pretty = True
>>> f = memoize(maxcalls(sin, 1))
>>> f(2)
0.909297426825682
>>> f(2)
0.909297426825682
>>> mp.dps = 25
>>> f(2)
Traceback (most recent call last):
...
NoConvergence: maxcalls: function evaluated 1 times
```

---

#### 4.11.8.7 maxcalls()

`mpmath.maxcalls(ctx, f, N)`

Return a wrapped copy of `f` that raises `NoConvergence` when `f` has been called more than `N` times:

---

```
>>> from mpmath import *
>>> mp.dps = 15
>>> f = maxcalls(sin, 10)
>>> print(sum(f(n) for n in range(10)))
1.95520948210738
>>> f(10)
Traceback (most recent call last):
...
NoConvergence: maxcalls: function evaluated 10 times
```

---

#### 4.11.8.8 monitor()

`mpmath.monitor(f, input='print', output='print')`

Returns a wrapped copy of `f` that monitors evaluation by calling `input` with every input (`args`, `kwargs`) passed to `f` and `output` with every value returned from `f`. The default action (specify using the special string value '`print`') is to print inputs and outputs to `stdout`, along with the total evaluation count:

---

```
>>> from mpmath import *
>>> mp.dps = 5; mp.pretty = False
>>> diff(monitor(exp), 1) # diff will eval f(x-h) and f(x+h)
in 0 (mpf('0.9999999906867742538452148'),) {}
out 0 mpf('2.7182818259274480055282064')
in 1 (mpf('1.0000000009313225746154785'),) {}
out 1 mpf('2.7182818309906424675501024')
mpf('2.7182808')
```

---

To disable either the input or the output handler, you may pass `None` as argument.

Custom input and output handlers may be used e.g. to store results for later analysis:

---

```
>>> mp.dps = 15
>>> input = []
>>> output = []
>>> findroot(monitor(sin, input.append, output.append), 3.0)
mpf('3.1415926535897932')
>>> len(input) # Count number of evaluations
9
>>> print(input[3]); print(output[3])
((mpf('3.141507658334066'),), {})
8.49952562843408e-5
>>> print(input[4]); print(output[4])
((mpf('3.1415928201669122'),), {})
-1.66577118985331e-7
```

---

**4.11.8.9 timing()**

```
mpmath.timing(f, *args, **kwargs)
```

Returns time elapsed for evaluating `f()`. Optionally arguments may be passed to time the execution of `f(*args, **kwargs)`.

If the first call is very quick, `f` is called repeatedly and the best time is returned.

# Chapter 5

## Elementary Functions

### 5.1 Constants

#### 5.1.1 Mathematical constants

Mpmath supports arbitrary-precision computation of various common (and less common) mathematical constants. These constants are implemented as lazy objects that can evaluate to any precision. Whenever the objects are used as function arguments or as operands in arithmetic operations, they automagically evaluate to the current working precision. A lazy number can be converted to a regular mpf using the unary + operator, or by calling it as a function:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.2831853071795862')
>>> +pi
mpf('3.1415926535897931')
>>> pi()
mpf('3.1415926535897931')
>>> mp.dps = 40
>>> pi
<pi: 3.14159~>
>>> 2*pi
mpf('6.283185307179586476925286766559005768394338')
>>> +pi
mpf('3.141592653589793238462643383279502884197169')
>>> pi()
mpf('3.141592653589793238462643383279502884197169')
```

---

#### Function **pi()** As mpNum

---

The function pi returns pi: 3.14159...

---

#### Function **degree()** As mpNum

---

The function `degree` returns  $\text{degree} = 1 \text{ deg} = \pi / 180: 0.0174533\dots$

---

**Function `e()` As mpNum**

---

The function `e` returns the base of the natural logarithm,  $e = \exp(1): 2.71828\dots$

---

**Function `phi()` As mpNum**

---

The function `phi` returns Golden ratio  $\phi: 1.61803\dots$

---

**Function `euler()` As mpNum**

---

The function `euler` returns Euler's constant:  $0.577216\dots$

---

**Function `catalan()` As mpNum**

---

The function `catalan` returns Catalan's constant:  $0.915966\dots$

---

**Function `apery()` As mpNum**

---

The function `apery` returns Apery's constant:  $1.20206\dots$

---

**Function `khinchin()` As mpNum**

---

The function `khinchin` returns Khinchin's constant:  $2.68545\dots$

---

**Function `glaisher()` As mpNum**

---

The function `glaisher` returns Glaisher's constant:  $1.28243\dots$

---

**Function `mertens()` As mpNum**

---

The function `mertens` returns Mertens' constant:  $0.261497\dots$

---

**Function `twinprime()` As mpNum**

---

The function `twinprime` returns Twin prime constant:  $0.660162\dots$

### 5.1.2 Special values

The predefined objects `j` (imaginary unit), `inf` (positive infinity) and `nan` (not-a-number) are shortcuts to `mpc` and `mpf` instances with these fixed values.

---

**Function `inf()` As mpNum**

---

The function `inf` returns the value of the representation of  $+\infty$  in the current precision.

---

**Function `nan()` As mpNum**

---

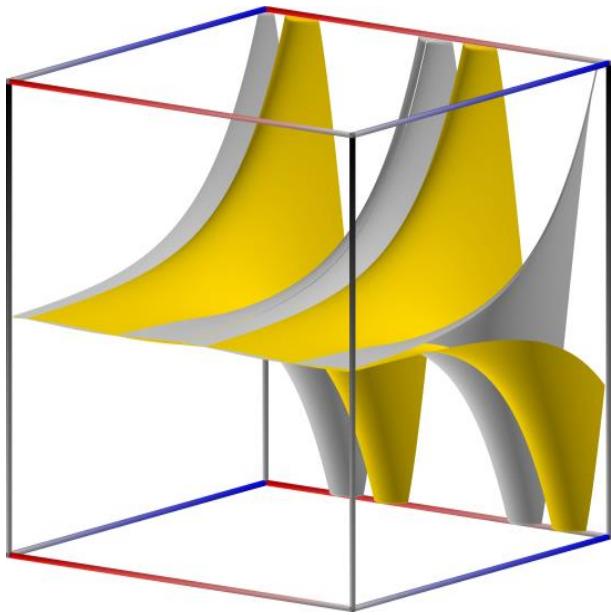
The function `nan` returns the value of the representation of Not a Number (NaN) in the current precision.

## 5.2 Exponential and Logarithmic Functions

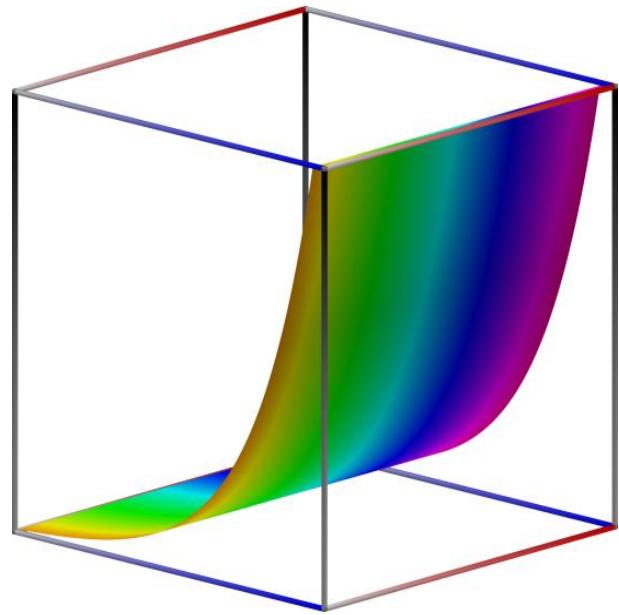
### 5.2.1 Exponential Function $e^z = \exp(z)$

The function `IMEXP(z)` returns the complex exponential function of  $z$ :

$$\exp(z) = e^x \cos(y) + ie^x \sin(y). \quad (5.2.1)$$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.1:** Surface plots of  $z = \exp(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `exp(z As mpNum) As mpNum`

---

The function `exp` returns the complex exponential of  $z$

##### Parameter:

$z$ : A complex number.

Computes the exponential function,

$$\exp(x) = e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}. \quad (5.2.2)$$

For complex numbers, the exponential function also satisfies

$$\exp(x + yi) = e^x(\cos(y) + i \sin(y)). \quad (5.2.3)$$

Basic examples

Some values of the exponential function:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> exp(0)
1.0
>>> exp(1)
2.718281828459045235360287
>>> exp(-1)
0.3678794411714423215955238
>>> exp(10000)
+inf
>>> exp(-10000)
0.0
```

---

Arguments can be arbitrarily large:

---

```
>>> exp(10000)
8.806818225662921587261496e+4342
>>> exp(-10000)
1.135483865314736098540939e-4343
```

---

Evaluation is supported for interval arguments via `mpFormulaPy.iv.exp()`:

---

```
>>> iv.dps = 25; iv.pretty = True
>>> iv.exp([-inf,0])
[0.0, 1.0]
>>> iv.exp([0,1])
[1.0, 2.71828182845904523536028749558]
```

---

The exponential function can be evaluated efficiently to arbitrary precision:

---

```
>>> mp.dps = 10000
>>> exp(pi)
23.140692632779269005729...8984304016040616
```

---

Functional properties

Numerical verification of Euler's identity for the complex exponential function:

---

```
>>> mp.dps = 15
>>> exp(j*pi)+1
(0.0 + 1.22464679914735e-16j)
>>> chop(exp(j*pi)+1)
0.0
```

---

This recovers the coefficients (reciprocal factorials) in the Maclaurin series expansion of `exp`:

---

```
>>> nprint(taylor(exp, 0, 5))
[1.0, 1.0, 0.5, 0.166667, 0.0416667, 0.00833333]
```

---

The exponential function is its own derivative and antiderivative:

---

```
>>> exp(pi)
23.1406926327793
>>> diff(exp, pi)
```

---

```
23.1406926327793
>>> quad(exp, [-inf, pi])
23.1406926327793
```

---

The exponential function can be evaluated using various methods, including direct summation of the series, limits, and solving the defining differential equation:

---

```
>>> nsum(lambda k: pi**k/fac(k), [0,inf])
23.1406926327793
>>> limit(lambda k: (1+pi/k)**k, inf)
23.1406926327793
>>> odefun(lambda t, x: x, 0, 1)(pi)
23.1406926327793
```

---

### 5.2.1.1 expj(z)

---

Function **expj(z As mpNum)** As mpNum

---

The function `expj` returns  $10^z$

**Parameter:**

*z*: A complex number.

Convenience function for computing  $e^{iz}$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> expj(0)
(1.0 + 0.0j)
>>> expj(-1)
(0.5403023058681397174009366 - 0.8414709848078965066525023j)
>>> expj(j)
(0.3678794411714423215955238 + 0.0j)
>>> expj(1+j)
(0.1987661103464129406288032 + 0.3095598756531121984439128j)
```

---

### 5.2.1.2 expjpi(x)

---

Function **expjpi(z As mpNum)** As mpNum

---

The function `expjpi` returns  $10^z$

**Parameter:**

*z*: A complex number.

Convenience function for computing  $e^{i\pi x}$ . Evaluation is accurate near zeros (see also `cospi()`, `sinpi()`):

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> expjpi(0)
(1.0 + 0.0j)
>>> expjpi(1)
```

---

```

(-1.0 + 0.0j)
>>> expjpi(0.5)
(0.0 + 1.0j)
>>> expjpi(-1)
(-1.0 + 0.0j)
>>> expjpi(j)
(0.04321391826377224977441774 + 0.0j)
>>> expjpi(1+j)
(-0.04321391826377224977441774 + 0.0j)

```

---

### 5.2.1.3 `expm1(x)`

---

Function **expm1(z As mpNum) As mpNum**

---

The function `expm1` returns  $10^z$

**Parameter:**

`z`: A complex number.

Convenience function for computing  $e^x - 1$  accurately for small  $x$ .

Unlike the expression `exp(x) - 1`, `expm1(x)` does not suffer from potentially catastrophic cancellation:

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> exp(1e-10)-1; print(expm1(1e-10))
1.00000008274037e-10
1.00000000005e-10
>>> exp(1e-20)-1; print(expm1(1e-20))
0.0
1.0e-20
>>> 1/(exp(1e-20)-1)
Traceback (most recent call last):
...
ZeroDivisionError
>>> 1/expm1(1e-20)
1.0e+20

```

---

Evaluation works for extremely tiny values:

---

```

>>> expm1(0)
0.0
>>> expm1('1e-10000000')
1.0e-10000000

```

---

### 5.2.2 Exponential Function $10^z = \exp_{10}(z)$

---

Function **exp10(z As mpNum) As mpNum**

---

The function `exp10` returns  $10^z$

**Parameter:**

*z*: A complex number.

The function `exp10(z)` returns  $10^z = \exp_{10}(z) = \exp(z \cdot \ln(10))$ .

### 5.2.3 Exponential Function $2^z = \exp_2(z)$

---

Function **exp2(z As mpNum)** As mpNum

---

The function `exp2` returns  $2^z$

**Parameter:**

*z*: A complex number.

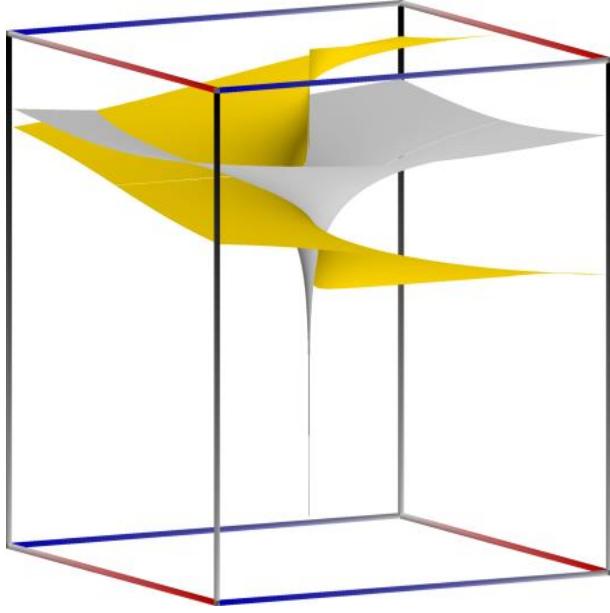
The function `cplxExp2(z)` returns  $2^z = \exp_2(z) = \exp(z \cdot \ln(2))$ .

### 5.2.4 Natural logarithm $\ln(x) = \log_e(x)$

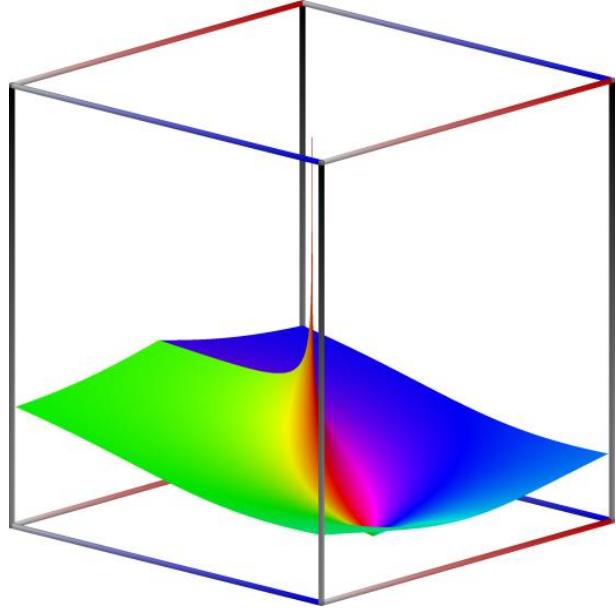
The function `cplxLn(z)` returns the complex natural logarithm of  $z$ :

$$\ln(z) = \log_e(z) = \ln(r) + i\theta, \quad (5.2.4)$$

where  $r = \sqrt{x^2 + y^2}$ , and  $\theta = \arctan(y/x)$ .



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.2:** Surface plots of  $z = \log(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

#### 5.2.4.1 `log(x, b=None)`

---

Function **Logb**(*x* As `mpNum`, *b* As `mpNum`) As `mpNum`

---

The function `Logb` returns the value of the logarithm to base  $b$ :  $\log_b(x) = \log_b(x)$ .

**Parameters:**

*x*: A real number.

*b*: A real number.

---

Function **log**(*z* As `mpNum`, *base* As `mpNum`) As `mpNum`

---

The function `log` returns the complex natural logarithm of  $z$

**Parameters:**

*z*: A complex number.

*base*: the base of the logarithm. A real number.

---

Function **ln**(*z* As `mpNum`) As `mpNum`

---

The function `ln` returns the complex natural logarithm of  $z$

**Parameter:**

$z$ : A complex number.

Computes the base- $b$  logarithm of  $x$ ,  $\log_b(x)$ . If  $b$  is unspecified, `log()` computes the natural (base  $e$ ) logarithm and is equivalent to `ln()`. In general, the base  $b$  logarithm is defined in terms of the natural logarithm as  $\log_b(x) = \ln(x)/\ln(b)$ .

By convention, we take  $\log(0) = -\infty$ .

The natural logarithm is real if  $x > 0$  and complex if  $x < 0$  or if  $x$  is complex. The principal branch of the complex logarithm is used, meaning that  $\Im(\ln(x)) = -\pi < \arg(x) \leq \pi$ .

Examples Some basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> log(1)
0.0
>>> log(2)
0.693147180559945
>>> log(1000,10)
3.0
>>> log(4, 16)
0.5
>>> log(j)
(0.0 + 1.5707963267949j)
>>> log(-1)
(0.0 + 3.14159265358979j)
>>> log(0)
-inf
>>> log(inf)
+inf
```

---

The natural logarithm is the antiderivative of  $1/x$ :

---

```
>>> quad(lambda x: 1/x, [1, 5])
1.6094379124341
>>> log(5)
1.6094379124341
>>> diff(log, 10)
0.1
```

---

The Taylor series expansion of the natural logarithm around  $x = 1$  has coefficients  $(-1)^{n+1}/n$ :

---

```
>>> nprint(taylor(log, 1, 7))
[0.0, 1.0, -0.5, 0.333333, -0.25, 0.2, -0.166667, 0.142857]
```

---

`log()` supports arbitrary precision evaluation:

---

```
>>> mp.dps = 50
>>> log(pi)
1.1447298858494001741434273513530587116472948129153
>>> log(pi, pi**3)
```

### 5.2.5 Common (decadic) logarithm $\log_{10}(z)$

The function `log10(z)` returns the complex natural logarithm of  $z$ :

$$\log_{10}(z) = \ln(z)/\ln(10). \quad (5.2.5)$$

$\log_{10}(x)$  is equivalent to  $\log(x, 10)$ .

Function **log10**(*z As mpNum*) As mpNum

The function `log10` returns  $\log_{10}(z)$

## Parameter:

$z$ : A complex number.

### 5.2.6 Binary logarithm $\log_2(z)$

The function `cplxLn(z)` returns the complex natural logarithm of  $z$ :

$$\log_2(z) = \ln(z)/\ln(2). \quad (5.2.6)$$

Function **log2(z As mpNum)** As mpNum

The function `log2` returns  $\log_2(z)$

## Parameter:

$z$ : A complex number.

### 5.2.7 Auxiliary Function $\ln(1 + x)$

Function **Inp1**(*x As mpNum*) As mpNum

The function `lnp1` returns the value of the function  $\ln(1 + x)$ .

## Parameter:

$x$ : A real number.

## 5.3 Roots and Power Functions

### 5.3.1 Square: $z^2$

---

Function **square**(*z* As *mpNum*) As *mpNum*

---

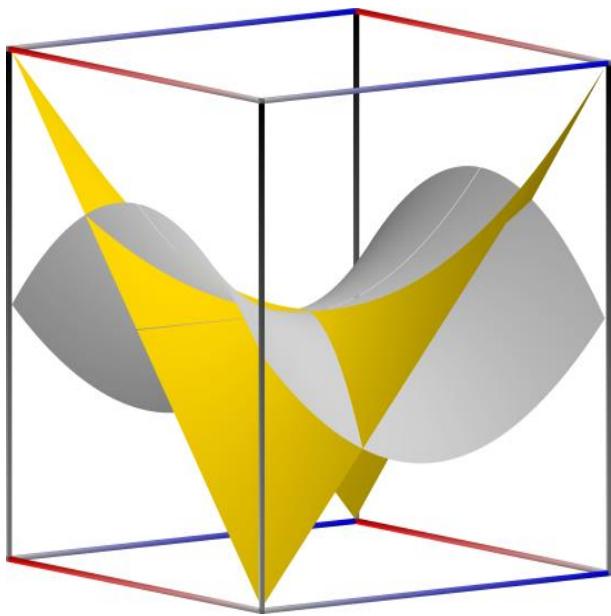
The function **square** returns the square of *z*.

**Parameter:**

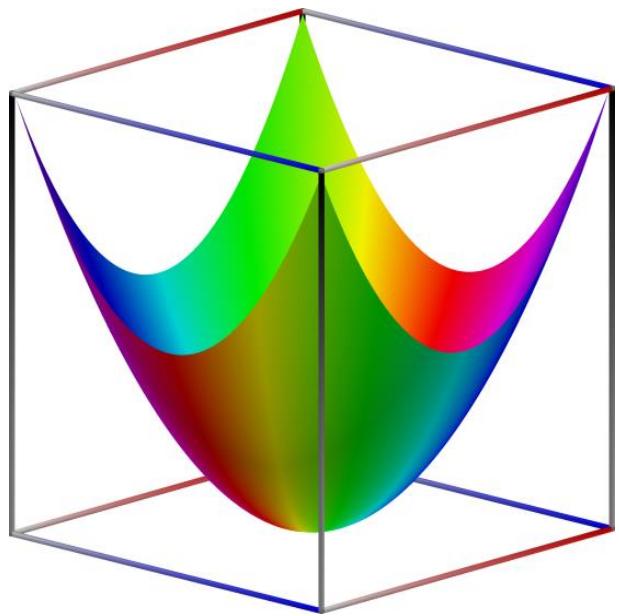
*z*: A complex number.

The function **cplxSqr**(*z1*, *z2*) returns the square of *z*:

$$z^2 = x^2 - y^2 + i(2xy). \quad (5.3.1)$$



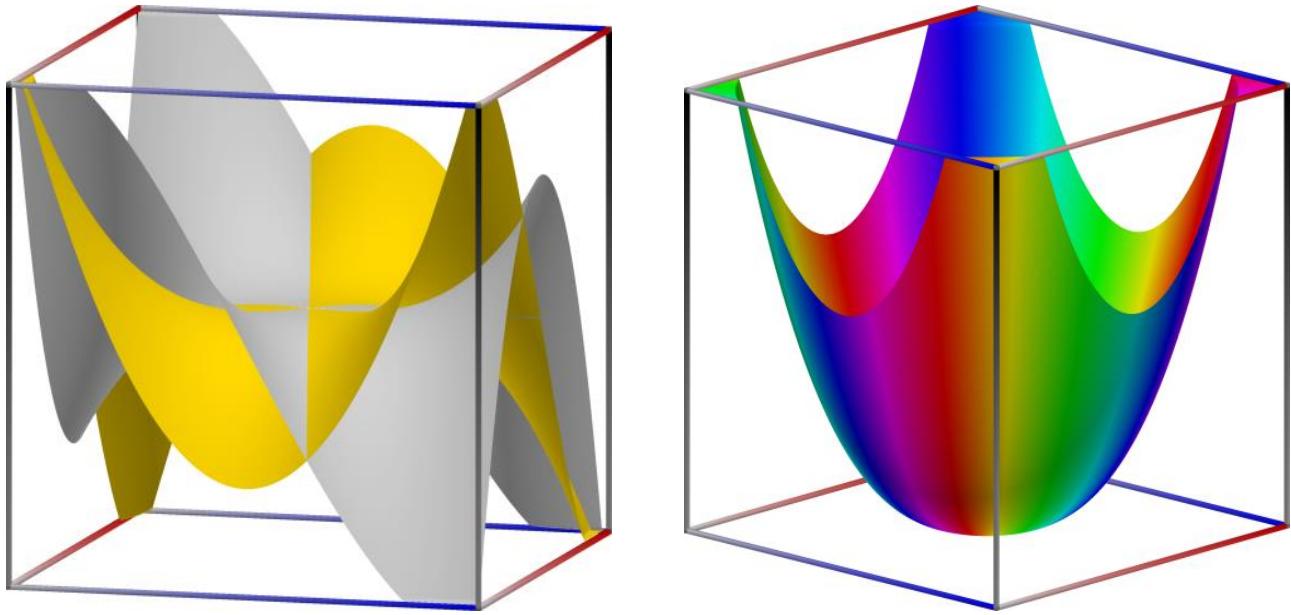
(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.3:** Surface plots of  $z = (x + iy)^2$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

### 5.3.2 Power Function



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.4:** Surface plots of  $z = (x + iy)^3$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function **power(z1 As mpNum, z2 As mpNum) As mpNum**

---

The function **power** returns an complex power of  $z$

**Parameters:**

**z1:** A complex number.

**z2:** A complex number.

Converts  $x$  and  $y$  to mpFormulaPy numbers and evaluates  $x^y = \exp(y \log(x))$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 30; mp.pretty = True
>>> power(2, 0.5)
1.41421356237309504880168872421
```

---

This shows the leading few digits of a large Mersenne prime (performing the exact calculation  $2^{43112609} - 1$  and displaying the result in Python would be very slow):

---

```
>>> power(2, 43112609)-1
3.16470269330255923143453723949e+12978188
```

---

The function **cplxPower(z, k)** returns an integer power of  $z$ :

$$z^k = r^k \cos(k\theta) + i(r^k \sin(k\theta)), \quad k \in \mathbb{Z}, \quad (5.3.2)$$

where  $r = \sqrt{x^2 + y^2}$ , and  $\theta = \arctan(y/x)$ .

The function `cplxPowR(z, k)` returns a real power of  $z$ :

$$z^a = r^a \cos(a\theta) + i(r^a \sin(a\theta)), \quad a \in \mathbb{R}, \quad (5.3.3)$$

where  $r = \sqrt{x^2 + y^2}$ , and  $\theta = \arctan(y/x)$ .

The function `cplxPowC(z, k)` returns a complex power of  $z_1$ :

$$z_1^{z_2} = \exp(\ln(z_1)z_2), \quad z_1, z_2 \in \mathbb{C}. \quad (5.3.4)$$

### 5.3.2.1 `powm1(x, y)`

---

#### Function `powm1(z As mpNum, k As mpNum) As mpNum`

---

The function `powm1` returns an integer power of  $z$

**Parameters:**

$z$ : A complex number.

$k$ : A complex number.

Convenience function for computing  $x^x - 1$  accurately when  $x^y$  is very close to 1. This avoids potentially catastrophic cancellation:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> power(0.99999995, 1e-10) - 1
0.0
>>> powm1(0.99999995, 1e-10)
-5.00000012791934e-18
```

---

Powers exactly equal to 1, and only those powers, yield 0 exactly:

---

```
>>> powm1(-j, 4)
(0.0 + 0.0j)
>>> powm1(3, 0)
0.0
>>> powm1(fadd(-1, 1e-100, exact=True), 4)
-4.0e-100
```

---

Evaluation works for extremely tiny  $y$ :

---

```
>>> powm1(2, '1e-100000')
6.93147180559945e-100001
>>> powm1(j, '1e-1000')
(-1.23370055013617e-2000 + 1.5707963267949e-1000j)
```

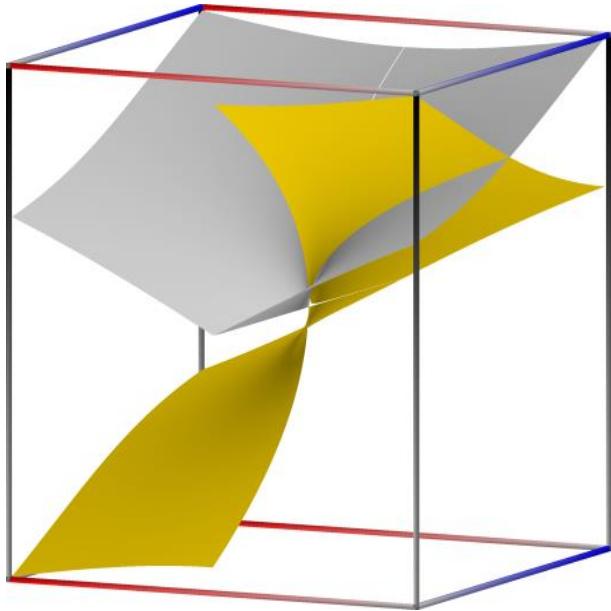
---

### 5.3.3 Square Root: $\sqrt{z}$

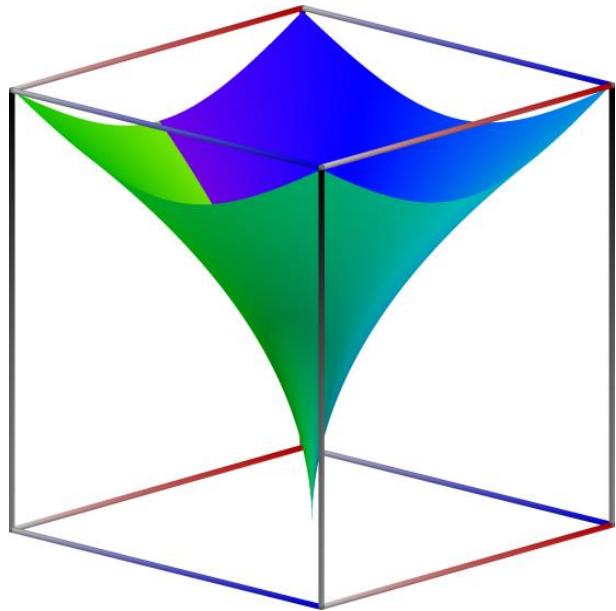
The function `cplxSqrt(z)` returns the square root of  $z$ :

$$\sqrt{z} = \sqrt{r} \cos\left(\frac{1}{2}\theta\right) + i\sqrt{r} \sin\left(\frac{1}{2}\theta\right), \quad (5.3.5)$$

where  $r = \sqrt{x^2 + y^2}$ , and  $\theta = \arctan(y/x)$ .



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.5:** Surface plots of  $z = \sqrt{x+iy}$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `sqrt(z As mpNum) As mpNum`

---

The function `sqrt` returns the square root of  $z$

##### Parameter:

$z$ : A complex number.

`sqrt(x)` gives the principal square root of  $x, \sqrt{x}$ . For positive real numbers, the principal root is simply the positive square root. For arbitrary complex numbers, the principal square root is defined to satisfy  $\sqrt{x} = \exp(\log(x)/2)$ . The function thus has a branch cut along the negative half real axis. For all `mpFormulaPy` numbers  $x$ , calling `sqrt(x)` is equivalent to performing  $x^{**0.5}$ . Examples

Basic examples and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> sqrt(10)
```

---

```
3.16227766016838
>>> sqrt(100)
10.0
>>> sqrt(-4)
(0.0 + 2.0j)
>>> sqrt(1+1j)
(1.09868411346781 + 0.455089860562227j)
>>> sqrt(inf)
+inf
```

---

Square root evaluation is fast at huge precision:

---

```
>>> mp.dps = 50000
>>> a = sqrt(3)
>>> str(a)[-10:]
'9329332814'
```

---

mpFormulaPy.iv.sqrt() supports interval arguments:

---

```
>>> iv.dps = 15; iv.pretty = True
>>> iv.sqrt([16,100])
[4.0, 10.0]
>>> iv.sqrt(2)
[1.4142135623730949234, 1.4142135623730951455]
>>> iv.sqrt(2) ** 2
[1.99999999999995559, 2.0000000000000004441]
```

---

### 5.3.4 Auxiliary Function $\sqrt{x^2 + y^2}$

Computes the Euclidean norm of the vector  $(x, y)$  , equal to  $\sqrt{x^2 + y^2}$  . Both  $x$  and  $y$  must be real.

---

Function **hypot(x As mpNum, y As mpNum)** As mpNum

---

The function **hypot** returns the value of  $\sqrt{x^2 + y^2}$ .

**Parameters:**

**x:** A real number.

**y:** A real number.

### 5.3.5 Cube Root: $\sqrt[n]{x}, n = 2, 3, \dots$

---

Function **cbrt(z As mpNum)** As mpNum

---

The function **cbrt** returns the square root of  $z$

**Parameter:**

**z:** A complex number.

**cbrt(x)** computes the cube root of  $x$ ,  $x^{1/3}$  . This function is faster and more accurate than raising to a floating-point fraction:

---

```
>>> from mpFormulaPy import *
```

---

```
>>> mp.dps = 15; mp.pretty = False
>>> 125**(mpf(1)/3)
mpf('4.999999999999991')
>>> cbrt(125)
mpf('5.0')
```

---

Every nonzero complex number has three cube roots. This function returns the cube root defined by  $\exp(\log(x)/3)$ , where the principal branch of the natural logarithm is used. Note that this does not give a real cube root for negative real numbers:

---

```
>>> mp.pretty = True
>>> cbrt(-1)
(0.5 + 0.866025403784439j)
```

---

### 5.3.6 Nth Root: $\sqrt[n]{z}$ , $n = 2, 3, \dots$

---

Function **root**(*z* As *mpNum*, *n* As *mpNum*) As *mpNum*

---

The function **root** returns the value of the  $n^{th}$  root of *x*,  $\sqrt[n]{x}$ ,  $n = 2, 3, \dots$ .

**Parameters:**

*z*: A complex number.

*n*: An integer.

The  $n^{th}$  root of *z* is defined as:

$$\sqrt[n]{z} = z^{1/n} = \sqrt[n]{r} \exp\left(\frac{i\theta}{n}\right), \quad n \in \mathbb{N}, \quad (5.3.6)$$

where  $r = \sqrt{x^2 + y^2}$ , and  $\theta = \arctan(y/x)$ . This is the principal root if  $-\pi < \theta \leq \pi$ . The other roots are given by

$$\sqrt[n]{z} = \sqrt[n]{r} \exp\left(\frac{i(\theta + 2\pi k)}{n}\right), \quad k = 1, 2, \dots, n-1. \quad (5.3.7)$$

---

Function **nthroot**(*n* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function **nthroot** returns the value of the  $n^{th}$  root of *x*,  $\sqrt[n]{x}$ ,  $n = 2, 3, \dots$ .

**Parameters:**

*n*: An integer.

*y*: A real number.

This is an alternative version for **root** (see above).

Every complex number  $z \neq 0$  has *n* distinct *n*-th roots, which are equidistant points on a circle with radius  $|z|^{1/n}$ , centered around the origin. A specific root may be selected using the optional index *k*. The roots are indexed counterclockwise, starting with *k* = 0 for the root closest to the positive real half-axis.

The *k* = 0 root is the so-called principal *n*-th root, often denoted by  $\sqrt[n]{z}$  or  $z^{1/n}$ , and also given by  $\exp(\log(z)/n)$ . If *z* is a positive real number, the principal root is just the unique positive *n*-th root of *z*. Under some circumstances, non-principal real roots exist: for positive real *z*, *n* even,

there is a negative root given by  $k = n/2$ ; for negative real  $z$ ,  $n$  odd, there is a negative root given by  $k = (n - 1)/2$ .

To obtain all roots with a simple expression, use

---

```
[root(z, n, k) for k in range(n)].
```

---

An important special case, `root(1, n, k)` returns the  $k$ -th  $n$ -th root of unity,  $\zeta_k = e^{2\pi ik/n}$ . Alternatively, `unitroots()` provides a slightly more convenient way to obtain the roots of unity, including the option to compute only the primitive roots of unity.

Both  $k$  and  $n$  should be integers;  $k$  outside of `range(n)` will be reduced modulo  $n$ . If  $n$  is negative,  $x^{-1/n} = 1/x^{1/n}$  (or the equivalent reciprocal for a non-principal root with  $k \neq 0$ ) is computed.

`root()` is implemented to use Newton's method for small  $n$ . At high precision, this makes  $x^{1/n}$  not much more expensive than the regular exponentiation,  $x^n$ . For very large  $n$ , `nthroot()` falls back to use the exponential function.

Examples

`nthroot()`/`root()` is faster and more accurate than raising to a floating-point fraction:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> 16807 ** (mpf(1)/5)
mpf('7.000000000000009')
>>> root(16807, 5)
mpf('7.0')
>>> nthroot(16807, 5) # Alias
mpf('7.0')
```

---

A high-precision root:

---

```
>>> mp.dps = 50; mp.pretty = True
>>> nthroot(10, 5)
1.584893192461113485202101373391507013269442133825
>>> nthroot(10, 5) ** 5
10.0
```

---

Computing principal and non-principal square and cube roots:

---

```
>>> mp.dps = 15
>>> root(10, 2)
3.16227766016838
>>> root(10, 2, 1)
-3.16227766016838
>>> root(-10, 3)
(1.07721734501594 + 1.86579517236206j)
>>> root(-10, 3, 1)
-2.15443469003188
>>> root(-10, 3, 2)
(1.07721734501594 - 1.86579517236206j)
```

---

All the 7th roots of a complex number:

---

```
>>> for r in [root(3+4j, 7, k) for k in range(7)]:
...     print("%s %s" % (r, r**7))
...
(1.24747270589553 + 0.166227124177353j) (3.0 + 4.0j)
(0.647824911301003 + 1.07895435170559j) (3.0 + 4.0j)
(-0.439648254723098 + 1.17920694574172j) (3.0 + 4.0j)
(-1.19605731775069 + 0.391492658196305j) (3.0 + 4.0j)
(-1.05181082538903 - 0.691023585965793j) (3.0 + 4.0j)
(-0.115529328478668 - 1.25318497558335j) (3.0 + 4.0j)
(0.907748109144957 - 0.871672518271819j) (3.0 + 4.0j)
```

---

Cube roots of unity:

---

```
>>> for k in range(3): print(root(1, 3, k))
...
1.0
(-0.5 + 0.866025403784439j)
(-0.5 - 0.866025403784439j)
```

---

Some exact high order roots:

---

```
>>> root(75**210, 105)
5625.0
>>> root(1, 128, 96)
(0.0 - 1.0j)
>>> root(4**128, 128, 96)
(0.0 - 4.0j)
```

---

### 5.3.6.1 unitroots(n, primitive=False)

unitroots(n) returns  $\zeta_0, \zeta_1, \dots, \zeta_{n-1}$ , all the distinct  $n$ -th roots of unity, as a list. If the option primitive=True is passed, only the primitive roots are returned.

Every  $n$ -th root of unity satisfies  $(\zeta_k)^n = 1$ . There are distinct roots for each  $n$  ( $\zeta_k$  and  $\zeta_j$  are the same when  $k = j \pmod n$ ), which form a regular polygon with vertices on the unit circle. They are ordered counterclockwise with increasing  $k$ , starting with  $\zeta_0 = 1$ .

Examples

The roots of unity up to  $n = 4$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(unitroots(1))
[1.0]
>>> nprint(unitroots(2))
[1.0, -1.0]
>>> nprint(unitroots(3))
[1.0, (-0.5 + 0.866025j), (-0.5 - 0.866025j)]
>>> nprint(unitroots(4))
[1.0, (0.0 + 1.0j), -1.0, (0.0 - 1.0j)]
```

---

Roots of unity form a geometric series that sums to 0:

---

```
>>> mp.dps = 50
>>> chop(fsum(unitroots(25)))
0.0
```

---

Primitive roots up to  $n = 4$ :

---

```
>>> mp.dps = 15
>>> nprint(unitroots(1, primitive=True))
[1.0]
>>> nprint(unitroots(2, primitive=True))
[-1.0]
>>> nprint(unitroots(3, primitive=True))
[(-0.5 + 0.866025j), (-0.5 - 0.866025j)]
>>> nprint(unitroots(4, primitive=True))
[(0.0 + 1.0j), (0.0 - 1.0j)]
```

---

There are only four primitive 12th roots:

---

```
>>> nprint(unitroots(12, primitive=True))
[(0.866025 + 0.5j), (-0.866025 + 0.5j), (-0.866025 - 0.5j), (0.866025 - 0.5j)]
```

---

The  $n$ -th roots of unity form a group, the cyclic group of order  $n$ . Any primitive root  $r$  is a generator for this group, meaning that  $r^0, r^1, \dots, r^{n-1}$  gives the whole set of unit roots (in some permuted order):

---

```
>>> for r in unitroots(6): print(r)
...
1.0
(0.5 + 0.866025403784439j)
(-0.5 + 0.866025403784439j)
-1.0
(-0.5 - 0.866025403784439j)
(0.5 - 0.866025403784439j)
>>> r = unitroots(6, primitive=True)[1]
>>> for k in range(6): print(chop(r**k))
...
1.0
(0.5 - 0.866025403784439j)
(-0.5 - 0.866025403784439j)
-1.0
(-0.5 + 0.866025403784438j)
(0.5 + 0.866025403784438j)
```

---

The number of primitive roots equals the Euler totient function  $\phi(n)$ :

---

```
>>> [len(unitroots(n, primitive=True)) for n in range(1,20)]
[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18]
```

---

## 5.4 Trigonometric Functions

### 5.4.1 Trigonometric functions: overview

Except where otherwise noted, the trigonometric functions take a radian angle as input and the inverse trigonometric functions return radian angles.

The ordinary trigonometric functions are single-valued functions defined everywhere in the complex plane (except at the poles of tan, sec, csc, and cot). They are defined generally via the exponential function, e.g.

$$\cos(x) = \frac{1}{2}(e^{ix} + e^{-ix}) \quad (5.4.1)$$

The inverse trigonometric functions are multivalued, thus requiring branch cuts, and are generally real-valued only on a part of the real line. Definitions and branch cuts are given in the documentation of each function. The branch cut conventions used by mpFormulaPy are essentially the same as those found in most standard mathematical software, such as Mathematica and Python's own cmath library (as of Python 2.6; earlier Python versions implement some functions erroneously).

### 5.4.2 Conversion between Degrees and radians

---

#### Function **degrees(x As mpNum)** As mpNum

---

The function **degrees** returns the value of  $x$  converted to degrees, with the input  $x$  in radians.

**Parameter:**

$x$ : A real number.

Converts the radian angle  $x$  to a degree angle:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> degrees(pi/3)
60.0
```

---

The following formula is used:

$$\text{Radians}(x) = x \cdot \frac{\pi}{180}. \quad (5.4.2)$$

---

#### Function **radians(x As mpNum)** As mpNum

---

The function **radians** returns the value of  $x$  converted to radians, with the input  $x$  in degrees.

**Parameter:**

$x$ : A real number.

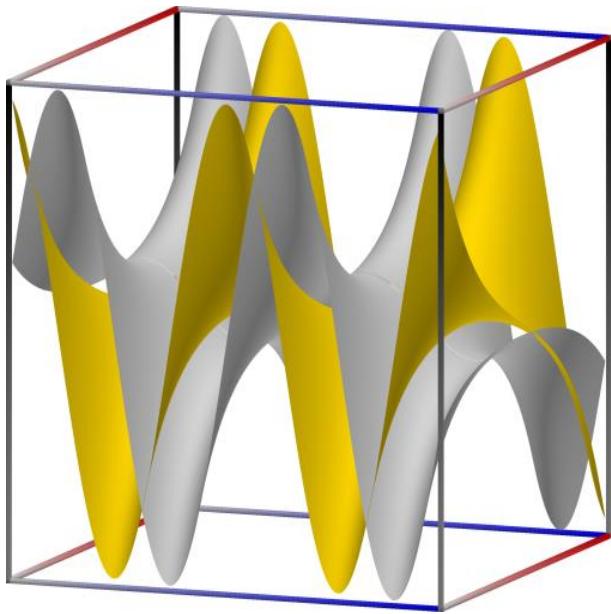
Converts the degree angle  $x$  to radians:

---

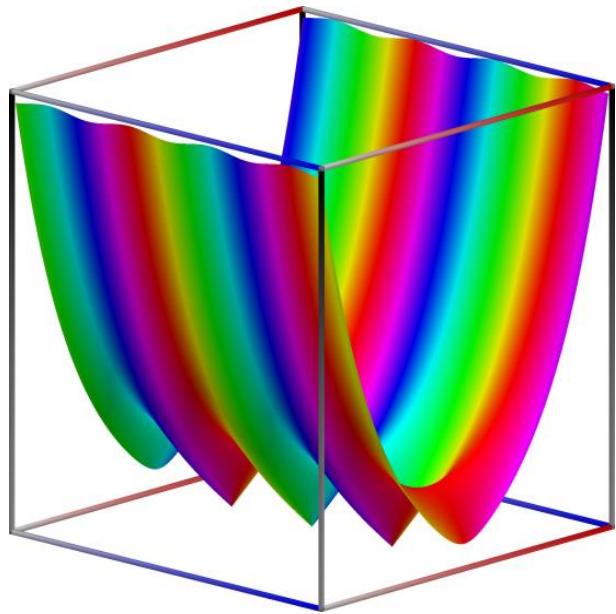
```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> radians(60)
1.0471975511966
```

---

### 5.4.3 Sine: $\sin(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.6:** Surface plots of  $z = \sin(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function **sin(z As mpNum)** As mpNum

---

The function **sin** returns complex sine of  $z$

**Parameter:**

$z$ : A complex number.

The function **cplxSin(z)** returns the complex sine of  $z$ :

$$\sin(z) = \sin(x) \cosh(y) + i \cos(x) \sinh(y). \quad (5.4.3)$$

Computes the sine of  $x$ ,  $\sin(x)$ .

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> sin(pi/3)
0.8660254037844386467637232
>>> sin(100000001)
0.1975887055794968911438743
>>> sin(2+3j)
(9.1544991469114295734673 - 4.168906959966564350754813j)
>>> sin(inf)
nan
>>> nprint(chop(taylor(sin, 0, 6)))
```

---

```
[0.0, 1.0, 0.0, -0.166667, 0.0, 0.00833333, 0.0]
```

---

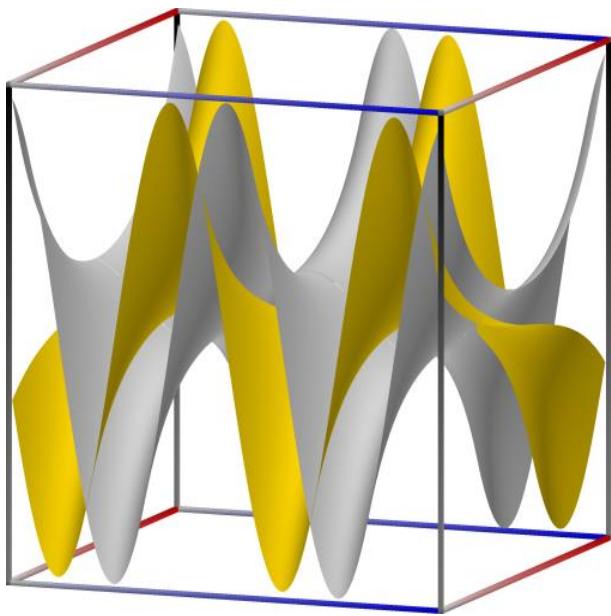
Near multiples of  $\pi$ , the relative error can become large, when using hardware based double precision.

---

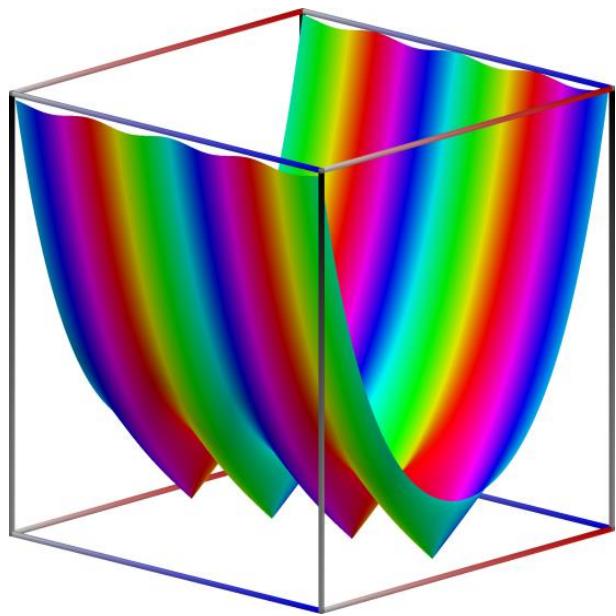
```
>>> a=3.14159265358979
>>> d=Decimal(a)
>>> d
Decimal('3.141592653589790007373494518105871975421905517578125')
>>> xl.SIN(a)
3.2311393144412999e-15
>>> mp.sin(a)
mpf('3.231089148865173630908775263e-15')
>>> iv.sin(a)
mpi('3.2310891488651735e-15', '3.2310891488651739e-15')
>>>
```

---

### 5.4.4 Cosine: $\cos(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.7:** Surface plots of  $z = \cos(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function **cos(z As mpNum) As mpNum**

---

The function **cos** returns complex cosine of  $z$ .

**Parameter:**

**$z$ :** A complex number.

The function **cplxCos(z)** returns the complex cosine of  $z$ :

$$\cos(z) = \cos(x) \cosh(y) - i \sin(x) \sinh(y). \quad (5.4.4)$$

Computes the cosine of  $x$ ,  $\cos(x)$ .

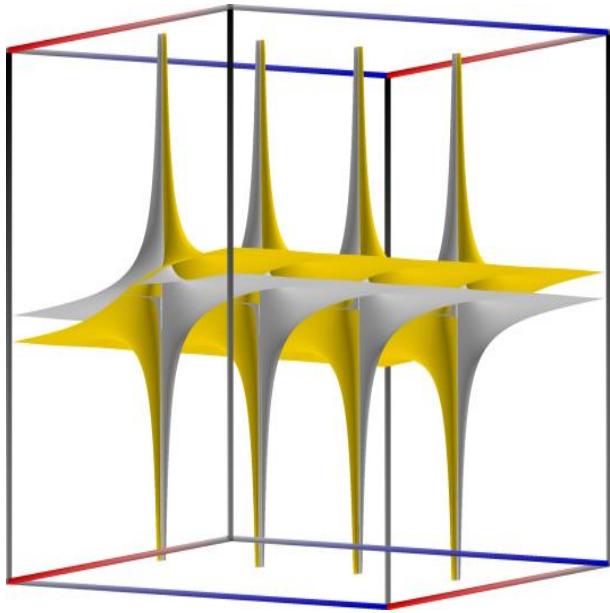
---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> cos(pi/3)
0.5
>>> cos(100000001)
-0.9802850113244713353133243
>>> cos(2+3j)
(-4.189625690968807230132555 - 9.109227893755336597979197j)
>>> cos(inf)
nan
>>> nprint(chop(taylor(cos, 0, 6)))
```

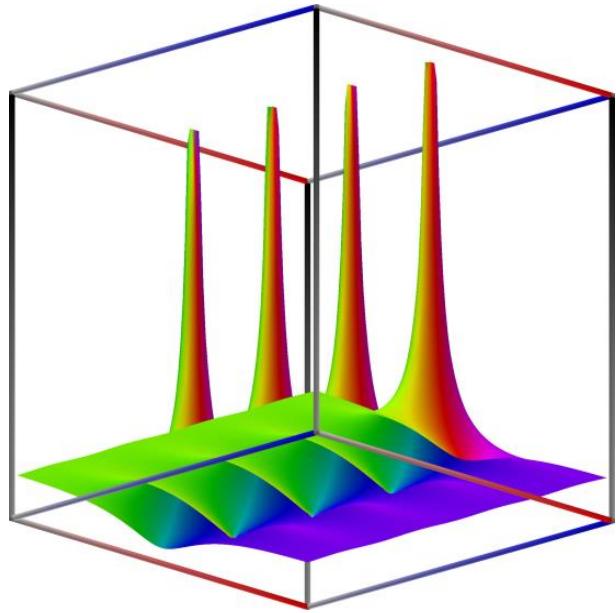
[1.0, 0.0, -0.5, 0.0, 0.0416667, 0.0, -0.00138889]

---

### 5.4.5 Tangent: $\tan(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.8:** Surface plots of  $z = \tan(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function **tan(z As mpNum) As mpNum**

---

The function **tan** returns complex tangent of  $z$

**Parameter:**

$z$ : A complex number.

The function **cplxTan(z)** returns the tangent tangent of  $z$ :

$$\tan(z) = \frac{\sin(z)}{\cos(z)} = \frac{\sin(2x) + i \sinh(2y)}{\cos(2x) + i \cosh(2y)} \quad (5.4.5)$$

Computes the tangent of  $x$ ,  $\tan(x) = \frac{\sin(x)}{\cos(x)}$ . The tangent function is singular at  $x = (n + \frac{1}{2})\pi$ , but  $\tan(x)$  always returns a finite result since  $(n + \frac{1}{2})\pi$  cannot be represented exactly using floating-point arithmetic.

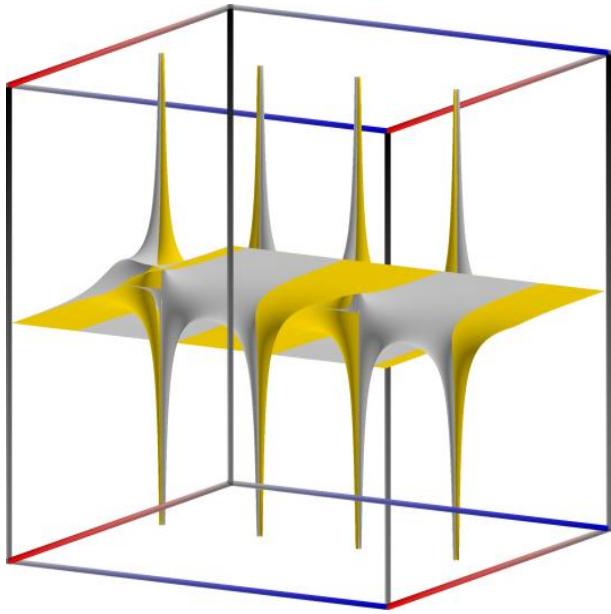
---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> tan(pi/3)
1.732050807568877293527446
>>> tan(100000001)
-0.2015625081449864533091058
>>> tan(2+3j)
```

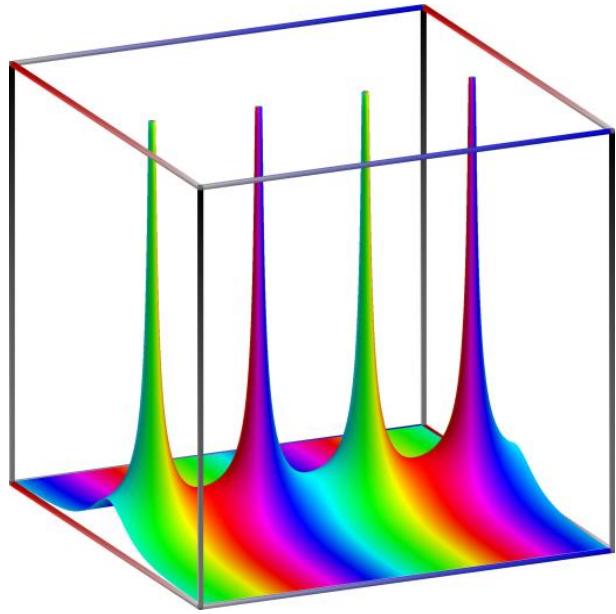
```
(-0.003764025641504248292751221 + 1.003238627353609801446359j)
>>> tan(inf)
nan
>>> nprint(chop(taylor(tan, 0, 6)))
[0.0, 1.0, 0.0, 0.333333, 0.0, 0.133333, 0.0]
```

---

### 5.4.6 Secant: $\sec(z) = 1/\cos(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.9:** Surface plots of  $z = \sec(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `sec(z As mpNum) As mpNum`

---

The function `sec` returns the complex secant of  $z$

**Parameter:**

$z$ : A complex number.

The function `cplxSec(z)` returns the complex secant of  $z$ :

$$\sec(z) = 1/\cos(z). \quad (5.4.6)$$

Computes the secant of  $x$ ,  $\sec(x) = \frac{1}{\cos(x)}$ . The secant function is singular at  $x = (n + \frac{1}{2})\pi$ , but  $\sec(x)$  always returns a finite result since  $(n + \frac{1}{2})\pi$  cannot be represented exactly using floating-point arithmetic.

---

```

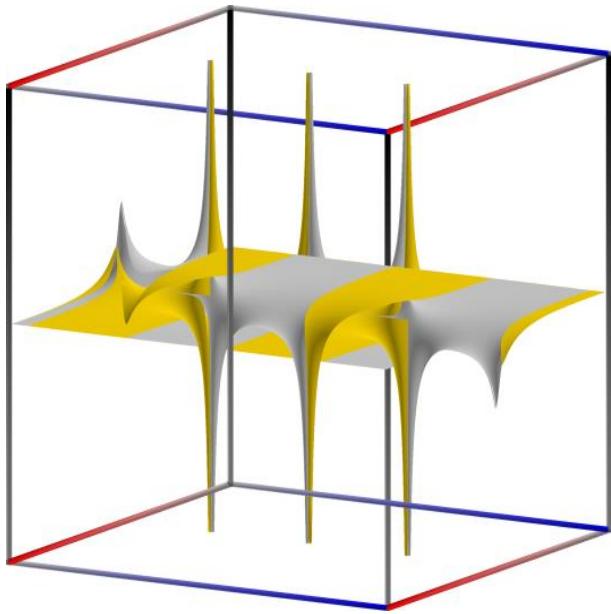
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> sec(pi/3)
2.0
>>> sec(10000001)
-1.184723164360392819100265
>>> sec(2+3j)
(-0.04167496441114427004834991 + 0.0906111371962375965296612j)
>>> sec(inf)

```

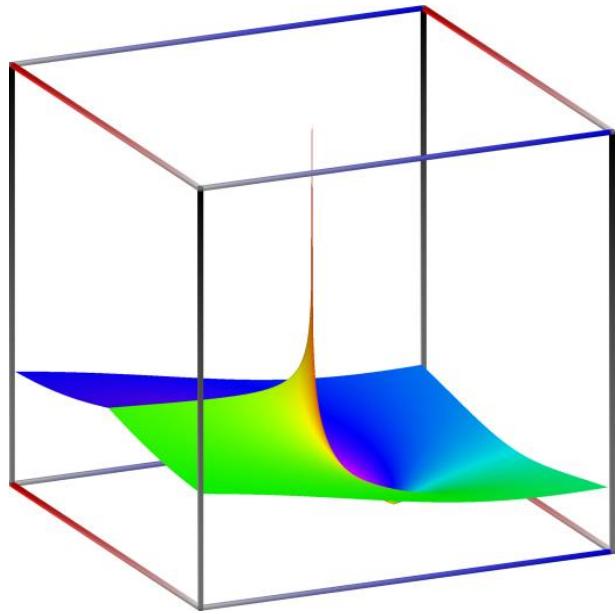
```
nan
>>> npprint(chop(taylor(sec, 0, 6)))
[1.0, 0.0, 0.5, 0.0, 0.208333, 0.0, 0.0847222]
```

---

### 5.4.7 Cosecant: $\csc(z) = 1/\sin(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.10:** Surface plots of  $z = \csc(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `csc(z As mpNum) As mpNum`

---

The function `csc` returns the complex cosecant of  $z$

**Parameter:**

$z$ : A complex number.

The function `cplxCsc(z)` returns the complex cosecant of  $z$ :

$$\sec(z) = 1/\sin(z). \quad (5.4.7)$$

Computes the cosecant of  $x$ ,  $\csc(x) = \frac{1}{\sin(x)}$ . This cosecant function is singular at  $x = n\pi$ , but with the exception of the point  $x = 0$ , `csc(x)` returns a finite result since  $n\pi$  cannot be represented exactly using floating-point arithmetic.

---

```

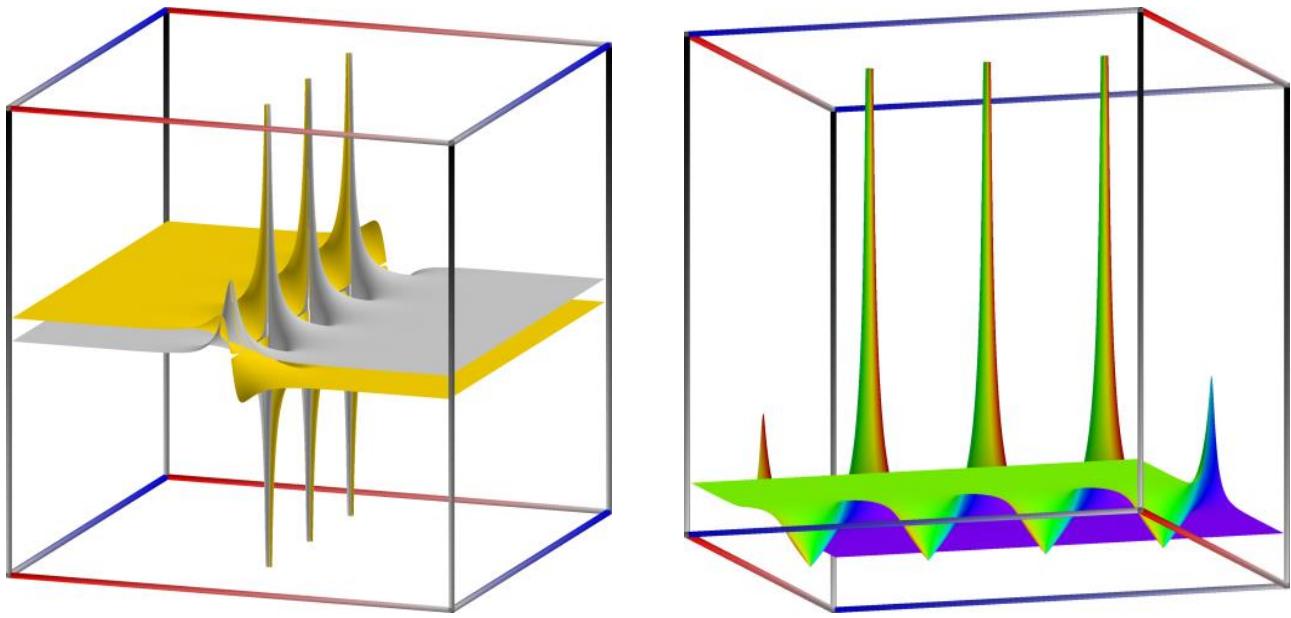
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> csc(pi/3)
1.154700538379251529018298
>>> csc(10000001)
-1.864910497503629858938891
>>> csc(2+3j)
(0.09047320975320743980579048 + 0.04120098628857412646300981j)
>>> csc(inf)

```

`nan`

---

### 5.4.8 Cotangent: $\cot(z) = 1/\tan(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.11:** Surface plots of  $z = \cot(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `cot(z As mpNum) As mpNum`

---

The function `cot` returns the complex cotangent of  $z$ .

**Parameter:**

$z$ : A complex number.

The function `cplxCot(z)` returns the complex cotangent of  $z$ :

$$\cot(z) = \frac{\cos(z)}{\sin(z)} = \frac{\sin(2x) - i \sinh(2y)}{\cosh(2y) - i \cos(2x)} \quad (5.4.8)$$

Computes the cotangent of  $x$ ,  $\cot(x) = \frac{1}{\tan(x)} = \frac{\cos(x)}{\sin(x)}$ . The cotangent function is singular at  $x = n\pi$ , but with the exception of the point  $x = 0$ ,  $\csc(x)$  returns a finite result since  $n\pi$  cannot be represented exactly using floating-point arithmetic.

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> cot(pi/3)
0.5773502691896257645091488
>>> cot(10000001)
1.574131876209625656003562
>>> cot(2+3j)
(-0.003739710376336956660117409 - 0.9967577965693583104609688j)

```

```
>>> cot(inf)
nan
```

---

### 5.4.9 Sinc function

#### 5.4.9.1 `sinc(x)`

`sinc(x)` computes the unnormalized sinc function, defined as

$$\text{sinc}(x) = \begin{cases} \sin(x)/x & \text{for } x \neq 0 \\ 1 & \text{for } x = 0. \end{cases} \quad (5.4.9)$$

See `sincpi()` for the normalized sinc function.

Simple values and limits include

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> sinc(0)
1.0
>>> sinc(1)
0.841470984807897
>>> sinc(inf)
0.0
```

---

The integral of the sinc function is the sine integral Si:

---

```
>>> quad(sinc, [0, 1])
0.946083070367183
>>> si(1)
0.946083070367183
```

---

#### 5.4.9.2 `sincpi(x)`

`sincpi(x)` computes the normalized sinc function, defined as

$$\text{sinc}_\pi(x) = \begin{cases} \sin(\pi x)/(\pi x) & \text{for } x \neq 0 \\ 1 & \text{for } x = 0. \end{cases} \quad (5.4.10)$$

Equivalently, we have  $\text{sinc}_\pi(x) = \text{sinc}(\pi x)$ . The normalization entails that the function integrates to unity over the entire real line:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> quadosc(sincpi, [-inf, inf], period=2.0)
1.0
```

---

Like, `sinpi()`, `sincpi()` is evaluated accurately at its roots:

---

```
>>> sincpi(10)
0.0
```

---

### 5.4.10 Trigonometric functions with modified argument

#### 5.4.10.1 `cospi(x)`

Computes  $\cos(\pi x)$ , more accurately than the expression  $\cos(\pi * x)$ :

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> cospi(10**10), cos(pi*(10**10))
(1.0, 0.999999999997493)
>>> cospi(10**10+0.5), cos(pi*(10**10+0.5))
(0.0, 1.59960492420134e-6)
```

---

#### 5.4.10.2 sinpi(x)

Computes  $\sin(\pi x)$ , more accurately than the expression  $\sin(\pi \cdot x)$ :

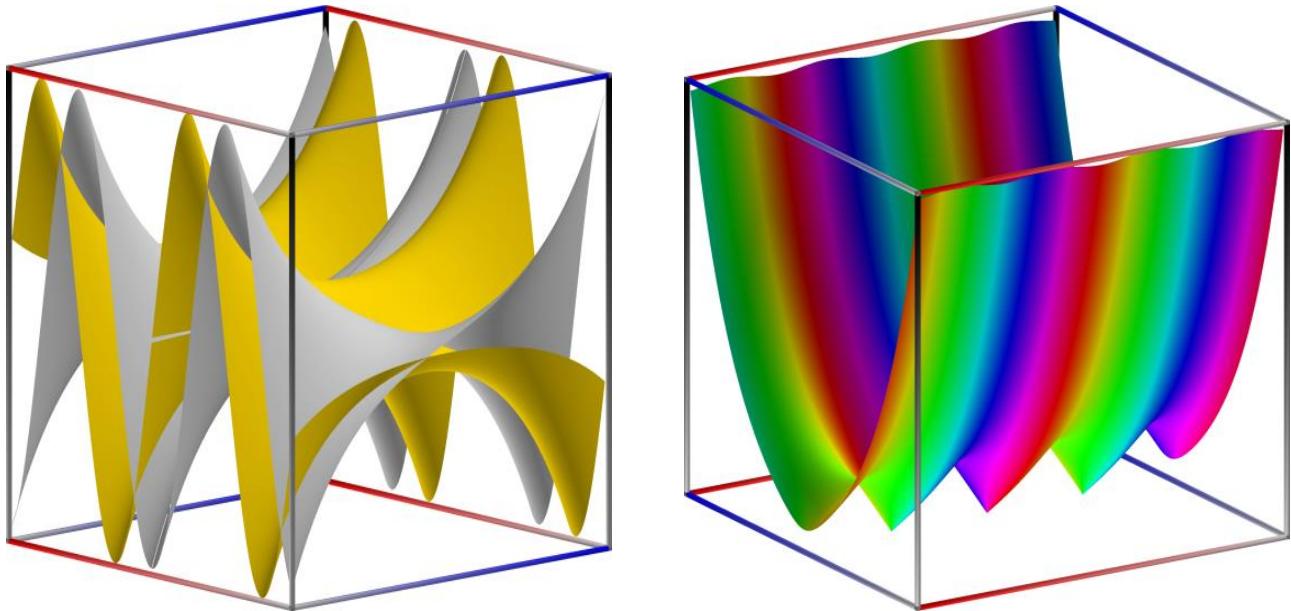
---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> sinpi(10**10), sin(pi*(10**10))
(0.0, -2.23936276195592e-6)
```

---

## 5.5 Hyperbolic Functions

### 5.5.1 Hyperbolic Sine: $\sinh(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.12:** Surface plots of  $z = \sinh(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `sinh(z As mpNum) As mpNum`

---

The function `sinh` returns the complex hyperbolic sine of  $z$

**Parameter:**

$z$ : A complex number.

The function `cplxSinh(z)` returns the complex hyperbolic sine of  $z$ :

$$\sinh(z) = \sinh(x) \cos(y) + i \cosh(x) \sin(y). \quad (5.5.1)$$

Computes the hyperbolic sine of  $x$ ,  $\sinh(x) = (e^x - e^{-x})/2$ . Values and limits include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> sinh(0)
0.0
>>> sinh(1)
1.175201193643801456882382
>>> sinh(-inf), sinh(+inf)
(-inf, +inf)
```

---

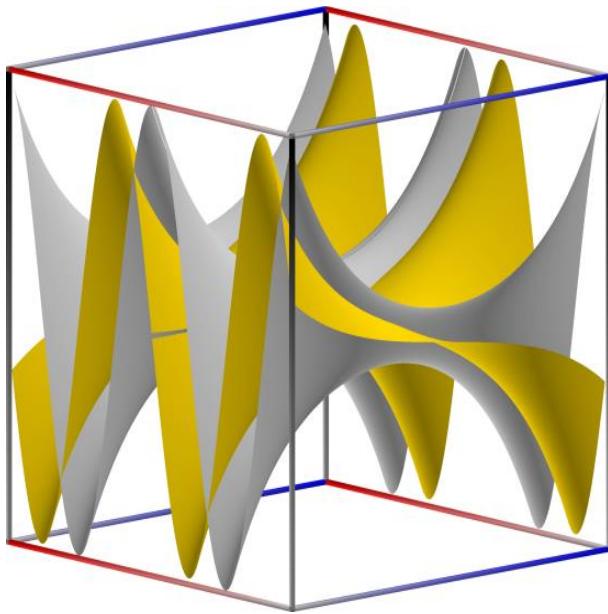
Generalized to complex numbers, the hyperbolic sine is essentially a sine with a rotation  $i$  applied to the argument; more precisely,  $\sinh(x) = -i \sin(ix)$ :

---

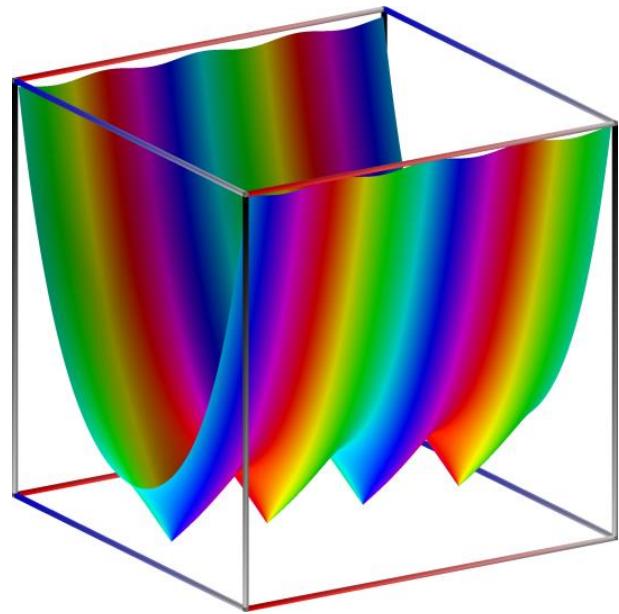
```
>>> sinh(2+3j)
(-3.590564589985779952012565 + 0.5309210862485198052670401j)
>>> j*sin(3-2j)
(-3.590564589985779952012565 + 0.5309210862485198052670401j)
```

---

### 5.5.2 Hyperbolic Cosine: $\cosh(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.13:** Surface plots of  $z = \cosh(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `cosh(z As mpNum) As mpNum`

---

The function `cosh` returns the complex hyperbolic cosine of  $z$

**Parameter:**

$z$ : A complex number.

The function `cplxCosh(z)` returns the complex hyperbolic cosine of  $z$ :

$$\cosh(z) = \cosh(x) \cos(y) + i \sinh(x) \sin(y). \quad (5.5.2)$$

Computes the hyperbolic cosine of  $x$ ,  $\cosh(x) = (e^x + e^{-x})/2$ . Values and limits include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> cosh(0)
1.0
>>> cosh(1)
1.543080634815243778477906
>>> cosh(-inf), cosh(+inf)
(+inf, +inf)
```

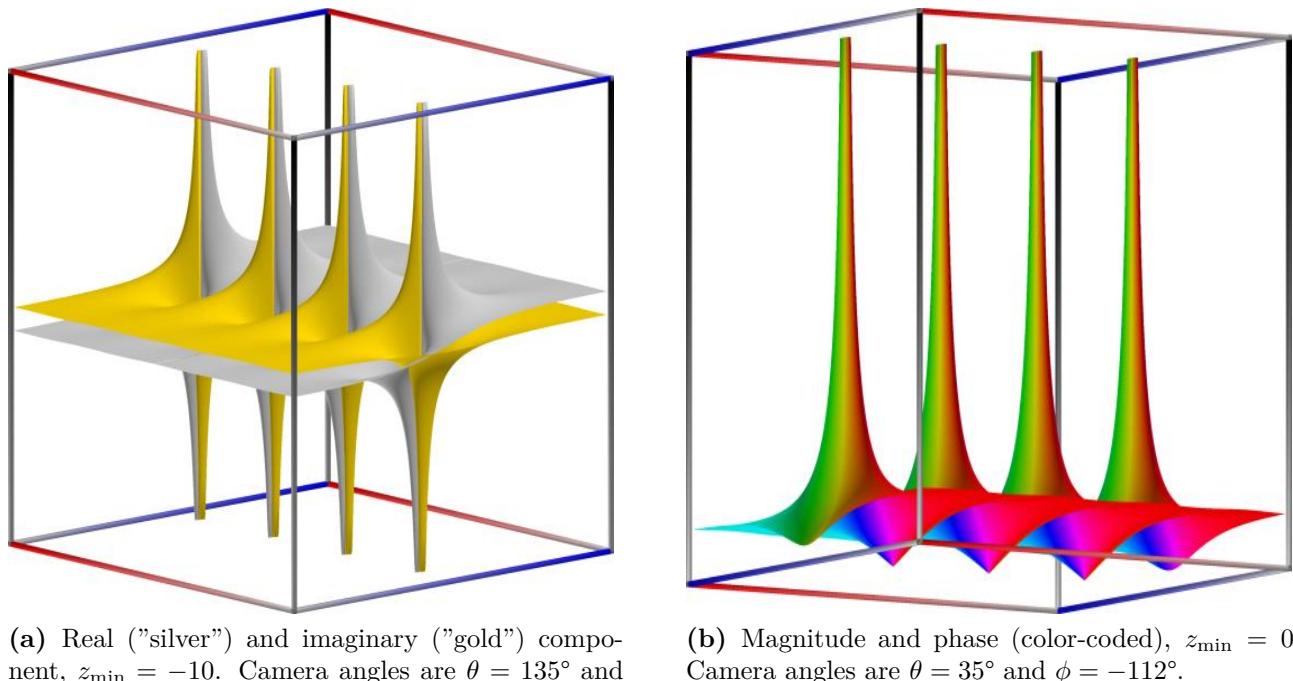
---

Generalized to complex numbers, the hyperbolic cosine is equivalent to a cosine with the argument rotated in the imaginary direction, or  $\cosh(x) = \cos(ix)$ :

```
>>> cosh(2+3j)
(-3.724545504915322565473971 + 0.5118225699873846088344638j)
>>> cos(3-2j)
(-3.724545504915322565473971 + 0.5118225699873846088344638j)
```

---

### 5.5.3 Hyperbolic Tangent: $\tanh(z)$



**Figure 5.14:** Surface plots of  $z = \tanh(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function **tanh(z As mpNum)** As mpNum

---

The function **tanh** returns the complex hyperbolic tangent of  $z$

**Parameter:**

$z$ : A complex number.

The function **cplxTanh(z)** returns the complex hyperbolic tangent of  $z$ :

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{\sinh(2x) + i \sin(2y)}{\cosh(2x) + i \cos(2y)} \quad (5.5.3)$$

Computes the hyperbolic tangent of  $x$ ,  $\tanh(x) = \sinh(x)/\cosh(x)$ . Values and limits include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> tanh(0)
0.0
>>> tanh(1)
0.7615941559557648881194583
>>> tanh(-inf), tanh(inf)
(-1.0, 1.0)
```

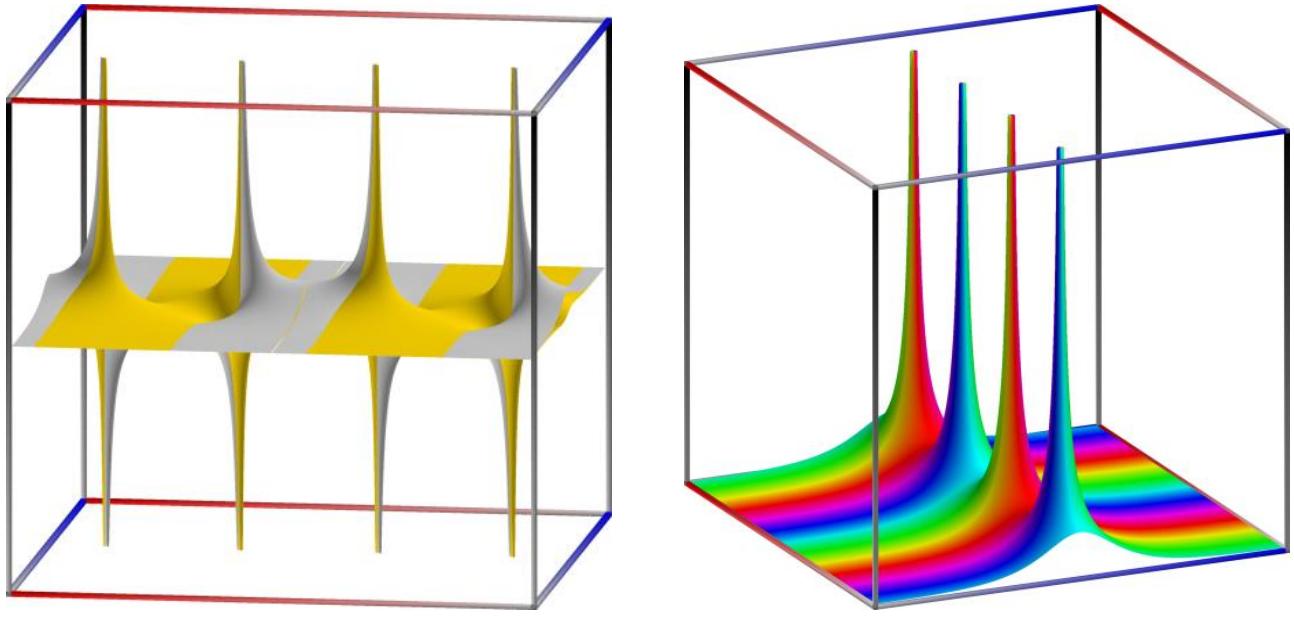
---

Generalized to complex numbers, the hyperbolic tangent is essentially a tangent with a rotation  $i$  applied to the argument; more precisely,  $\tanh(x) = -i \tan(ix)$ :

```
>>> tanh(2+3j)
(0.9653858790221331242784803 - 0.009884375038322493720314034j)
>>> j*tan(3-2j)
(0.9653858790221331242784803 - 0.009884375038322493720314034j)
```

---

### 5.5.4 Hyperbolic Secant: $\text{sech}(x) = 1/\cosh(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.15:** Surface plots of  $z = \text{sech}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function **sech(z As mpNum) As mpNum**

---

The function **sech** returns the complex hyperbolic secant of  $z$

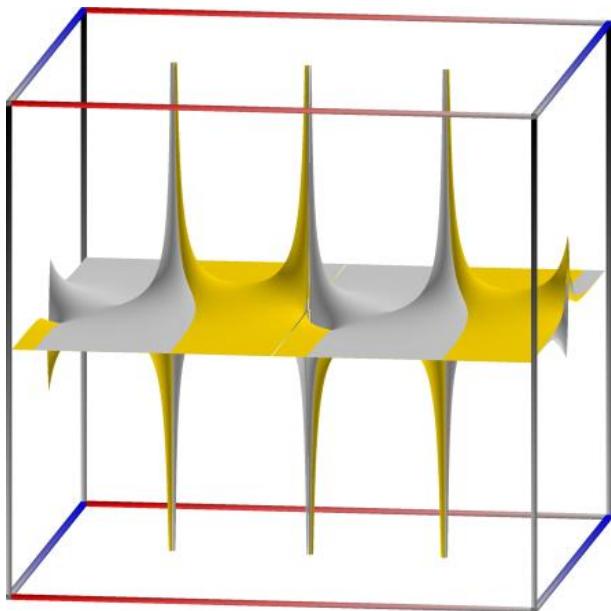
**Parameter:**

**z:** A complex number.

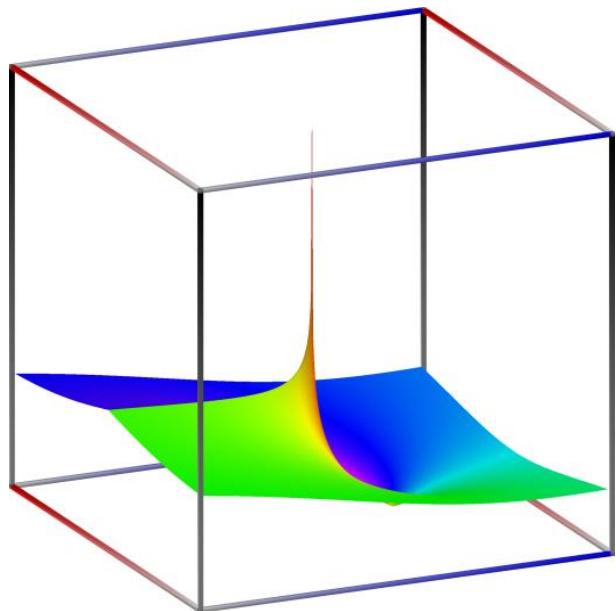
The function **cplxSech(z)** returns the complex hyperbolic secant of  $z$ :

$$\text{sech}(z) = 1/\cosh(z). \quad (5.5.4)$$

### 5.5.5 Hyperbolic Cosecant: $\text{csch}(x) = 1/\sinh(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ . wrong, missing

**Figure 5.16:** Surface plots of  $z = \text{csch}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function `csch(z As mpNum) As mpNum`

---

The function `csch` returns the complex hyperbolic cosecant of  $z$

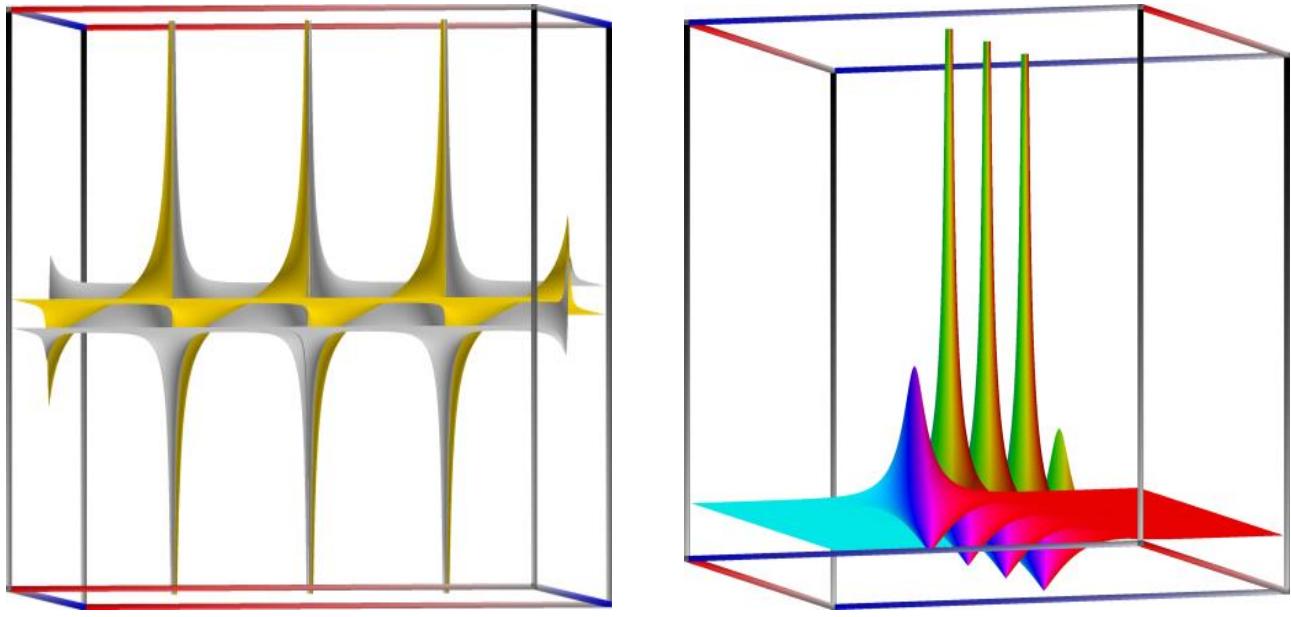
**Parameter:**

$z$ : A complex number.

The function `csch(z)` returns the complex hyperbolic cosecant of  $z$ :

$$\text{csch}(z) = 1/\sinh(z). \quad (5.5.5)$$

### 5.5.6 Hyperbolic Cotangent: $\coth(x) = 1/\tanh(z)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .

(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.17:** Surface plots of  $z = \coth(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

---

#### Function **coth(z As mpNum) As mpNum**

---

The function **coth** returns the complex hyperbolic cotangent of  $z$

**Parameter:**

$z$ : A complex number.

The function **coth(z)** returns the complex hyperbolic cotangent of  $z$ :

$$\coth(z) = \frac{\cosh(z)}{\sinh(z)} = \frac{\sinh(2x) - i \sin(2y)}{\cosh(2x) - i \cos(2y)} \quad (5.5.6)$$

## 5.6 Inverse Trigonometric Functions

The formulas in section follow [Olver et al. \(2010\)](#), equations 4.23.34 - 4.23.38 for sections [5.6.1](#) - [5.6.3](#), equation 4.23.9 for section [5.6.5](#), and [Abramowitz & Stegun. \(1970\)](#), equations 4.6.14 - 4.6.19 for sections [5.7.1](#) - [5.7.4](#).

### 5.6.1 Arcsine: $\text{asin}(z)$

---

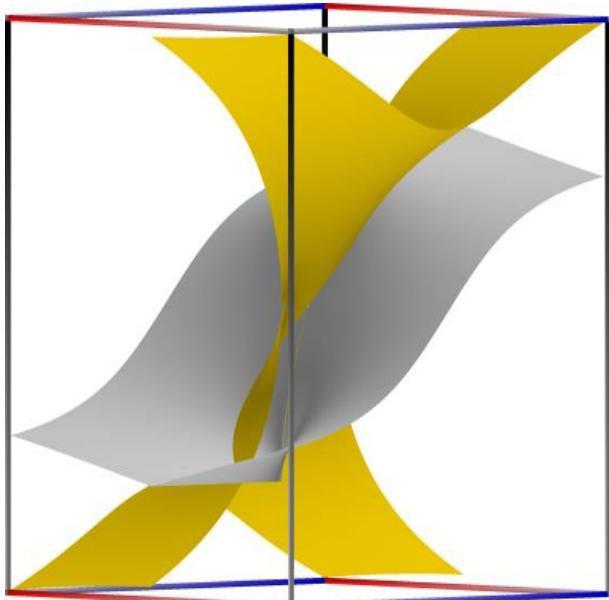
Function **asin(z As mpNum)** As mpNum

---

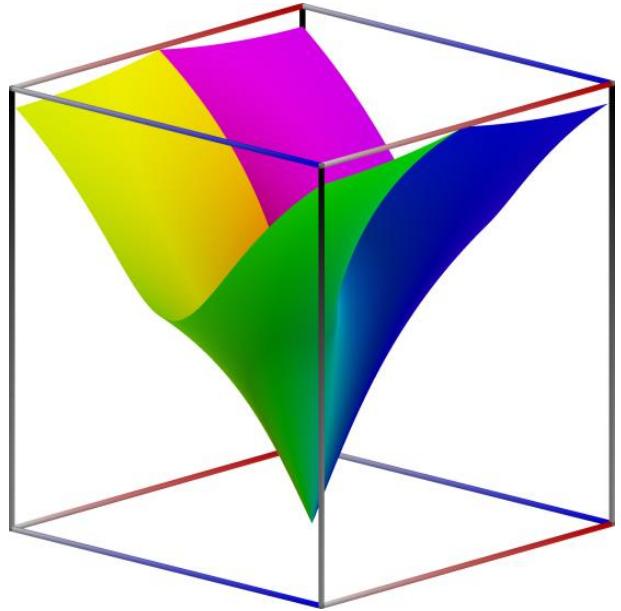
The function **asin** returns the inverse complex sine of  $z$

**Parameter:**

$z$ : A complex number.



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.18:** Surface plots of  $z = \text{asin}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section [2.3.3](#) for more information about charts for complex functions.

The function **cplxASin(z)** returns the inverse complex sine of  $z = x + iy$ :

$$\arcsin(z) = \arcsin(\beta) + i \ln \left( \alpha + \sqrt{\alpha^2 - 1} \right), \quad \text{where} \quad (5.6.1)$$

$$\alpha = \frac{1}{2} \sqrt{(x+1)^2 + y^2} + \frac{1}{2} \sqrt{(x-1)^2 + y^2}, \quad (5.6.2)$$

$$\beta = \frac{1}{2} \sqrt{(x+1)^2 + y^2} - \frac{1}{2} \sqrt{(x-1)^2 + y^2}, \quad (5.6.3)$$

and  $x \in [-1, 1]$ .

Computes the inverse sine or arcsine of  $x$ ,  $\sin^{-1}(x)$ . Since  $-1 \leq \cos(x) \leq 1$  for real  $x$ , the inverse sine is real-valued only for  $-1 < x < 1$ . On this interval, `asin()` is defined to be a monotonically decreasing function assuming values between  $-\pi/2$  and  $\pi/2$ .

Basic values are:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> asin(-1)
-1.570796326794896619231322
>>> asin(0)
0.0
>>> asin(1)
1.570796326794896619231322
>>> nprint(chop(taylor(asin, 0, 6)))
[0.0, 1.0, 0.0, 0.166667, 0.0, 0.075, 0.0]
```

---

`asin()` is defined so as to be a proper inverse function of  $\sin(\theta)$  for  $-\pi/2 < \theta < \pi/2$ . We have  $\sin(\sin^{-1}(x) = x)$  for all  $x$ , but  $\sin^{-1}(\sin(x)) = x$  only for  $-\pi/2 \leq \Re[x] < \pi/2$ :

---

```
>>> for x in [1, 10, -1, 1+3j, -2+3j]:
...     print("%s %s" % (chop(sin(asin(x))), asin(sin(x))))
...
1.0 1.0
10.0 -0.5752220392306202846120698
-1.0 -1.0
(1.0 + 3.0j) (1.0 + 3.0j)
(-2.0 + 3.0j) (-1.141592653589793238462643 - 3.0j)
```

---

The inverse sine has two branch points:  $x = \pm 1$ . `asin()` places the branch cuts along the line segments  $(-\infty, -1)$  and  $(+1, +\infty)$ . In general,

$$\sin^{-1}(x) = -i \log \left( ix + \sqrt{1 - x^2} \right) \quad (5.6.4)$$

where the principal-branch log and square root are implied.

## 5.6.2 Arccosine: $\text{acos}(z)$

---

### Function $\text{acos}(z)$ As $\text{mpNum}$ As $\text{mpNum}$

---

The function  $\text{acos}$  returns the inverse complex cosine of  $z$

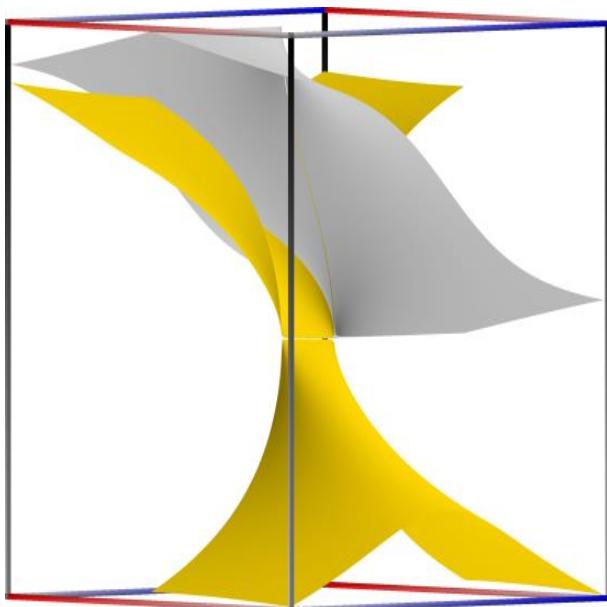
**Parameter:**

$z$ : A complex number.

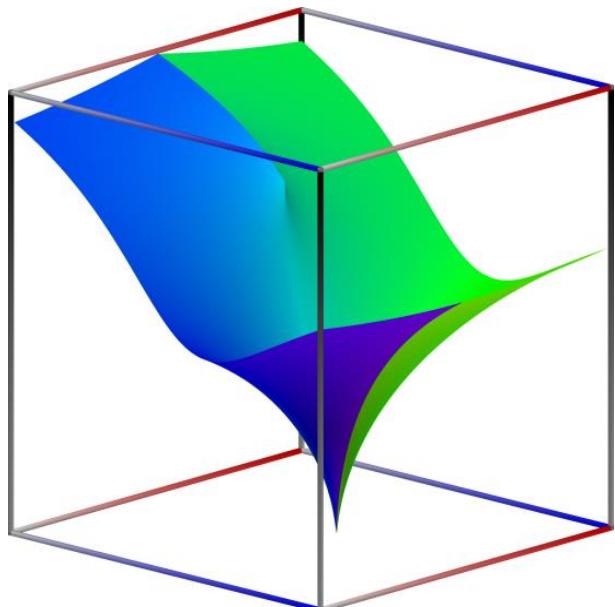
The function  $\text{cplxACos}(z)$  returns the inverse complex cosine of  $z = x + iy$ :

$$\arccos(z) = \arccos(\beta) - i \ln \left( \alpha + \sqrt{\alpha^2 - 1} \right), \quad \text{where} \quad (5.6.5)$$

$\alpha$  and  $\beta$  are defined in equations 5.6.2 and 5.6.3, and  $x \in [-1, 1]$ .



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.19:** Surface plots of  $z = \text{acos}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

Computes the inverse cosine or arccosine of  $x$ ,  $\cos^{-1}(x)$ . Since  $-1 \leq \cos(x) \leq 1$  for real  $x$ , the inverse cosine is real-valued only for  $-1 < x < 1$ . On this interval,  $\text{acos}()$  is defined to be a monotonically decreasing function assuming values between  $+\pi$  and 0.

Basic values are:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> acos(-1)
3.141592653589793238462643
>>> acos(0)
1.570796326794896619231322
```

---

```
>>> acos(1)
0.0
>>> nprint(chop(taylor(acos, 0, 6)))
[1.5708, -1.0, 0.0, -0.166667, 0.0, -0.075, 0.0]
```

---

`acos()` is defined so as to be a proper inverse function of  $\cos(\theta)$  for  $0 < \theta < \pi$ . We have  $\cos(\cos^{-1}(x) = x)$  for all  $x$ , but  $\cos^{-1}(\cos(x)) = x$  only for  $0 \leq \Re[x] < \pi$ :

---

```
>>> for x in [1, 10, -1, 2+3j, 10+3j]:
...     print("%s %s" % (cos(acos(x)), acos(cos(x))))
...
1.0 1.0
(10.0 + 0.0j) 2.566370614359172953850574
-1.0 1.0
(2.0 + 3.0j) (2.0 + 3.0j)
(10.0 + 3.0j) (2.566370614359172953850574 - 3.0j)
```

---

The inverse cosine has two branch points:  $x = \pm 1$ . `acos()` places the branch cuts along the line segments  $(-\infty, -1)$  and  $(+1, +\infty)$ . In general,

$$\cos^{-1}(x) = \frac{\pi}{2} + i \log \left( ix + \sqrt{1 - x^2} \right) \quad (5.6.6)$$

where the principal-branch log and square root are implied.

### 5.6.3 Arctangent: $\text{atan}(z)$

---

#### Function $\text{atan}(z)$ As $\text{mpNum}$ ) As $\text{mpNum}$

---

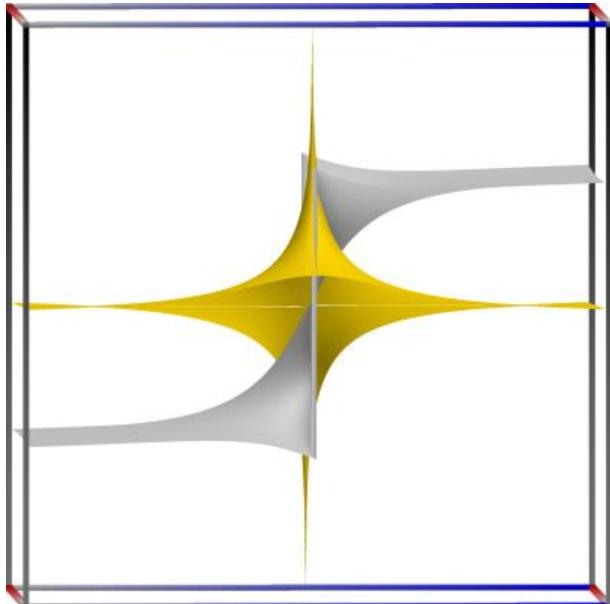
The function  $\text{atan}$  returns the inverse complex tangent of  $z$

**Parameter:**

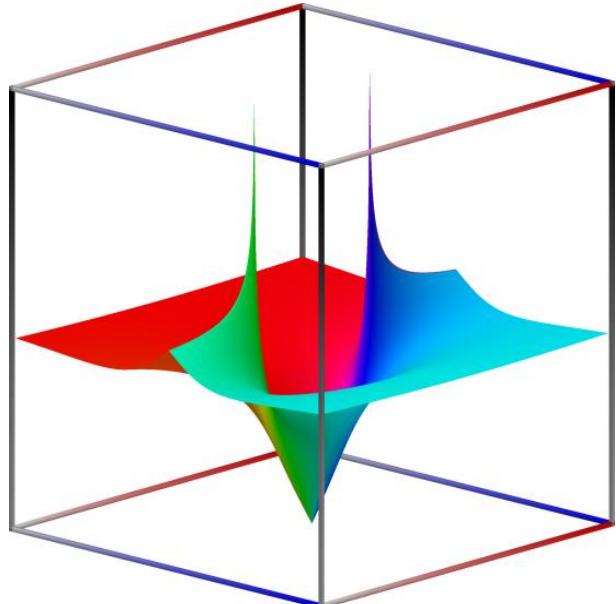
$z$ : A complex number.

The function  $\text{cplxATan}(z)$  returns the inverse complex tangent of  $z = x + iy$ :

$$\arctan(z) = \frac{1}{2} \arctan \left( \frac{2x}{1 - x^2 - y^2} \right) + \frac{1}{4}i \ln \left( \frac{x^2 + (y + 1)^2}{x^2 + (y - 1)^2} \right), \quad \text{where } |z| < 1. \quad (5.6.7)$$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.20:** Surface plots of  $z = \text{atan}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

Computes the inverse tangent or arctangent of  $x$ ,  $\tan^{-1}(x)$ . This is a real-valued function for all real  $x$ , with range  $(-\pi/2, \pi/2)$ .

Basic values are:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> atan(-inf)
-1.570796326794896619231322
>>> atan(-1)
-0.7853981633974483096156609
>>> atan(0)
```

---

```
0.0
>>> atan(1)
0.7853981633974483096156609
>>> atan(inf)
1.570796326794896619231322
>>> nprint(chop(taylor(atan, 0, 6)))
[0.0, 1.0, 0.0, -0.333333, 0.0, 0.2, 0.0]
```

---

The inverse tangent is often used to compute angles. However, the atan2 function is often better for this as it preserves sign (see atan2()).

atan() is defined so as to be a proper inverse function of  $\tan(\theta)$  for  $-\pi/2 < \theta < \pi/2$ . We have  $\tan(\tan^{-1}(x) = x)$  for all  $x$ , but  $\tan^{-1}(\tan(x)) = x$  only for  $-\pi/2 \leq \Re[x] < \pi/2$ :

---

```
>>> mp.dps = 25
>>> for x in [1, 10, -1, 1+3j, -2+3j]:
...     print("%s %s" % (tan(atan(x)), atan(tan(x))))
...
1.0 1.0
10.0 0.5752220392306202846120698
-1.0 -1.0
(1.0 + 3.0j) (1.00000000000000000000000000000001 + 3.0j)
(-2.0 + 3.0j) (1.141592653589793238462644 + 3.0j)
```

---

The inverse tangent has two branch points:  $x = \pm i$ . atan() places the branch cuts along the line segments  $(-i\infty, -i)$  and  $(+i, +i\infty)$ . In general,

$$\tan^{-1}(x) = \frac{i}{2} (\log(1 - ix) - \log(1 + ix)) \quad (5.6.8)$$

where the principal-branch log and square root are implied.

### 5.6.4 Arc-tangent, version with 2 arguments: `atan2(x, y)`

---

Function **Atan2**(*x* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function Atan2 returns the value of the arc-tangent of *x* in radians.

**Parameters:**

- x*: A real number.
- y*: A real number.

Computes the two-argument arctangent,  $\text{atan2}(y, x)$ , giving the signed angle between the positive *x*-axis and the point  $(x, y)$  in the 2D plane. This function is defined for real *x* and *y* only.

The two-argument arctangent essentially computes  $\text{atan}(y/x)$ , but accounts for the signs of both *x* and *y* to give the angle for the correct quadrant. The following examples illustrate the difference:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> atan2(1,1), atan(1/1.)
(0.785398163397448, 0.785398163397448)
>>> atan2(1,-1), atan(1/-1.)
(2.35619449019234, -0.785398163397448)
>>> atan2(-1,1), atan(-1/1.)
(-0.785398163397448, -0.785398163397448)
>>> atan2(-1,-1), atan(-1/-1.)
(-2.35619449019234, 0.785398163397448)
```

---

The angle convention is the same as that used for the complex argument; see `arg()`.

### 5.6.5 Arccotangent: $\text{acot}(z)$

---

Function **acot(z As mpNum) As mpNum**

---

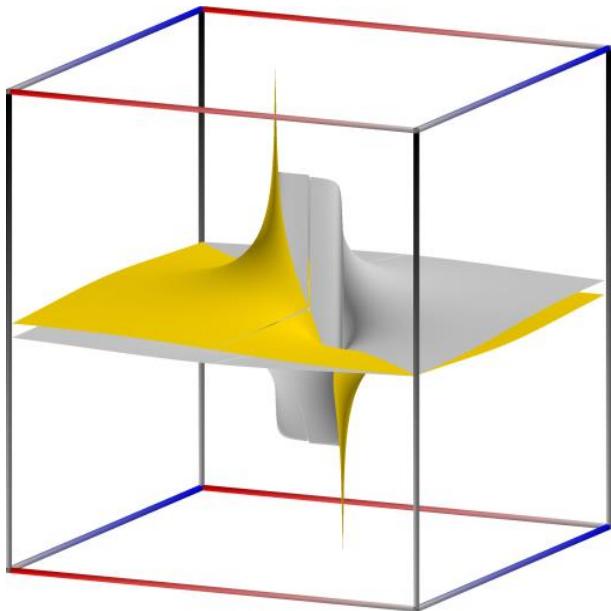
The function **acot** returns the inverse complex cotangent of  $z$

**Parameter:**

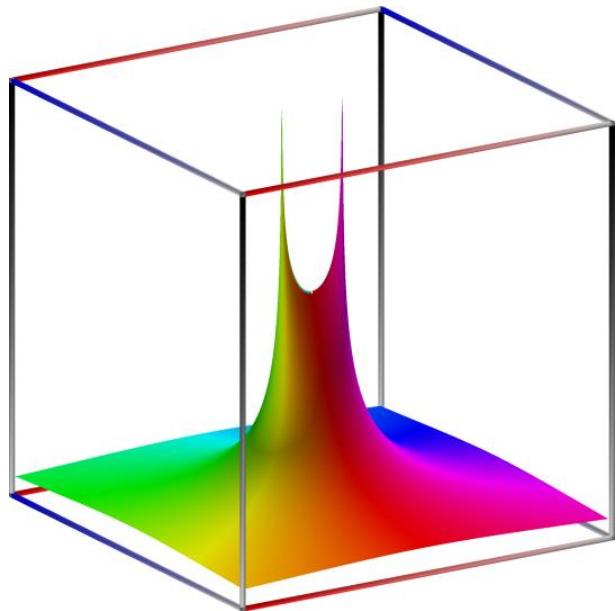
$z$ : A complex number.

The function **cplxACot(z)** returns the inverse complex cotangent of  $z$ :

$$\text{arccot}(z) = \arctan(1/z), \quad z \neq \pm i. \quad (5.6.9)$$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.21:** Surface plots of  $z = \text{acot}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

### 5.6.6 **asec(x)**

Computes the inverse secant of  $x$ ,  $\sec^{-1} = \cos^{-1}(1/x)$ .

### 5.6.7 **acsc(x)**

Computes the inverse cosecant of  $x$ ,  $\csc^{-1} = \sin^{-1}(1/x)$ .

## 5.7 Inverse Hyperbolic Functions

### 5.7.1 Inverse Hyperbolic Sine: `asinh(z)`

---

Function `asinh(z As mpNum) As mpNum`

---

The function `asinh` returns the inverse complex hyperbolic sine of  $z$

**Parameter:**

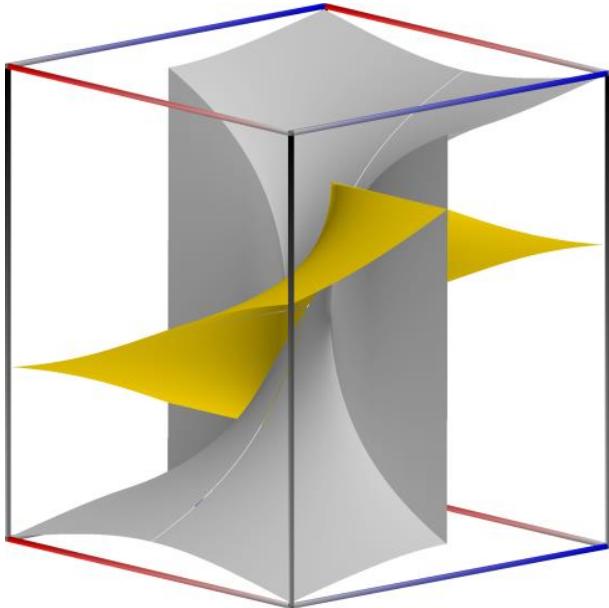
$z$ : A complex number.

The function `cplxASinh(z)` returns the inverse complex hyperbolic sine of  $z$ :

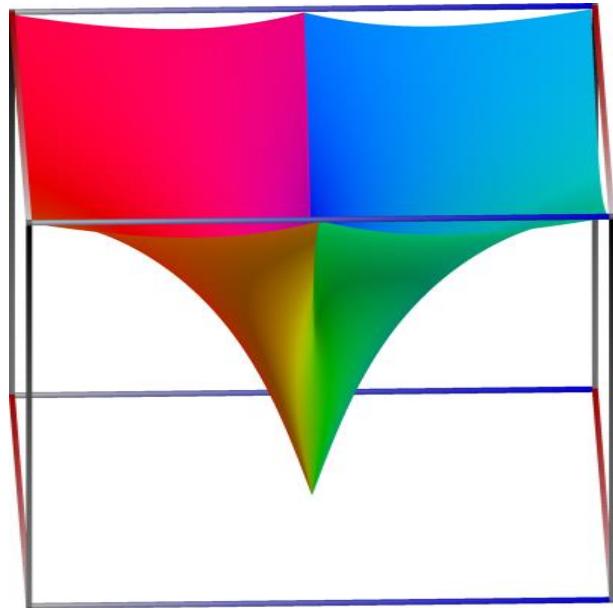
$$\operatorname{arcsinh}(z) = -i \operatorname{arcsin}(iz), \quad (5.7.1)$$

where  $\operatorname{arcsin}(z)$  is defined in section 5.6.1

Computes the inverse hyperbolic sine of  $x$ ,  $\sinh^{-1}(x) = \log(x + \sqrt{1 + x^2})$ .



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.22:** Surface plots of  $z = \operatorname{asinh}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

### 5.7.2 Inverse Hyperbolic Cosine: $\text{acosh}(z)$

---

Function **acosh(z As mpNum) As mpNum**

---

The function  $\text{acosh}$  returns the inverse complex hyperbolic cosine of  $z$

**Parameter:**

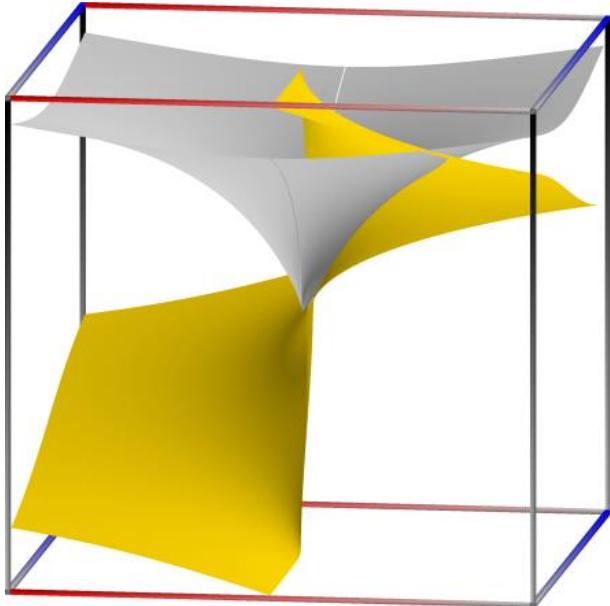
$z$ : A complex number.

The function  $\text{cplxACosh}(z)$  returns the inverse complex hyperbolic cosine of  $z$ :

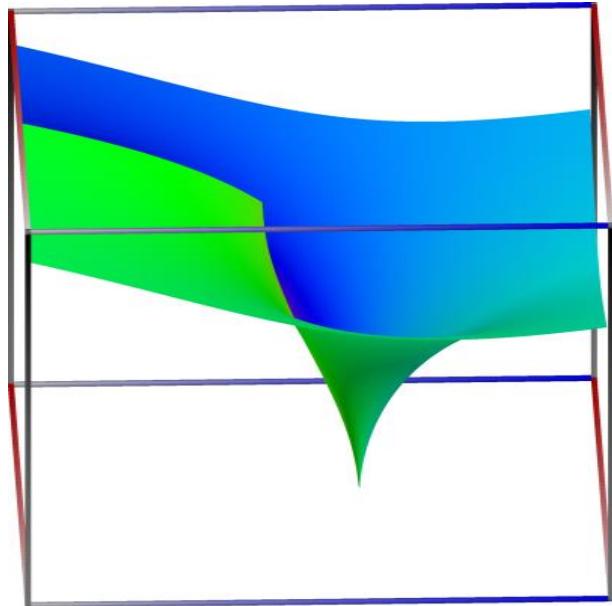
$$\text{arccosh}(z) = \pm i \arccos(z), \quad (5.7.2)$$

where  $\arccos(z)$  is defined in section 5.6.2

Computes the inverse hyperbolic cosine of  $x$ ,  $\cosh^{-1}(x) = \log(x + \sqrt{x + 1}\sqrt{x - 1})$ .



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.23:** Surface plots of  $z = \text{acosh}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

### 5.7.3 Inverse Hyperbolic Tangent: $\operatorname{atanh}(z)$

---

Function **atanh(z As mpNum)** As mpNum

---

The function **atanh** returns the inverse complex hyperbolic tangent of  $z$

**Parameter:**

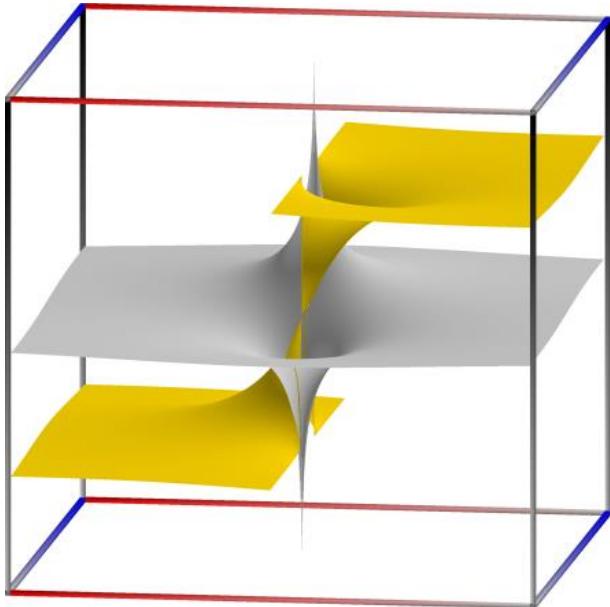
$z$ : A complex number.

The function **cplxATanh(z)** returns the inverse complex hyperbolic tangent of  $z$ :

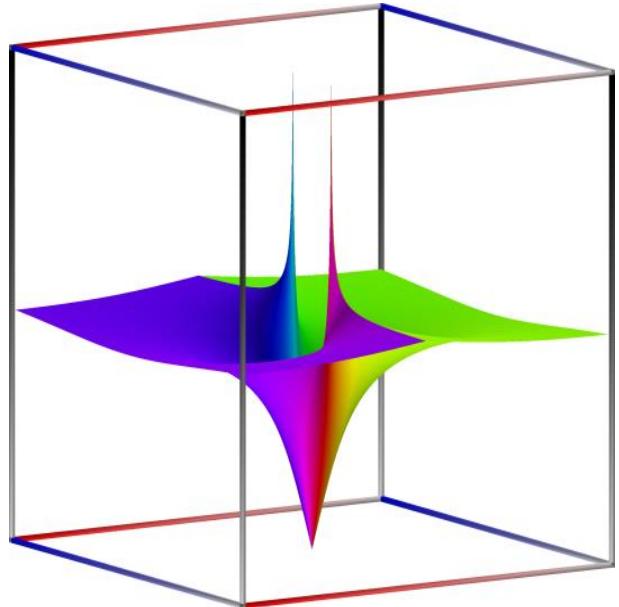
$$\operatorname{arctanh}(z) = -i \operatorname{arctan}(z), \quad (5.7.3)$$

where  $\operatorname{arctan}(z)$  is defined in section 5.6.3

Computes the inverse hyperbolic tangent of  $x$ ,  $\tanh^{-1}(x) = \frac{1}{2}(\log(1+x) - \log(1-x))$ .



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.24:** Surface plots of  $z = \operatorname{atanh}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

### 5.7.4 Inverse Hyperbolic Cotangent: $\text{acoth}(z)$

---

Function **acoth(z As mpNum) As mpNum**

---

The function  $\text{acoth}$  returns the inverse complex hyperbolic cotangent of  $z$

**Parameter:**

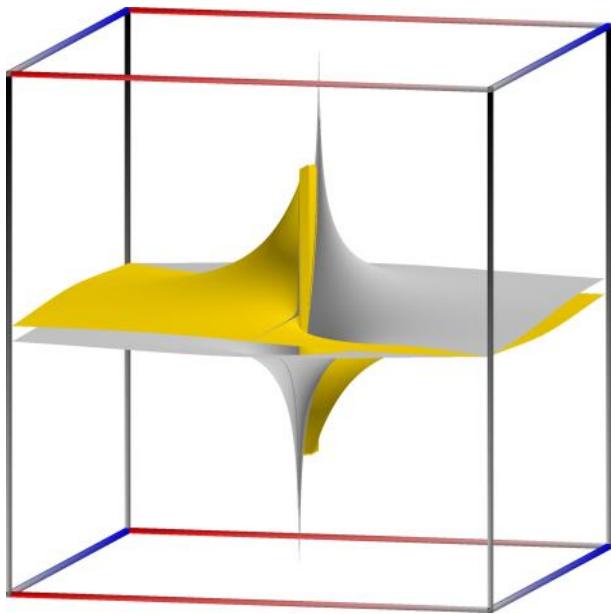
$z$ : A complex number.

The function  $\text{cplxACoth}(z)$  returns the inverse complex hyperbolic cotangent of  $z$ :

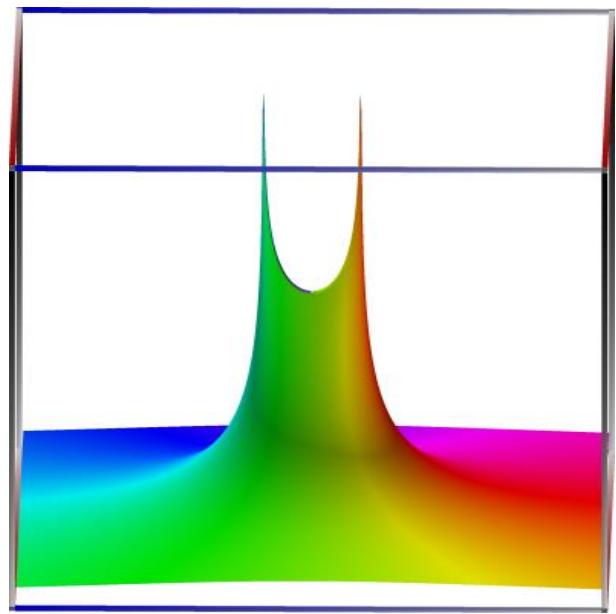
$$\text{arctanh}(z) = i \text{ arctan}(iz), \quad (5.7.4)$$

where  $\text{arctan}(z)$  is defined in section 5.6.5

Computes the inverse hyperbolic cotangent of  $x$ ,  $\coth^{-1}(x) = \tanh^{-1}(1/x)$



(a) Real ("silver") and imaginary ("gold") component,  $z_{\min} = -10$ . Camera angles are  $\theta = 135^\circ$  and  $\phi = -12^\circ$ .



(b) Magnitude and phase (color-coded),  $z_{\min} = 0$ . Camera angles are  $\theta = 35^\circ$  and  $\phi = -112^\circ$ .

**Figure 5.25:** Surface plots of  $z = \text{acoth}(x + iy)$ ,  $-3 \leq x \leq 3$  (blue axis),  $-2\pi \leq y \leq 2\pi$  (red axis),  $z_{\min} \leq z \leq 10$  (black axis).  $z$  values are truncated at  $\pm 10$ . There is a branch cut along the negative real axis. Orthographic camera. See section 2.3.3 for more information about charts for complex functions.

### 5.7.5 $\text{asech}(x)$

Computes the inverse hyperbolic secant of  $x$ ,  $\text{sech}^{-1}(x) = \cosh^{-1}(1/x)$

### 5.7.6 $\text{acsch}(x)$

Computes the inverse hyperbolic cosecant of  $x$ ,  $\text{csch}^{-1}(x) = \sinh^{-1}(1/x)$

# Chapter 6

## Linear Algebra

### 6.1 Norms

Sometimes you need to know how 'large' a matrix or vector is. Due to their multidimensional nature it is not possible to compare them, but there are several functions to map a matrix or a vector to a positive real number, the so called norms.

---

Function **norm(Y As mpNum[], Keywords As String)** As mpNumList

---

The function **norm** returns the entrywise  $p$ -norm of an iterable  $x$ , i.e. the vector norm.

#### Parameters:

$Y$ : An array of real numbers.

*Keywords*:  $p=2$ .

`norm(ctx, x, p=2)` Gives the entrywise  $p$ -norm of an iterable  $x$ , i.e. the vector norm

$$\left( \sum_k |x_k|^p \right)^{1/p}, \quad (6.1.1)$$

for any given  $1 \leq p \leq \infty$ .

Special cases:

If  $x$  is not iterable, this just returns `absmax(x)`.

$p=1$  gives the sum of absolute values.

$p=2$  is the standard Euclidean vector norm.

$p=\infty$  gives the magnitude of the largest element.

For  $x$  a matrix,  $p=2$  is the Frobenius norm. For operator matrix norms, use `mnorm()` instead.

You can use the string 'inf' as well as `float('inf')` or `mpf('inf')` to specify the infinity norm.

#### Examples

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> x = matrix([-10, 2, 100])
>>> norm(x, 1)
mpf('112.0')
>>> norm(x, 2)
mpf('100.5186549850325')
>>> norm(x, inf)
mpf('100.0')
```

---

---

**Function `mnorm(A As mpNum[], Keywords As String) As mpNumList`**


---

The function `mnorm` returns the matrix (operator)  $p$ -norm of  $A$ . Currently  $p=1$  and  $p=\infty$  are supported.

**Parameters:**

$A$ : An array of real numbers.

*Keywords*:  $p=1$ .

`mnorm(ctx, A, p=1)`

Gives the matrix (operator)  $p$ -norm of  $A$ . Currently  $p=1$  and  $p=\infty$  are supported:

$p=1$  gives the 1-norm (maximal column sum)

$p=\infty$  gives the  $\infty$ -norm (maximal row sum). You can use the string 'inf' as well as `float('inf')` or `mpf('inf')`

$p=2$  (not implemented) for a square matrix is the usual spectral matrix norm, i.e. the largest singular value.

$p='f'$  (or 'F', 'fro', 'Frobenius', 'frobenius') gives the Frobenius norm, which is the elementwise 2-norm. The Frobenius norm is an approximation of the spectral norm and satisfies

$$\frac{1}{\sqrt{\text{rank}(A)}} \|A\|_F \leq \|A\|_2 \leq \|A\|_F. \quad (6.1.2)$$

The Frobenius norm lacks some mathematical properties that might be expected of a norm.

For general elementwise  $p$ -norms, use `norm()` instead.

**Examples**

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = False
>>> A = matrix([[1, -1000], [100, 50]])
>>> mnorm(A, 1)
mpf('1050.0')
>>> mnorm(A, inf)
mpf('1001.0')
>>> mnorm(A, 'F')
mpf('1006.2310867787777')
```

---

## 6.2 Decompositions

---

Function **cholesky(A As mpNum[], Keywords As String)** As mpNum

---

The function **cholesky** returns the Cholesky decomposition of a symmetric positive-definite matrix  $A$ .

### Parameters:

**A:** A symmetric matrix.

**Keywords:** tol=None.

**cholesky(ctx, A, tol=None)**

Cholesky decomposition of a symmetric positive-definite matrix  $A$ . Returns a lower triangular matrix  $L$  such that  $A = L \times L^T$ . More generally, for a complex Hermitian positive-definite matrix, a Cholesky decomposition satisfying  $A = L \times L^H$  is returned.

The Cholesky decomposition can be used to solve linear equation systems twice as efficiently as LU decomposition, or to test whether  $A$  is positive-definite.

The optional parameter tol determines the tolerance for verifying positive-definiteness.

### Examples

Cholesky decomposition of a positive-definite symmetric matrix:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> A = eye(3) + hilbert(3)
>>> nprint(A)
[ 2.0 0.5 0.333333]
[ 0.5 1.33333 0.25]
[0.333333 0.25 1.2]
>>> L = cholesky(A)
>>> nprint(L)
[ 1.41421 0.0 0.0]
[0.353553 1.09924 0.0]
[0.235702 0.15162 1.05899]
>>> chop(A - L*L.T)
[0.0 0.0 0.0]
[0.0 0.0 0.0]
[0.0 0.0 0.0]
```

---

Cholesky decomposition of a Hermitian matrix:

---

```
>>> A = eye(3) + matrix([[0,0.25j,-0.5j],[-0.25j,0,0],[0.5j,0,0]])
>>> L = cholesky(A)
>>> nprint(L)
[ 1.0 0.0 0.0]
[(0.0 - 0.25j) (0.968246 + 0.0j) 0.0]
[ (0.0 + 0.5j) (0.129099 + 0.0j) (0.856349 + 0.0j)]
>>> chop(A - L*L.H)
[0.0 0.0 0.0]
[0.0 0.0 0.0]
[0.0 0.0 0.0]
```

---

Attempted Cholesky decomposition of a matrix that is not positive definite:

```
>>> A = -eye(3) + hilbert(3)
>>> L = cholesky(A)
Traceback (most recent call last):
...
ValueError: matrix is not positive-definite
```

## 6.3 Linear Equations

---

Function **lu\_solve(A As mpNum[], b As mpNum[], Keywords As String)** As mpNum

---

The function lu\_solve returns solves a linear equation system using a LU decomposition.

**Parameters:**

*A*: A symmetric matrix.

*b*: A symmetric matrix.

*Keywords*: tol=None.

You can for example solve the linear equation system using a LU decomposition:

---

```
x + 2*y = -10
3*x + 4*y = 10
```

---

using lu\_solve:

---

```
>>> A = matrix([[1, 2], [3, 4]])
>>> b = matrix([-10, 10])
>>> x = lu_solve(A, b)
>>> x
matrix(
[['30.0'],
 ['-20.0']])
```

---



---

Function **residual(A As mpNum[], b As mpNum[], x As mpNum[], Keywords As String)** As mpNum

---

The function residual returns the residual  $\|Ax - b\|$ .

**Parameters:**

*A*: A square matrix.

*b*: A vector.

*x*: A vector.

*Keywords*: tol=None.

Calculates the residual  $\|Ax - b\|$ :

---

```
>>> residual(A, x, b)
matrix(
[['3.46944695195361e-18'],
 ['3.46944695195361e-18']])
>>> str(eps)
'2.22044604925031e-16'
```

---

As you can see, the solution is quite accurate. The error is caused by the inaccuracy of the internal floating point arithmetic. Though, it is even smaller than the current machine epsilon, which basically means you can trust the result.

If you need more speed, use NumPy, or use fp instead mp matrices and methods:

---

```
>>> A = fp.matrix([[1, 2], [3, 4]])
>>> b = fp.matrix([-10, 10])
>>> fp.lu_solve(A, b)
matrix()
```

---

```
[[30.0],  
[-20.0]])
```

---

lu\_solve accepts overdetermined systems. It is usually not possible to solve such systems, so the residual is minimized instead. Internally this is done using Cholesky decomposition to compute a least squares approximation. This means that that lu\_solve will square the errors. If you cannot afford this, use qr\_solve instead. It is twice as slow but more accurate, and it calculates the residual automatically.

## 6.4 Matrix Factorization

---

### Function **lu**(*A* As *mpNum*[], *Keywords* As *String*) As *mpNum*

---

The function **lu** returns an explicit LU factorization of a matrix, returning P, L, U

**Parameters:**

*A*: A square matrix.

*Keywords*: tol=None.

The function **lu** computes an explicit LU factorization of a matrix:

---

```
>>> P, L, U = lu(matrix([[0,2,3],[4,5,6],[7,8,9]]))
>>> print P
[0.0 0.0 1.0]
[1.0 0.0 0.0]
[0.0 1.0 0.0]
>>> print L
[ 1.0 0.0 0.0]
[ 0.0 1.0 0.0]
[0.571428571428571 0.214285714285714 1.0]
>>> print U
[7.0 8.0 9.0]
[0.0 2.0 3.0]
[0.0 0.0 0.214285714285714]
>>> print P.T*L*U
[0.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
```

---



---

### Function **qr**(*A* As *mpNum*[], *Keywords* As *String*) As *mpNum*

---

The function **qr** returns an explicit QR factorization of a matrix, returning Q, R

**Parameters:**

*A*: A square matrix.

*Keywords*: tol=None.

Examples:

---

```
>>> A = matrix([[1, 2], [3, 4], [1, 1]])
>>> Q, R = qr(A)
>>> print Q
[-0.301511344577764 0.861640436855329 0.408248290463863]
[-0.904534033733291 -0.123091490979333 -0.408248290463863]
[-0.301511344577764 -0.492365963917331 0.816496580927726]
>>> print R
[-3.3166247903554 -4.52267016866645]
[ 0.0 0.738548945875996]
[ 0.0 0.0]
>>> print Q * R
[1.0 2.0]
[3.0 4.0]
[1.0 1.0]
```

---

```
>>> print chop(Q.T * Q)
[1.0 0.0 0.0]
[0.0 1.0 0.0]
[0.0 0.0 1.0]
```

---

# **Part III**

## **Special Functions**

# Chapter 7

# Factorials and gamma functions

Factorials and factorial-like sums and products are basic tools of combinatorics and number theory. Much like the exponential function is fundamental to differential equations and analysis in general, the factorial function (and its extension to complex numbers, the gamma function) is fundamental to difference equations and functional equations.

A large selection of factorial-like functions is implemented in mpFormulaPy. All functions support complex arguments, and arguments may be arbitrarily large. Results are numerical approximations, so to compute exact values a high enough precision must be set manually:

The gamma and polygamma functions are closely related to Zeta functions, L-series and polylogarithms. See also q-functions for q-analogs of factorial-like functions.

## 7.1 Factorials

### 7.1.1 Factorial

Function **Factorial(z As mpNum)** As mpNum

The function `Factorial` returns the factorial,  $x!$ .

## Parameter:

*z*: A real or complex number.

Function `fac(z As mpNum)` As mpNum

The function `fac` returns the factorial  $x!$

### Parameter:

$z$ : A real or complex number.

Computes the factorial,  $x!$ . For integers  $n > 0$ , we have  $n! = 1 \cdot 2 \cdots (n-1) \cdot n$  and more generally the factorial is defined for real or complex  $x$  by  $x! = \Gamma(x+1)$ .

Examples

Basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for k in range(6):
...     print("%s %s" % (k, fac(k)))
...
0 1.0
1 1.0
2 2.0
3 6.0
4 24.0
5 120.0
>>> fac(inf)
+inf
>>> fac(0.5), sqrt(pi)/2
(0.886226925452758, 0.886226925452758)
```

---

`fac()` supports evaluation for astronomically large values:

---

```
>>> fac(10**30)
6.22311232304258e+29565705518096748172348871081098
```

---

## 7.1.2 Double factorial

---

Function **fac2( $z$  As  $mpNum$ ) As  $mpNum$**

---

The function `fac2` returns the double factorial  $x!!$ .

**Parameter:**

$z$ : A real or complex number.

Computes the double factorial  $x!!$ , defined for integers  $x > 0$  by

$$x!! = \begin{cases} 1 \cdot 3 \cdots (x-2) \cdot x & \text{for } x \text{ odd} \\ 2 \cdot 4 \cdots (x-2) \cdot x & \text{for } x \text{ even} \end{cases} \quad (7.1.1)$$

and more generally by [1]

$$x!! = 2^{x/2} \left(\frac{\pi}{2}\right)^{(\cos(\pi x)-1)/4} \Gamma\left(\frac{x}{2} + 1\right) \quad (7.1.2)$$

Examples

The integer sequence of double factorials begins:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint([fac2(n) for n in range(10)])
[1.0, 1.0, 2.0, 3.0, 8.0, 15.0, 48.0, 105.0, 384.0, 945.0]
```

---

With the exception of the poles at negative even integers, `fac2()` supports evaluation for arbitrary complex arguments. The recurrence formula is valid generally:

---

```
>>> fac2(pi+2j)
(-1.3697207890154e-12 + 3.93665300979176e-12j)
>>> (pi+2j)*fac2(pi-2+2j)
(-1.3697207890154e-12 + 3.93665300979176e-12j)
```

---

## 7.2 Binomial coefficient

---

Function **binomial(*n* As mpNum, *k* As mpNum)** As mpNum

---

The function `binomial` returns the binomial coefficient.

**Parameters:**

*n*: A real or complex number.

*k*: A real or complex number.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \quad (7.2.1)$$

The binomial coefficient gives the number of ways that *k* items can be chosen from a set of *n* items. More generally, the binomial coefficient is a well-defined function of arbitrary real or complex *n* and *k*, via the gamma function.

Examples

Generate Pascal's triangle:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint([binomial(n,k) for k in range(n+1)])
...
[1.0]
[1.0, 1.0]
[1.0, 2.0, 1.0]
[1.0, 3.0, 3.0, 1.0]
[1.0, 4.0, 6.0, 4.0, 1.0]
```

---

`binomial()` supports large arguments:

---

```
>>> binomial(10**20, 10**20-5)
8.3333333333333e+97
>>> binomial(10**20, 10**10)
2.60784095465201e+104342944813
```

---

## 7.3 Pochhammer symbol, Rising and falling factorials

### 7.3.1 Relative Pochhammer symbol

---

Function **RelativePochhammerMpMath(*a* As mpNum, *x* As mpNum) As mpNum**

**NOT YET IMPLEMENTED**

---

The function RelativePochhammerMpMath returns the relative Pochhammer symbol.

**Parameters:**

*a*: An integer.

*x*: An integer.

The relative Pochhammer symbol defined as

$$\text{poch1}(a, x) = \frac{(a)_x - 1}{x}, \quad (7.3.1)$$

accurate even for small *x*. If  $|x|$  is small, cancellation errors are avoided by using an expansion by Fields and Luke with generalized Bernoulli polynomials. For  $x = 0$  the value  $\psi(a)$  is returned, otherwise the result is calculated from the definition.

In mathematics, the Pochhammer symbol introduced by Leo August Pochhammer is the notation  $(x)_n$ , where *n* is a non-negative integer. Depending on the context the Pochhammer symbol may represent either the rising factorial or the falling factorial as defined below. Care needs to be taken to check which interpretation is being used in any particular article.

### 7.3.2 Rising factorial

---

Function **rf(*x* As mpNum, *n* As mpNum) As mpNum**

The function rf returns the rising factorial.

**Parameters:**

*x*: A real or complex number.

*n*: A real or complex number.

Computes the rising factorial,

$$x^{(n)} = x(x + 1) \cdots (x + n - 1) = \frac{\Gamma(x + n)}{\Gamma(x)} \quad (7.3.2)$$

where the rightmost expression is valid for nonintegral *n*.

**Examples**

For integral *n*, the rising factorial is a polynomial:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(taylor(lambda x: rf(x,n), 0, n))
...
[1.0]
[0.0, 1.0]
[0.0, 1.0, 1.0]
[0.0, 2.0, 3.0, 1.0]
```

---

[0.0, 6.0, 11.0, 6.0, 1.0]

---

Evaluation is supported for arbitrary arguments:

---

```
>>> rf(2+3j, 5.5)
(-7202.03920483347 - 3777.58810701527j)
```

---

### 7.3.3 Falling factorial

---

Function **ff(x As mpNum, n As mpNum) As mpNum**

---

The function **ff** returns the falling factorial.

**Parameters:**

*x*: A real or complex number.

*n*: A real or complex number.

The falling factorial is defined as,

$$x_{(n)} = x(x - 1) \cdots (x - n + 1) = \frac{\Gamma(x + 1)}{\Gamma(x - n + 1)} \quad (7.3.3)$$

where the rightmost expression is valid for nonintegral *n*.

Examples

For integral *n*, the falling factorial is a polynomial:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(5):
...     nprint(taylor(lambda x: ff(x,n), 0, n))
...
[1.0]
[0.0, 1.0]
[0.0, -1.0, 1.0]
[0.0, 2.0, -3.0, 1.0]
[0.0, -6.0, 11.0, -6.0, 1.0]
```

---

Evaluation is supported for arbitrary arguments:

---

```
>>> ff(2+3j, 5.5)
(-720.41085888203 + 316.101124983878j)
```

---

## 7.4 Super- and hyperfactorials

### 7.4.1 Superfactorial

---

Function **superfac(z As mpNum)** As mpNum

---

The function `superfac` returns the superfactorial.

**Parameter:**

`z`: A real or complex number.

The superfactorial is defined as the product of consecutive factorials:

$$sf(n) = \prod_{k=1}^n k! \quad (7.4.1)$$

For general complex  $z$ ,  $sf(n)$  is defined in terms of the Barnes G-function (see `barnesg()`).

Examples

The first few superfactorials are (OEIS A000178):

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(10):
...     print("%s %s" % (n, superfac(n)))
...
0 1.0
1 1.0
2 2.0
3 12.0
4 288.0
5 34560.0
6 24883200.0
7 125411328000.0
8 5.05658474496e+15
9 1.83493347225108e+21
```

---

Superfactorials grow very rapidly:

---

```
>>> superfac(1000)
3.24570818422368e+1177245
>>> superfac(10**10)
2.61398543581249e+467427913956904067453
```

---

Evaluation is supported for arbitrary arguments:

---

```
>>> mp.dps = 25
>>> superfac(pi)
17.20051550121297985285333
>>> superfac(2+3j)
(-0.005915485633199789627466468 + 0.008156449464604044948738263j)
>>> diff(superfac, 1)
0.2645072034016070205673056
```

---

## 7.4.2 Hyperfactorial

---

### Function **hyperfac(z As mpNum) As mpNum**

---

The function `hyperfac` returns the hyperfactorial.

**Parameter:**

*z*: A real or complex number.

The hyperfactorial is defined for integers as the product

$$H(n) = \prod_{k=1}^n k^k \quad (7.4.2)$$

The hyperfactorial satisfies the recurrence formula  $H(z) = z^z H(z - 1)$ . It can be defined more generally in terms of the Barnes G-function (see `barnesg()`) and the gamma function by the formula.

$$H(z) = \frac{\Gamma(z + 1)^z}{G(z)}. \quad (7.4.3)$$

The extension to complex numbers can also be done via the integral representation

$$H(z) = (2\pi)^{-z/2} \exp \left[ \binom{z+1}{2} + \int_0^z \log(t!) dt \right]. \quad (7.4.4)$$

Examples

The rapidly-growing sequence of hyperfactorials begins (OEIS A002109):

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(10):
...     print("%s %s" % (n, hyperfac(n)))
...
0 1.0
1 1.0
2 4.0
3 108.0
4 27648.0
5 86400000.0
6 4031078400000.0
7 3.3197663987712e+18
8 5.56964379417266e+25
9 2.15779412229419e+34
```

---

Some even larger hyperfactorials are:

---

```
>>> hyperfac(1000)
5.46458120882585e+1392926
>>> hyperfac(10**10)
4.60408207642219e+489142638002418704309
```

---

Evaluation is supported for arbitrary arguments:

---

```
>>> hyperfac(0.5)
0.880449235173423
```

---

```

>>> diff(hyperfac, 1)
0.581061466795327
>>> hyperfac(pi)
205.211134637462
>>> hyperfac(-10+1j)
(3.01144471378225e+46 - 2.45285242480185e+46j)

```

---

### 7.4.3 Barnes G-function

---

Function **barnesg(z As mpNum) As mpNum**

---

The function **barnesg** returns the Barnes G-function.

**Parameter:**

*z*: A real or complex number.

The Barnes G-function generalizes the superfactorial (`superfac()`) and by extension also the hyperfactorial (`hyperfac()`) to the complex numbers in an analogous way to how the gamma function generalizes the ordinary factorial.

The Barnes G-function may be defined in terms of a Weierstrass product:

$$G(z+1) = (2\pi)^{z/2} e^{[z(z+1)+\gamma z^2]/2} \prod_{n=1}^{\infty} \left[ \left(1 + \frac{z}{n}\right)^n e^{-z+z^2/(2n)} \right] \quad (7.4.5)$$

For positive integers *n*, we have have relation to superfactorials  $G(n) = sf(n-2) = 0! \cdot 1! \cdots (n-2)!$ .  
 REF: Whittaker & Watson, A Course of Modern Analysis, Cambridge University Press, 4th edition (1927), p.264s

Examples

Some elementary values and limits of the Barnes G-function:

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> barnesg(1), barnesg(2), barnesg(3)
(1.0, 1.0, 1.0)
>>> barnesg(4)
2.0
>>> barnesg(5)
12.0
>>> barnesg(6)
288.0
>>> barnesg(7)
34560.0
>>> barnesg(8)
24883200.0
>>> barnesg(inf)
+inf
>>> barnesg(0), barnesg(-1), barnesg(-2)
(0.0, 0.0, 0.0)

```

---

## 7.5 Gamma functions

### 7.5.1 Gamma function

---

#### Function `gamma(z As mpNum) As mpNum`

---

The function `gamma` returns the gamma function,  $\Gamma(x)$ .

**Parameter:**

`z`: A real or complex number.

The gamma function is a shifted version of the ordinary factorial, satisfying  $\Gamma(n) = (n - 1)!$  for integers  $n > 0$ . More generally, it is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \quad (7.5.1)$$

for any real or complex  $x$  with  $\Re(x) > 0$  and for  $\Re(x) < 0$  by analytic continuation.

Examples

Basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for k in range(1, 6):
...     print("%s %s" % (k, gamma(k)))
...
1 1.0
2 1.0
3 2.0
4 6.0
5 24.0
>>> gamma(inf)
+inf
>>> gamma(0)
Traceback (most recent call last):
...
ValueError: gamma function pole
```

---

`gamma()` supports arbitrary-precision evaluation and complex arguments:

---

```
>>> mp.dps = 50
>>> gamma(sqrt(3))
0.91510229697308632046045539308226554038315280564184
>>> mp.dps = 25
>>> gamma(2j)
(0.009902440080927490985955066 - 0.07595200133501806872408048j)
```

---

Arguments can also be large. Note that the gamma function grows very quickly:

---

```
>>> mp.dps = 15
>>> gamma(10**20)
1.9328495143101e+1956570551809674817225
```

---

## 7.5.2 Reciprocal of the gamma function

---

### Function `rgamma(z As mpNum) As mpNum`

---

The function `rgamma` returns the reciprocal of the gamma function,  $1/\Gamma(z)$ .

**Parameter:**

`z`: A real or complex number.

This function evaluates to zero at the poles of the gamma function,  $z = 0, -1, -2, \dots$

Examples

Basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> rgamma(1)
1.0
>>> rgamma(4)
0.16666666666666666666666666666667
>>> rgamma(0); rgamma(-1)
0.0
0.0
>>> rgamma(1000)
2.485168143266784862783596e-2565
>>> rgamma(inf)
0.0
```

---

## 7.5.3 The product / quotient of gamma functions

---

### Function `gammaprod(a As mpNum, b As mpNum) As mpNum`

---

The function `gammaprod` returns the product / quotient of gamma functions.

**Parameters:**

`a`: A real or complex iterables.

`b`: A real or complex iterables.

Given iterables `a` and `b`, `gammaprod(a, b)` computes the product / quotient of gamma functions:

$$\frac{\Gamma(a_0)\Gamma(a_1)\cdots\Gamma(a_p)}{\Gamma(b_0)\Gamma(b_1)\cdots\Gamma(b_p)} \quad (7.5.2)$$

Unlike direct calls to `gamma()`, `gammaprod()` considers the entire product as a limit and evaluates this limit properly if any of the numerator or denominator arguments are nonpositive integers such that poles of the gamma function are encountered. That is, `gammaprod()` evaluates

$$\lim_{\epsilon \rightarrow 0} \frac{\Gamma(a_0 + \epsilon)\Gamma(a_1 + \epsilon)\cdots\Gamma(a_p + \epsilon)}{\Gamma(b_0 + \epsilon)\Gamma(b_1 + \epsilon)\cdots\Gamma(b_p + \epsilon)} \quad (7.5.3)$$

In particular:

If there are equally many poles in the numerator and the denominator, the limit is a rational number times the remaining, regular part of the product.

If there are more poles in the numerator, `gammaprod()` returns `+inf`.

If there are more poles in the denominator, gammaprod() returns 0.

Examples

The reciprocal gamma function  $1/\Gamma(x)$  evaluated at 0:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15
>>> gammaprod([], [0])
0.0
```

---

A limit:

---

```
>>> gammaprod([-4], [-3])
-0.25
>>> limit(lambda x: gamma(x-1)/gamma(x), -3, direction=1)
-0.25
>>> limit(lambda x: gamma(x-1)/gamma(x), -3, direction=-1)
-0.25
```

---

## 7.5.4 The log-gamma function

---

Function **loggamma(z As mpNum)** As mpNum

---

The function loggamma returns the principal branch of the log-gamma function,  $\ln \Gamma(z)$ .

**Parameter:**

*z*: A real or complex number.

Unlike  $\ln(\Gamma(z))$ , which has infinitely many complex branch cuts, the principal log-gamma function only has a single branch cut along the negative half-axis. The principal branch continuously matches the asymptotic Stirling expansion

$$\ln \Gamma(z) \approx \frac{\ln(2\pi)}{2} + \left(z - \frac{1}{2}\right) \ln(z) - z + O(z^{-1}) \quad (7.5.4)$$

The real parts of both functions agree, but their imaginary parts generally differ by  $2n\pi$  for some  $n \in \mathbb{Z}$ . They coincide for  $z \in \mathbb{R}, z > 0$ .

Computationally, it is advantageous to use loggamma() instead of gamma() for extremely large arguments.

Examples

Comparing with :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> loggamma('13.2'); log(gamma('13.2'))
20.49400419456603678498394
20.49400419456603678498394
>>> loggamma(3+4j)
(-1.756626784603784110530604 + 4.742664438034657928194889j)
>>> log(gamma(3+4j))
(-1.756626784603784110530604 - 1.540520869144928548730397j)
>>> log(gamma(3+4j)) + 2*pi*j
(-1.756626784603784110530604 + 4.742664438034657928194889j)
```

---

Note the imaginary parts for negative arguments:

---

```
>>> loggamma(-0.5); loggamma(-1.5); loggamma(-2.5)
(1.265512123484645396488946 - 3.141592653589793238462643j)
(0.8600470153764810145109327 - 6.283185307179586476925287j)
(-0.05624371649767405067259453 - 9.42477796076937971538793j)
```

---

### 7.5.5 Generalized incomplete gamma function

---

Function **gammainc**(*z* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function **gammainc** returns the incomplete gamma function with integration limits  $[a, b]$ .

**Parameters:**

*z*: A real or complex number.

*a*: A real or complex number (default = 0).

*b*: A real or complex number (default = inf).

**Keywords:** regularized=False.

The (generalized) incomplete gamma function with integration limits  $[a, b]$  is defined as:

$$\Gamma(z, a, b) = \int_a^b t^{z-1} e^{-t} dt \quad (7.5.5)$$

The generalized incomplete gamma function reduces to the following special cases when one or both endpoints are fixed:

$\Gamma(z, 0, \infty)$  is the standard ('complete') gamma function,  $\Gamma(z)$ , available directly as the *mpFormulaPy* function **gamma()**.

$\Gamma(z, a, \infty)$  is the 'upper' incomplete gamma function,  $\Gamma(z, a)$ .

$\Gamma(z, 0, b)$  is the 'lower' incomplete gamma function,  $\gamma(z, a)$ .

Of course, we have  $\Gamma(z, a, \infty) + \Gamma(z, 0, b) = \Gamma(z)$  for all *z* and *x*.

Note however that some authors reverse the order of the arguments when defining the lower and upper incomplete gamma function, so one should be careful to get the correct definition.

If also given the keyword argument **regularized=True**, **gammainc()** computes the 'regularized' incomplete gamma function

$$P(z, a, b) = \frac{\Gamma(z, a, b)}{\Gamma(z)}. \quad (7.5.6)$$

Examples

We can compare with numerical quadrature to verify that **gammainc()** computes the integral in the definition:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> gammainc(2+3j, 4, 10)
(0.00977212668627705160602312 - 0.0770637306312989892451977j)
>>> quad(lambda t: t**(2+3j-1) * exp(-t), [4, 10])
(0.00977212668627705160602312 - 0.0770637306312989892451977j)
```

---

Evaluation for arbitrarily large arguments:

---

```
>>> gammainc(10, 100)
4.083660630910611272288592e-26
>>> gammainc(10, 10000000000000000)
5.290402449901174752972486e-4342944819032375
>>> gammainc(3+4j, 1000000+100000j)
(-1.257913707524362408877881e-434284 + 2.556691003883483531962095e-434284j)
```

---

## 7.5.6 Derivative of the normalised incomplete gamma function

---

Function **GammaPDerivativeMpMath**(*a* As mpNum, *x* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **GammaPDerivativeMpMath** returns the partial derivative with respect to *x* of the incomplete gamma function  $P(a, x)$ .

**Parameters:**

*a*: A real number.

*x*: A real number.

The partial derivative with respect to *x* of the incomplete gamma function  $P(a, x)$  is defined as:

$$\frac{\partial}{\partial x} P(a, x) = \frac{e^{-x} x^{a-1}}{\Gamma(a)}. \quad (7.5.7)$$

## 7.5.7 Normalised incomplete gamma functions

---

Boost references are [Temme \(1979\)](#) and [Temme \(1994\)](#)

---

Function **GammaPMpMath**(*a* As mpNum, *x* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **GammaPMpMath** returns the normalised incomplete gamma function  $P(a, x)$ .

**Parameters:**

*a*: A real number.

*x*: A real number.

The normalised incomplete gamma function  $P(a, x)$  is defined as

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt \quad (7.5.8)$$

for  $a \geq 0$  and  $x \geq 0$ .

---

Function **GammaQMpMath**(*a* As mpNum, *x* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **GammaQMpMath** returns the normalised incomplete gamma function  $Q(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

The normalised incomplete gamma function  $Q(a, x)$  is defined as

$$Q(a, x) = \frac{1}{\Gamma(a)} \int_x^{\infty} t^{a-1} e^{-t} dt \quad (7.5.9)$$

for  $a \geq 0$  and  $x \geq 0$ .

### 7.5.8 Non-Normalised incomplete gamma functions

---

Function **NonNormalisedGammaPMpMath**(*a* As mpNum, *x* As mpNum) As mpNum

NOT YET IMPLEMENTED

---

The function NonNormalisedGammaPMpMath returns the non-normalised incomplete gamma function  $\Gamma(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

The non-normalised incomplete gamma function  $\Gamma(a, x)$  is defined as

$$\Gamma(a, x) = \int_0^x t^{a-1} e^{-t} dt \quad (7.5.10)$$

for  $a \geq 0$  and  $x \geq 0$ .

---

Function **NonNormalisedGammaQMpMath**(*a* As mpNum, *x* As mpNum) As mpNum

NOT YET IMPLEMENTED

---

The function NonNormalisedGammaQMpMath returns the non-normalised incomplete gamma function  $\gamma(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

The non-normalised incomplete gamma function  $\gamma(a, x)$  is defined as

$$\gamma(a, x) = \int_x^{\infty} t^{a-1} e^{-t} dt \quad (7.5.11)$$

for  $a \geq 0$  and  $x \geq 0$ .

Note: in Boost, the functions are referred to as TgammaLower and TgammaUpper.

### 7.5.9 Tricomi's entire incomplete gamma function

---

Function **TricomiGammaMpMath**(*a* As mpNum, *x* As mpNum) As mpNum

NOT YET IMPLEMENTED

---

The function **TricomiGammaMpMath** returns Tricomi's entire incomplete gamma function  $\gamma^*(a, x)$ .

**Parameters:**

*a*: A real number.

*x*: A real number.

This routine returns Tricomi's incomplete gamma function  $\gamma^*$ , defined as

$$\gamma^*(a, x) = e^{-x} \frac{M(1, a + 1, x)}{\Gamma(a + 1)} \quad (7.5.12)$$

Special cases are  $\gamma^*(0, x) = 1$ ,  $\gamma^*(a, 0) = 1/\Gamma(a + 1)$ , and  $\gamma^*(-n, x) = x^n$ , if  $-n$  is a negative integer. Otherwise there are the following relations to the other incomplete functions:

$$\gamma^*(a, x) = \frac{x^{-a}}{\Gamma(a)} \gamma(a, x) = x^{-a} P(a, x). \quad (7.5.13)$$

### 7.5.10 Inverse normalised incomplete gamma functions

---

Function **GammaPinvMpMath(a As mpNum, p As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function **GammaPinvMpMath** returns the inverse of the normalised incomplete gamma function  $P(a, x)$ .

**Parameters:**

*a*: A real number.

*p*: A real number.

This function returns the inverse normalised incomplete gamma function, i.e. it calculates *x* with  $P(a, x) = p$ . The input parameters are  $a > 0$ ,  $p \geq 0$ , and  $p + q = 1$ .

---

Function **GammaQinvMpMath(a As mpNum, q As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function **GammaQinvMpMath** returns the inverse of the normalised incomplete gamma function  $Q(a, x)$ .

**Parameters:**

*a*: A real number.

*q*: A real number.

## 7.6 Polygamma functions and harmonic numbers

### 7.6.1 Polygamma function

---

#### Function **polygamma(*m* As mpNum, *z* As mpNum)** As mpNum

---

The function **polygamma** returns the polygamma function of order *m* of *z*,  $\psi^{(m)}(z)$ .

**Parameters:**

*m*: A real or complex number.

*z*: A real or complex number.

Special cases are known as the digamma function ( $\psi^{(0)}(z)$ ), the trigamma function ( $\psi^{(1)}(z)$ ), etc. The polygamma functions are defined as the logarithmic derivatives of the gamma function:

$$\psi^{(m)}(z) = \left( \frac{d}{dz} \right)^{m+1} \log \Gamma(z). \quad (7.6.1)$$

In particular,  $\psi^{(0)}(z) = \Gamma'(z)/\Gamma(z)$ . In the present implementation of **psi()**, the order *m* must be a nonnegative integer, while the argument *z* may be an arbitrary complex number (with exception for the polygamma function's poles at *z* = 0, -1, -2, ...).

Examples

For various rational arguments, the polygamma function reduces to a combination of standard mathematical constants:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> polygamma(0, 1), -euler
(-0.5772156649015328606065121, -0.5772156649015328606065121)
>>> polygamma(1, '1/4'), pi**2+8*catalan
(17.19732915450711073927132, 17.19732915450711073927132)
>>> polygamma(2, '1/2'), -14*apery
(-16.82879664423431999559633, -16.82879664423431999559633)
```

---

Evaluation for a complex argument:

---

```
>>> polygamma(2, -1-2j)
(0.03902435405364952654838445 + 0.1574325240413029954685366j)
```

---

Evaluation is supported for large orders and/or large arguments :

---

```
>>> psi(3, 10**100)
2.0e-300
>>> psi(250, 10**30+10**20*j)
(-1.293142504363642687204865e-7010 + 3.232856260909107391513108e-7018j)
```

---



---

#### Function **psi(*m* As mpNum, *z* As mpNum)** As mpNum

---

The function **psi** returns the polygamma function of order *m* of *z*,  $\psi^{(m)}(z)$ .

**Parameters:**

*m*: A real or complex number.

*z*: A real or complex number.

A shortcut for `polygamma(m,z)`.

## 7.6.2 Digamma function

---

Function **digamma(z As mpNum) As mpNum**

---

The function `digamma` returns the digamma function.

**Parameter:**

*z*: A real or complex number.

A shortcut for `psi(0,z)`.

## 7.6.3 Harmonic numbers

---

Function **harmonic(n As mpNum) As mpNum**

---

The function `harmonic` returns a floating-point approximation of the *n*-th harmonic number  $H(n)$ .

**Parameter:**

*n*: An real or complex number.

If *n* is an integer, `harmonic(n)` gives a floating-point approximation of the *n*-th harmonic number  $H(n)$ , defined as

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \quad (7.6.2)$$

The first few harmonic numbers are:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(8):
...     print("%s %s" % (n, harmonic(n)))
...
0 0.0
1 1.0
2 1.5
3 1.8333333333333
4 2.0833333333333
5 2.2833333333333
6 2.45
7 2.59285714285714
```

---

`harmonic()` supports arbitrary precision evaluation:

---

```
>>> mp.dps = 50
>>> harmonic(11)
3.0198773448773448773448773448773448773448773448773
>>> harmonic(pi)
1.8727388590273302654363491032336134987519132374152
```

---

## 7.7 Beta Functions

### 7.7.1 Beta function B(a, b)

#### 7.7.2 Beta function

---

Function **beta(x As mpNum, y As mpNum) As mpNum**

---

The function `beta` returns the beta function,  $B(x, y) = \Gamma(x)\Gamma(y)/\Gamma(x + y)$ .

**Parameters:**

*x*: A real or complex number.

*y*: A real or complex number.

Computes the beta function,  $B(x, y) = \Gamma(x)\Gamma(y)/\Gamma(x + y)$ . The beta function is also commonly defined by the integral representation

$$B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1} dt \quad (7.7.1)$$

Examples

`beta()` supports complex numbers and arbitrary precision evaluation:

---

```
>>> beta(1, 2+j)
(0.4 - 0.2j)
>>> mp.dps = 25
>>> beta(j, 0.5)
(1.079424249270925780135675 - 1.410032405664160838288752j)
>>> mp.dps = 50
>>> beta(pi, e)
0.037890298781212201348153837138927165984170287886464
```

---

### 7.7.3 Logarithm of B(a, b)

---

Function **LnBetaMpMath(a As mpNum, b As mpNum) As mpNum**

---

**NOT YET IMPLEMENTED**

---

The function `LnBetaMpMath` returns the logarithm of the beta function  $\ln B(a, b)$  with  $a, b \neq 0, -1, -2, \dots$

**Parameters:**

*a*: A real number.

*b*: A real number.

### 7.7.4 Generalized incomplete beta function

---

Function **betainc(a As mpNum, b As mpNum, x1 As mpNum, x2 As mpNum, *Keywords* As String) As mpNum**

---

The function `betainc` returns the generalized incomplete beta function.

**Parameters:**

*a*: A real or complex number.

*b*: A real or complex number.

*x1*: A real or complex number (default = 0).

*x2*: A real or complex number (default = 1).

*Keywords*: regularized=False.

The generalized incomplete beta function is defined as,

$$I_{x_1}^{x_2}(a, b) = \int_{x_1}^{x_2} t^{a-1}(1-t)^{b-1} dt \quad (7.7.2)$$

When  $x_1 = 0, x_2 = 1$ , this reduces to the ordinary (complete) beta function  $B(a, b)$ ; see beta().

With the keyword argument regularized=True, betainc() computes the regularized incomplete beta function  $I_{x_1}^{x_2}(a, b)/B(a, b)$ . This is the cumulative distribution of the beta distribution with parameters *a*, *b*.

Examples

Verifying that betainc() computes the integral in the definition:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> x,y,a,b = 3, 4, 0, 6
>>> betainc(x, y, a, b)
-4010.4
>>> quad(lambda t: t**(x-1) * (1-t)**(y-1), [a, b])
-4010.4
```

---

The arguments may be arbitrary complex numbers:

---

```
>>> betainc(0.75, 1-4j, 0, 2+3j)
(0.2241657956955709603655887 + 0.3619619242700451992411724j)
```

---

With regularization:

---

```
>>> betainc(1, 2, 0, 0.25, regularized=True)
0.4375
>>> betainc(pi, e, 0, 1, regularized=True) # Complete
1.0
```

---

## 7.7.5 Non-Normalised incomplete beta functions

The algorithm is implemented as in [DiDonato & Morris \(1986\)](#)

---

Function **IBetaNonNormalizedMpMath(*a* As mpNum, *b* As mpNum, *x* As mpNum) As mpNum**  
**NOT YET IMPLEMENTED**

---

The function IBetaNonNormalizedMpMath returns the non-normalised incomplete beta function.

**Parameters:**

*a*: A real number.

*b*: A real number.

*x*: A real number.

This function returns the non-normalised incomplete beta function  $B_x(a, b)$  for  $a > 0$ ,  $b > 0$ , and  $0 \leq x \leq 1$ :

$$B_x(a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt. \quad (7.7.3)$$

There are some special cases

$$B_0(a, b) = 0, \quad B_1(a, b) = B(a, b), \quad B_x(a, 1) = \frac{x^a}{a}, \quad B_x(1, b) = \frac{1 - (1-x)^b}{b}, \quad (7.7.4)$$

and the relation  $B_{1-x}(a, b) = B(a, b) - B_x(b, a)$ , which is used if  $x > a/(a+b)$ .

When  $a \leq 0$  or  $b \leq 0$ , the Gauss hypergeometric function  ${}_2F_1(\cdot)$  is applied: If  $a \neq 0$  is not a negative integer, the result is

$$B_x(a, b) = \frac{x^a}{a} {}_2F_1(a, 1-b, a+1, x), \quad -a \notin \mathbb{N} \quad (7.7.5)$$

else if  $b \neq 0$  is not a negative integer, the result is

$$B_x(a, b) = B(a, b) - \frac{(1-x)^b x^a}{b} {}_2F_1(1, a+b, b+1, 1-x), \quad -b \notin \mathbb{N}. \quad (7.7.6)$$

## 7.7.6 Normalised incomplete beta functions

---

Function **IBetaMpMath**(*a* As mpNum, *b* As mpNum, *x* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **IBetaMpMath** returns the normalised incomplete beta function.

**Parameters:**

*a*: A real number.

*b*: A real number.

*x*: A real number.

This function returns the normalised incomplete beta function  $I_x(a, b)$  for  $a > 0$ ,  $b > 0$ , and  $0 \leq x \leq 1$ :

$$I_x(a, b) = \frac{B_x(a, b)}{B(a, b)}, \quad B_x(a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt. \quad (7.7.7)$$

There are some special cases

$$I_0(a, b) = 0, \quad I_1(a, b) = 1, \quad I_x(a, 1) = x^a, \quad (7.7.8)$$

and the symmetry relation  $I_x(a, b) = 1 - I_{1-x}(b, a)$ , which is used for  $x > a/(a+b)$ .

# Chapter 8

## Exponential integrals and error functions

Exponential integrals give closed-form solutions to a large class of commonly occurring transcendental integrals that cannot be evaluated using elementary functions. Integrals of this type include those with an integrand of the form  $t^a e^t$  or  $e^{-x^2}$ , the latter giving rise to the Gaussian (or normal) probability distribution.

All functions in this section can be reduced to the incomplete gamma function. The incomplete gamma function, in turn, can be expressed using hypergeometric functions (see Hypergeometric functions).

### 8.1 Exponential integrals

#### 8.1.1 Exponential integral Ei

---

Function **ei(z As mpNum) As mpNum**

---

The function `ei` returns the exponential integral.

**Parameter:**

`z`: A real or complex number.

The exponential integral is defined as

$$\text{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t} dt \quad (8.1.1)$$

When the integration range includes  $t = 0$ , the exponential integral is interpreted as providing the Cauchy principal value.

For real  $x$ , the Ei-function behaves roughly like  $\text{Ei}(x) \approx \exp(x) + \log(|x|)$ .

The Ei-function is related to the more general family of exponential integral functions denoted by  $E_n$ , which are available as `expint()`.

`ei()` supports complex arguments and arbitrary precision evaluation:

---

```
>>> mp.dps = 50
>>> ei(pi)
10.928374389331410348638445906907535171566338835056
>>> mp.dps = 25
>>> ei(3+4j)
```

---

(-4.154091651642689822535359 + 4.294418620024357476985535j)

---

### 8.1.2 Exponential integral E1

---

Function **e1(z As mpNum) As mpNum**

---

The function **e1** returns the exponential integral  $E_1(x)$ .

**Parameter:**

*z*: A real or complex number.

The exponential integral  $E_1(x)$  is defined as

$$E_1(x) = \int_z^\infty \frac{e^t}{t} dt \quad (8.1.2)$$

This is equivalent to `expint()` with  $n = 1$ .

The E1-function is essentially the same as the Ei-function (`ei()`) with negated argument, except for an imaginary branch cut term:

---

```
>>> e1(2.5)
0.02491491787026973549562801
>>> -ei(-2.5)
0.02491491787026973549562801
>>> e1(-2.5)
(-7.073765894578600711923552 - 3.141592653589793238462643j)
>>> -ei(2.5)
-7.073765894578600711923552
```

---

### 8.1.3 Generalized exponential integral En

---

Function **expint(n As mpNum, z As mpNum) As mpNum**

---

The function `expint` returns the generalized exponential integral or En-function.

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

The generalized exponential integral or En-function is defined as

$$E_n(x) = \int_1^\infty \frac{e^{-zt}}{t^n} dt \quad (8.1.3)$$

where *n* and *z* may both be complex numbers. The case with *n* is also given by `e1()`.

Examples

Evaluation at real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> expint(1, 6.25)
0.0002704758872637179088496194
>>> expint(-3, 2+3j)
```

---

```
(0.00299658467335472929656159 + 0.06100816202125885450319632j)
>>> expint(2+3j, 4-5j)
(0.001803529474663565056945248 - 0.002235061547756185403349091j)
```

---

### 8.1.4 Generalized Exponential Integrals $E_p$

---

Function **GeneralizedExponentialIntegralEpMpMath**(*x* As *mpNum*, *p* As *mpNum*) As *mpNum*  
 NOT YET IMPLEMENTED

---

The function **GeneralizedExponentialIntegralEpMpMath** returns the generalized exponential integrals  $E_n(x)$  of real order  $p$ .

**Parameters:**

*x*: A real number.  
*p*: A real number.

This function returns the generalized exponential integrals  $E_n(x)$  of real order  $p \in \mathbb{R}$

$$E_p(x) = x^{p-1} \int_x^\infty \frac{e^{-t}}{t^p} dt = \int_1^\infty \frac{e^{-xt}}{t^p} dt \quad (8.1.4)$$

with  $x > 0$  for  $p \leq 1$ , and  $x \geq 0$  for  $p > 1$ .

## 8.2 Logarithmic integral

### 8.2.1 logarithmic integral li

---

Function **li(z As mpNum) As mpNum**

---

The function **li** returns the logarithmic integral.

**Parameter:**

**z:** A real or complex number.

The logarithmic integral or li-function  $\text{li}(x)$  is defined by

$$\text{li}(x) = \int_0^x \frac{1}{\log(t)} dt \quad (8.2.1)$$

The logarithmic integral has a singularity at  $x = 1$ .

Alternatively, **li(x, offset=True)** computes the offset logarithmic integral (used in number theory)

$$\text{Li}(x) = \int_2^x \frac{1}{\log(t)} dt \quad (8.2.2)$$

These two functions are related via the simple identity  $\text{Li}(x) = \text{li}(x) - \text{li}(2)$ .

The logarithmic integral should also not be confused with the polylogarithm (also denoted by **Li**), which is implemented as **polylog()**.

Examples

Some basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 30; mp.pretty = True
>>> li(0)
0.0
>>> li(1)
-inf
>>> li(1)
-inf
>>> li(2)
1.04516378011749278484458888919
>>> findroot(li, 2)
1.45136923488338105028396848589
>>> li(inf)
+inf
>>> li(2, offset=True)
0.0
>>> li(1, offset=True)
-inf
>>> li(0, offset=True)
-1.04516378011749278484458888919
```

---

The logarithmic integral can be evaluated for arbitrary complex arguments:

---

```
>>> mp.dps = 20
>>> li(3+4j)
(3.1343755504645775265 + 2.6769247817778742392j)
```

---

## 8.3 Trigonometric integrals

### 8.3.1 cosine integral ci

---

Function **ci(z As mpNum) As mpNum**

---

The function ci returns the cosine integral.

**Parameter:**

*z*: A real or complex number.

The cosine integral is defined as

$$\text{Ci}(x) = \int_x^{\infty} \frac{\cos(t)}{t} dt = \gamma + \log(x) + \int_0^x \frac{\cos(t) - 1}{t} dt \quad (8.3.1)$$

Examples

Some values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> ci(0)
-inf
>>> ci(1)
0.3374039229009681346626462
>>> ci(pi)
0.07366791204642548599010096
>>> ci(inf)
0.0
>>> ci(-inf)
(0.0 + 3.141592653589793238462643j)
>>> ci(2+3j)
(1.408292501520849518759125 - 2.983617742029605093121118j)
```

---

### 8.3.2 sine integral si

---

Function **si(z As mpNum) As mpNum**

---

The function si returns the sine integral.

**Parameter:**

*z*: A real or complex number.

The sine integral is defined as

$$\text{Si}(x) = \int_0^x \frac{\sin(t)}{t} dt \quad (8.3.2)$$

The sine integral is thus the antiderivative of the sinc function (see sinc()).

Examples

Some values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
```

```
>>> si(0)
0.0
>>> si(1)
0.9460830703671830149413533
>>> si(-1)
-0.9460830703671830149413533
>>> si(pi)
1.851937051982466170361053
>>> si(inf)
1.570796326794896619231322
>>> si(-inf)
-1.570796326794896619231322
>>> si(2+3j)
(4.547513889562289219853204 + 1.399196580646054789459839j)
```

---

## 8.4 Hyperbolic integrals

### 8.4.1 hyperbolic cosine integral chi

---

Function **chi(z As mpNum) As mpNum**

---

The function **chi** returns the hyperbolic cosine integral.

**Parameter:**

**z:** A real or complex number.

The hyperbolic cosine integral, in analogy with the cosine integral (see **ci()**), is defined as

$$\text{Chi}(x) = \int_x^{\infty} \frac{\cosh(t)}{t} dt = \gamma + \log(x) + \int_0^x \frac{\cosh(t) - 1}{t} dt \quad (8.4.1)$$

Some values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> chi(0)
-inf
>>> chi(1)
0.8378669409802082408946786
>>> chi(inf)
+inf
>>> findroot(chi, 0.5)
0.5238225713898644064509583
>>> chi(2+3j)
(-0.1683628683277204662429321 + 2.625115880451325002151688j)
```

---

### 8.4.2 hyperbolic sine integral shi

---

Function **shi(z As mpNum) As mpNum**

---

The function **shi** returns the hyperbolic sine integral.

**Parameter:**

**z:** A real or complex number.

Computes the hyperbolic sine integral, defined in analogy with the sine integral (see **si()**) as

$$\text{Shi}(x) = \int_0^x \frac{\sinh(t)}{t} dt \quad (8.4.2)$$

Some values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> shi(0)
0.0
>>> shi(1)
1.057250875375728514571842
>>> shi(-1)
```

---

---

```
-1.057250875375728514571842
>>> shi(inf)
+inf
>>> shi(2+3j)
(-0.1931890762719198291678095 + 2.645432555362369624818525j)
```

---

## 8.5 Error functions

### 8.5.1 Error Function

---

Function **erf(z As mpNum)** As mpNum

---

The function `erf` returns the error function,  $\text{erf}(x)$ .

**Parameter:**

`z`: A real or complex number.

The error function is the normalized antiderivative of the Gaussian function  $\exp(-t^2)$ . More precisely,

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt \quad (8.5.1)$$

Basic examples

Simple values and limits include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> erf(0)
0.0
>>> erf(1)
0.842700792949715
>>> erf(-1)
-0.842700792949715
>>> erf(inf)
1.0
>>> erf(-inf)
-1.0
```

---

`erf()` implements arbitrary-precision evaluation and supports complex numbers:

---

```
>>> mp.dps = 50
>>> erf(0.5)
0.52049987781304653768274665389196452873645157575796
>>> mp.dps = 25
>>> erf(1+j)
(1.316151281697947644880271 + 0.1904534692378346862841089j)
```

---

See also `erfc()`, which is more accurate for large  $x$ , and `erfi()` which gives the antiderivative of  $\exp(t^2)$ . The Fresnel integrals `fresnels()` and `fresnelc()` are also related to the error function

## 8.5.2 Complementary Error Function

---

### Function `erfc(z As mpNum) As mpNum`

---

The function `erfc` returns the complementary error function,  $\text{erfc}(x) = 1 - \text{erf}(x)$ .

**Parameter:**

$z$ : A real or complex number.

This function avoids cancellation that occurs when naively computing the complementary error function as  $1 - \text{erf}(x)$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> 1 - erf(10)
0.0
>>> erfc(10)
2.08848758376254e-45
```

---

`erfc()` works accurately even for ludicrously large arguments:

---

```
>>> erfc(10**10)
4.3504398860243e-43429448190325182776
```

---

Complex arguments are supported:

---

```
>>> erfc(500+50j)
(1.19739830969552e-107492 + 1.46072418957528e-107491j)
```

---

## 8.5.3 Imaginary Error Function

---

### Function `erfi(z As mpNum) As mpNum`

---

The function `erfi` returns the imaginary error function,  $\text{erfi}(x)$ .

**Parameter:**

$z$ : A real or complex number.

The imaginary error function is defined in analogy with the error function, but with a positive sign in the integrand:

$$\text{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(t^2) dt \quad (8.5.2)$$

Whereas the error function rapidly converges to 1 as grows, the imaginary error function rapidly diverges to infinity. The functions are related as  $\text{erfi}(x) = -i \text{erf}(ix)$  for all complex numbers  $x$ .

Examples

Basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> erfi(0)
0.0
```

---

---

```
>>> erfi(1)
1.65042575879754
>>> erfi(-1)
-1.65042575879754
>>> erfi(inf)
+inf
>>> erfi(-inf)
-inf
```

---

Large arguments are supported:

---

```
>>> erfi(1000)
1.71130938718796e+434291
>>> erfi(10**10)
7.3167287567024e+43429448190325182754
>>> erfi(-10**10)
-7.3167287567024e+43429448190325182754
>>> erfi(1000-500j)
(2.49895233563961e+325717 + 2.6846779342253e+325717j)
>>> erfi(100000j)
(0.0 + 1.0j)
>>> erfi(-100000j)
(0.0 - 1.0j)
```

---

Complex arguments are supported:

---

```
>>> erfc(500+50j)
(1.19739830969552e-107492 + 1.46072418957528e-107491j)
```

---

## 8.5.4 Inverse Error Function

---

Function **erfinv(x As mpNum)** As mpNum

---

The function `erfinv` returns the inverse error function,  $\text{erfinv}(x)$ .

**Parameter:**

*x*: A real number.

The inverse error function satisfies  $\text{erf}(\text{erfinv}(x)) = \text{erfinv}(\text{erf}(x)) = x$ . This function is defined only for  $-1 < x < 1$ .

Examples

Special values include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> erfinv(0)
0.0
>>> erfinv(1)
+inf
>>> erfinv(-1)
-inf
```

---

`erfinv()` supports arbitrary-precision evaluation:

---

```
>>> mp.dps = 50
>>> x = erf(2)
>>> x
0.99532226501895273416206925636725292861089179704006
>>> erfinv(x)
2.0
```

---

## 8.6 The normal distribution

### 8.6.1 The normal probability density function

---

Function **npdf(*x* As mpNum, *mu* As mpNum, *sigma* As mpNum)** As mpNum

---

The function **npdf** returns the normal probability density function.

**Parameters:**

*x*: A real number.

*mu*: A real number.

*sigma*: A real number.

**npdf(*x*, mu=0, sigma=1)** evaluates the probability density function of a normal distribution with mean value  $\mu$  and variance  $\sigma^2$ .

Elementary properties of the probability distribution can be verified using numerical integration:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> quad(npdf, [-inf, inf])
1.0
>>> quad(lambda x: npdf(x, 3), [3, inf])
0.5
>>> quad(lambda x: npdf(x, 3, 2), [3, inf])
0.5
```

---

### 8.6.2 The normal cumulative distribution function

---

Function **ncdf(*x* As mpNum, *mu* As mpNum, *sigma* As mpNum)** As mpNum

---

The function **ncdf** returns the normal cumulative distribution function.

**Parameters:**

*x*: A real number.

*mu*: A real number.

*sigma*: A real number.

**ncdf(*x*, mu=0, sigma=1)** evaluates the cumulative distribution function of a normal distribution with mean value  $\mu$  and variance  $\sigma^2$ .

See also **npdf()**, which gives the probability density.

Elementary properties include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> ncdf(pi, mu=pi)
0.5
>>> ncdf(-inf)
0.0
>>> ncdf(+inf)
1.0
```

---

## 8.7 Fresnel integrals

### 8.7.1 Fresnel sine integral

---

Function **fresnels(z As mpNum) As mpNum**

---

The function `fresnels` returns the Fresnel sine integral.

**Parameter:**

*z*: A real or complex number.

The Fresnel sine integral is defined as

$$S(x) = \int_0^x \sin\left(\frac{\pi t^2}{2}\right) dt \quad (8.7.1)$$

Note that some sources define this function without the normalization factor  $\pi/2$ .

Examples

Some basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> fresnels(0)
0.0
>>> fresnels(inf)
0.5
>>> fresnels(-inf)
-0.5
>>> fresnels(1)
0.4382591473903547660767567
>>> fresnels(1+2j)
(36.72546488399143842838788 + 15.58775110440458732748279j)
```

---

### 8.7.2 Fresnel cosine integral

---

Function **fresnelc(z As mpNum) As mpNum**

---

The function `fresnelc` returns the Fresnel cosine integral.

**Parameter:**

*z*: A real or complex number.

The Fresnel cosine integral is defined as

$$C(x) = \int_0^x \cos\left(\frac{\pi t^2}{2}\right) dt \quad (8.7.2)$$

Note that some sources define this function without the normalization factor  $\pi/2$ .

Examples

Some basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
```

---

```
>>> fresnelc(0)
0.0
>>> fresnelc(inf)
0.5
>>> fresnelc(-inf)
-0.5
>>> fresnelc(1)
0.7798934003768228294742064
>>> fresnelc(1+2j)
(16.08787137412548041729489 - 36.22568799288165021578758j)
```

---

## 8.8 Other Special Functions

### 8.8.1 Lambert W function

---

Function **lambertw(z As mpNum, *Keywords* As String)** As mpNum

---

The function `lambertw` returns the Lambert W function.

**Parameters:**

*z*: A real or complex number.

*Keywords*: *k*=0.

The Lambert W function  $W(z)$  is defined as the inverse function of  $w \exp(w)$ . In other words, the value of  $W(z)$  is such that  $z = W(z) \exp(W(z))$  for any complex number  $z$ .

The Lambert W function is a multivalued function with infinitely many branches  $W_k(z)$ , indexed by  $k \in \mathbb{Z}$ . Each branch gives a different solution  $w$  of the equation  $z = w \exp(w)$ . All branches are supported by `lambertw()`:

`lambertw(z)` gives the principal solution (branch 0).

`lambertw(z, k)` gives the solution on branch  $k$ .

The Lambert W function has two partially real branches: the principal branch ( $k = 0$ ) is real for real  $z > -1/e$ , and the branch  $k = -1$  is real for  $-1/e < z < 0$ . All branches except  $k = 0$  have a logarithmic singularity at  $z = 0$ .

The definition, implementation and choice of branches is based on [Corless \*et al.\* \(1996\)](#).

**Basic examples**

The Lambert W function is the inverse of  $w \exp(w)$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> w = lambertw(1)
>>> w
0.5671432904097838729999687
>>> w*exp(w)
1.0
```

---

Any branch gives a valid inverse:

---

```
>>> w = lambertw(1, k=3)
>>> w
(-2.853581755409037807206819 + 17.11353553941214591260783j)
>>> w = lambertw(1, k=25)
>>> w
(-5.047020464221569709378686 + 155.4763860949415867162066j)
>>> chop(w*exp(w))
1.0
```

---

## 8.8.2 Arithmetic-geometric mean

---

Function **agm(*a* As mpNum, *b* As mpNum)** As mpNum

---

The function **agm** returns the arithmetic-geometric mean of *a* and *b*.

**Parameters:**

*a*: A real or complex number.

*b*: A real or complex number.

`agm(a, b)` computes the arithmetic-geometric mean of *a* and *b*, defined as the limit of the following iteration:

$$a_0 = a; \quad b_0 = b; \quad a_{n+1} = \frac{1}{2}(a_n + b_n); \quad b_{n+1} = \sqrt{a_n b_n}. \quad (8.8.1)$$

This function can be called with a single argument, computing  $\text{agm}(a, 1) = \text{agm}(1, a)$ .

Examples

It is a well-known theorem that the geometric mean of two distinct positive numbers is less than the arithmetic mean. It follows that the arithmetic-geometric mean lies between the two means:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> a = mpf(3)
>>> b = mpf(4)
>>> sqrt(a*b)
3.46410161513775
>>> agm(a,b)
3.48202767635957
>>> (a+b)/2
3.5
```

---

The arithmetic-geometric mean can also be computed for complex numbers:

---

```
>>> agm(3, 2+j)
(2.51055133276184 + 0.547394054060638j)
```

---

A formula for  $\Gamma(1/4)$ :

---

```
>>> gamma(0.25)
3.62560990822191
>>> sqrt(2*sqrt(2*pi**3)/agm(1,sqrt(2)))
3.62560990822191
```

---

# Chapter 9

## Bessel functions and related functions

The functions in this section arise as solutions to various differential equations in physics, typically describing wavelike oscillatory behavior or a combination of oscillation and exponential decay or growth. Mathematically, they are special cases of the confluent hypergeometric functions  ${}_0F_1$ ,  ${}_1F_1$  and  ${}_1F_2$  (see Hypergeometric functions).

### 9.1 Bessel functions

#### 9.1.1 Exponentially scaled Bessel function $I_{\nu,e}(x)$

---

Function **BesselMpMath**(*x* As *mpNum*, *ν* As *mpNum*) As *mpNum*

---

NOT YET IMPLEMENTED

---

The function **BesselMpMath** returns  $I_{\nu,e}(x) = I_{\nu}(x) \exp(-|x|)$ , the exponentially scaled modified Bessel function  $I_{\nu}(z)$  of the first kind of order  $\nu$ ,  $x \geq 0$  if  $\nu$  is not an integer.

**Parameters:**

*x*: A real number.

*ν*: A real number.

#### 9.1.2 Exponentially scaled Bessel function $K_{\nu,e}(x)$

---

Function **BesselKeMpMath**(*x* As *mpNum*, *ν* As *mpNum*) As *mpNum*

---

NOT YET IMPLEMENTED

---

The function **BesselKeMpMath** returns  $K_{\nu,e}(x) = K_{\nu}(x) \exp(x)$ , the exponentially scaled modified Bessel function  $K_{\nu}(z)$  of the first kind of order  $\nu$ ,  $x > 0$ .

**Parameters:**

*x*: A real number.

*ν*: A real number.

#### 9.1.3 Bessel function of the first kind

---

Function **besselj**(*n* As *mpNum*, *x* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function **besselj** returns the Bessel function of the first kind  $J_n(x)$ .

**Parameters:***n*: A real or complex number.*x*: A real or complex number.*Keywords*: derivative=0.

Bessel functions of the first kind are defined as solutions of the differential equation

$$x^2y'' + xy' + (x^2 - n^2)y = 0 \quad (9.1.1)$$

which appears, among other things, when solving the radial part of Laplace's equation in cylindrical coordinates. This equation has two solutions for given *n*, where the  $J_n$ -function is the solution that is nonsingular at  $x = 0$ . For positive integer *n*,  $J_n(x)$  behaves roughly like a sine (odd *n*) or cosine (even *n*) multiplied by a magnitude factor that decays slowly as  $x \rightarrow \pm\infty$ .

Generally,  $J_n$  is a special case of the hypergeometric function  ${}_0F_1$ :

$$J_n(x) = \frac{x^n}{2^n \Gamma(n+1)} {}_0F_1 \left( n+1, -\frac{x^2}{4} \right) \quad (9.1.2)$$

With derivative = *m*  $\neq 0$ , the *m*-th derivative

$$\frac{d^m}{dx^m} J_n(x) \quad (9.1.3)$$

is computed.

Evaluation is supported for arbitrary arguments, and at arbitrary precision:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> besselj(2, 1000)
-0.024777229528606
>>> besselj(4, 0.75)
0.000801070086542314
>>> besselj(2, 1000j)
(-2.48071721019185e+432 + 6.41567059811949e-437j)
>>> mp.dps = 25
>>> besselj(0.75j, 3+4j)
(-2.778118364828153309919653 - 1.5863603889018621585533j)
>>> mp.dps = 50
>>> besselj(1, pi)
0.28461534317975275734531059968613140570981118184947
```

---

Arguments may be large:

---

```
>>> mp.dps = 25
>>> besselj(0, 10000)
-0.007096160353388801477265164
>>> besselj(0, 10**10)
0.000002175591750246891726859055
>>> besselj(2, 10**100)
7.337048736538615712436929e-51
>>> besselj(2, 10**5*j)
(-3.540725411970948860173735e+43426 + 4.4949812409615803110051e-43433j)
```

---

Derivatives of any order can be computed (negative orders correspond to integration):

---

```

>>> mp.dps = 25
>>> besselj(0, 7.5, 1)
-0.1352484275797055051822405
>>> diff(lambda x: besselj(0,x), 7.5)
-0.1352484275797055051822405
>>> besselj(0, 7.5, 10)
-0.1377811164763244890135677
>>> diff(lambda x: besselj(0,x), 7.5, 10)
-0.1377811164763244890135677
>>> besselj(0,7.5,-1) - besselj(0,3.5,-1)
-0.1241343240399987693521378
>>> quad(j0, [3.5, 7.5])
-0.1241343240399987693521378

```

---

### Function **j0(x As mpNum) As mpNum**

The function **j0** returns the Bessel function  $J_0(x)$ .

**Parameter:**

*x*: A real or complex number.

Computes the Bessel function  $J_0(x)$ . See **besselj()**.

---

### Function **j1(x As mpNum) As mpNum**

The function **j1** returns the Bessel function  $J_1(x)$ .

**Parameter:**

*x*: A real or complex number.

Computes the Bessel function  $J_1(x)$ . See **besselj()**.

## 9.1.4 Bessel function of the second kind

---

### Function **bessely(n As mpNum, x As mpNum, *Keywords* As String) As mpNum**

The function **bessely** returns the Bessel function of the second kind  $Y_n(x)$ .

**Parameters:**

*n*: A real or complex number.

*x*: A real or complex number.

**Keywords:** derivative=0.

The Bessel function of the second kind are defined as

$$Y_n(x) = \frac{J_n(x) \cos(\pi n) - J_{-n}(x)}{\sin(\pi n)} \quad (9.1.4)$$

For *n* an integer, this formula should be understood as a limit. With derivative = *m*  $\neq 0$ , the *m*-th derivative

$$\frac{d^m}{dx^m} Y_n(x) \quad (9.1.5)$$

is computed.

Evaluation is supported for arbitrary arguments, and at arbitrary precision:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> bessely(0,0), bessely(1,0), bessely(2,0)
(-inf, -inf, -inf)
>>> bessely(1, pi)
0.3588729167767189594679827
>>> bessely(0.5, 3+4j)
(9.242861436961450520325216 - 3.085042824915332562522402j)
```

---

Arguments may be large:

---

```
>>> bessely(0, 10000)
0.00364780555898660588668872
>>> bessely(2.5, 10**50)
-4.8952500412050989295774e-26
>>> bessely(2.5, -10**50)
(0.0 + 4.8952500412050989295774e-26j)
```

---

Derivatives and antiderivatives of any order can be computed:

---

```
>>> bessely(2, 3.5, 1)
0.3842618820422660066089231
>>> diff(lambda x: bessely(2, x), 3.5)
0.3842618820422660066089231
>>> bessely(0.5, 3.5, 1)
-0.2066598304156764337900417
>>> diff(lambda x: bessely(0.5, x), 3.5)
-0.2066598304156764337900417
>>> diff(lambda x: bessely(2, x), 0.5, 10)
-208173867409.5547350101511
>>> bessely(2, 0.5, 10)
-208173867409.5547350101511
>>> bessely(2, 100.5, 100)
0.02668487547301372334849043
>>> quad(lambda x: bessely(2,x), [1,3])
-1.377046859093181969213262
>>> bessely(2,3,-1) - bessely(2,1,-1)
-1.377046859093181969213262
```

---

### 9.1.5 Modified Bessel function of the first kind

---

Function **besseli**(*n* As mpNum, *x* As mpNum, **Keywords** As String) As mpNum

---

The function **besseli** returns the modified Bessel function of the first kind  $J_n(x)$ .

**Parameters:**

*n*: A real or complex number.

*x*: A real or complex number.

**Keywords:** derivative=0.

The modified Bessel function of the first kind are defined as

$$I_n(x) = i^{-n} J_n(ix) \quad (9.1.6)$$

With derivative =  $m \neq 0$ , the  $m$ -th derivative

$$\frac{d^m}{dx^m} I_n(x) \quad (9.1.7)$$

is computed.

Evaluation is supported for arbitrary arguments, and at arbitrary precision:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> besseli(0,0)
1.0
>>> besseli(1,0)
0.0
>>> besseli(0,1)
1.266065877752008335598245
>>> besseli(3.5, 2+3j)
(-0.2904369752642538144289025 - 0.4469098397654815837307006j)
```

---

Arguments may be large:

---

```
>>> besseli(2, 1000)
2.480717210191852440616782e+432
>>> besseli(2, 10**10)
4.299602851624027900335391e+4342944813
>>> besseli(2, 6000+10000j)
(-2.114650753239580827144204e+2603 + 4.385040221241629041351886e+2602j)
```

---

Derivatives and antiderivatives of any order can be computed:

---

```
>>> mp.dps = 25
>>> besseli(2, 7.5, 1)
195.8229038931399062565883
>>> diff(lambda x: besseli(2,x), 7.5)
195.8229038931399062565883
>>> besseli(2, 7.5, 10)
153.3296508971734525525176
>>> diff(lambda x: besseli(2,x), 7.5, 10)
153.3296508971734525525176
>>> besseli(2,7.5,-1) - besseli(2,3.5,-1)
202.5043900051930141956876
>>> quad(lambda x: besseli(2,x), [3.5, 7.5])
202.5043900051930141956876
```

---

## 9.1.6 Modified Bessel function of the second kind

---

Function **besselk**(*n* As *mpNum*, *x* As *mpNum*) As *mpNum*

---

The function **besselk** returns the modified Bessel function of the second kind  $K_n(x)$ .

**Parameters:**

*n*: A real or complex number.

*x*: A real or complex number.

The modified Bessel function of the second kind are defined as

$$K_n(x) = \frac{\pi}{2} \frac{I_{-n}(x) - I_n(x)}{\sin(\pi n)} \quad (9.1.8)$$

For *n* an integer, this formula should be understood as a limit.

Evaluation is supported for arbitrary arguments, and at arbitrary precision:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> besselk(0,1)
0.4210244382407083333356274
>>> besselk(0, -1)
(0.4210244382407083333356274 - 3.97746326050642263725661j)
>>> besselk(3.5, 2+3j)
(-0.02090732889633760668464128 + 0.2464022641351420167819697j)
>>> besselk(2+3j, 0.5)
(0.9615816021726349402626083 + 0.1918250181801757416908224j)
```

---

Arguments may be large:

---

```
>>> besselk(0, 100)
4.656628229175902018939005e-45
>>> besselk(1, 10**6)
4.131967049321725588398296e-434298
>>> besselk(1, 10**6*j)
(0.001140348428252385844876706 - 0.0005200017201681152909000961j)
>>> besselk(4.5, fmul(10**50, j, exact=True))
(1.561034538142413947789221e-26 + 1.243554598118700063281496e-25j)
```

---

## 9.2 Bessel function zeros

### 9.2.1 Zeros of the Bessel function of the first kind

---

Function **besseljzero(*v* As mpNum, *m* As mpNum, *Keywords* As String) As mpNum**

---

The function `besseljzero` returns the *m*-th positive zero of the Bessel function of the first kind

**Parameters:**

*v*: A real or complex number.

*m*: A real or complex number.

*Keywords*: derivative=0.

For a real order  $\nu \geq 0$  and a positive integer  $m$ , returns  $j_{\nu,m}$ , the *m*-th positive zero of the Bessel function of the first kind  $J_\nu(z)$  (see `besselj()`). Alternatively, with derivative=1, gives the first nonnegative simple zero  $j'_{\nu,m}$  of  $J'_\nu(z)$ .

The indexing convention is that used by Abramowitz & Stegun and the DLMF. Note the special case  $j'_{0,1} = 0$ , while all other zeros are positive. In effect, only simple zeros are counted (all zeros of Bessel functions are simple except possibly  $z = 0$ ) and becomes a monotonic function of both  $\nu$  and  $m$ .

The zeros are interlaced according to the inequalities

$$j'_{\nu,k} < j_{\nu,k} < j'_{\nu,k+1} \quad (9.2.1)$$

$$j_{\nu,1} < j_{\nu+1,2} < j_{\nu,2} < j'_{\nu,k+1} < j_{\nu+1,2} < j_{\nu,3} \quad (9.2.2)$$

Initial zeros of the Bessel functions  $J_0(z)$ ,  $J_1(z)$ ,  $J_2(z)$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> besseljzero(0,1); besseljzero(0,2); besseljzero(0,3)
2.404825557695772768621632
5.520078110286310649596604
8.653727912911012216954199
>>> besseljzero(1,1); besseljzero(1,2); besseljzero(1,3)
3.831705970207512315614436
7.01558666981561875353705
10.17346813506272207718571
>>> besseljzero(2,1); besseljzero(2,2); besseljzero(2,3)
5.135622301840682556301402
8.417244140399864857783614
11.61984117214905942709415
```

---

Initial zeros of  $J'_0(z)$ ,  $J'_1(z)$ ,  $J'_2(z)$ :

---

```
>>> besseljzero(0,1,1); besseljzero(0,2,1); besseljzero(0,3,1)
0.0
3.831705970207512315614436
7.01558666981561875353705
>>> besseljzero(1,1,1); besseljzero(1,2,1); besseljzero(1,3,1)
1.84118378134065930264363
5.331442773525032636884016
8.536316366346285834358961
>>> besseljzero(2,1,1); besseljzero(2,2,1); besseljzero(2,3,1)
```

---

```
3.054236928227140322755932
6.706133194158459146634394
9.969467823087595793179143
```

---

Zeros with large index:

---

```
>>> besseljzero(0,100); besseljzero(0,1000); besseljzero(0,10000)
313.3742660775278447196902
3140.807295225078628895545
31415.14114171350798533666
>>> besseljzero(5,100); besseljzero(5,1000); besseljzero(5,10000)
321.1893195676003157339222
3148.657306813047523500494
31422.9947255486291798943
>>> besseljzero(0,100,1); besseljzero(0,1000,1); besseljzero(0,10000,1)
311.8018681873704508125112
3139.236339643802482833973
31413.57032947022399485808
```

---

## 9.2.2 Zeros of the Bessel function of the second kind

---

Function **besselyzero**(*v* As *mpNum*, *m* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function **besselyzero** returns the *m*-th positive zero of the Bessel function of the second kind

**Parameters:**

*v*: A real or complex number.

*m*: A real or complex number.

**Keywords**: derivative=0.

For a real order  $\nu > 0$  and a positive integer  $m$ , returns  $y_{\nu,m}$ , the *m*-th positive zero of the Bessel function of the second kind  $Y_\nu(z)$  (see **besselj()**). Alternatively, with derivative=1, gives the first nonnegative simple zero  $y'_{\nu,m}$  of  $Y'_\nu(z)$ .

The zeros are interlaced according to the inequalities

$$y'_{\nu,k} < y_{\nu,k} < y'_{\nu,k+1} \quad (9.2.3)$$

$$y_{\nu,1} < y_{\nu+1,2} < y_{\nu,2} < y'_{\nu,k+1} < y_{\nu+1,2} < y_{\nu,3} \quad (9.2.4)$$

Initial zeros of the Bessel functions  $Y_0(z)$ ,  $Y_1(z)$ ,  $Y_2(z)$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> besselyzero(0,1); besselyzero(0,2); besselyzero(0,3)
0.8935769662791675215848871
3.957678419314857868375677
7.086051060301772697623625
>>> besselyzero(1,1); besselyzero(1,2); besselyzero(1,3)
2.197141326031017035149034
5.429681040794135132772005
8.596005868331168926429606
>>> besselyzero(2,1); besselyzero(2,2); besselyzero(2,3)
3.384241767149593472701426
```

```
6.793807513268267538291167
10.02347797936003797850539
```

---

Initial zeros of  $Y'_0(z)$ ,  $Y'_1(z)$ ,  $Y'_2(z)$ ::

```
>>> besselyzero(0,1,1); besselyzero(0,2,1); besselyzero(0,3,1)
2.197141326031017035149034
5.429681040794135132772005
8.596005868331168926429606
>>> besselyzero(1,1,1); besselyzero(1,2,1); besselyzero(1,3,1)
3.683022856585177699898967
6.941499953654175655751944
10.12340465543661307978775
>>> besselyzero(2,1,1); besselyzero(2,2,1); besselyzero(2,3,1)
5.002582931446063945200176
8.350724701413079526349714
11.57419546521764654624265
```

---

Zeros with large index:

```
>>> besselyzero(0,100); besselyzero(0,1000); besselyzero(0,10000)
311.8034717601871549333419
3139.236498918198006794026
31413.57034538691205229188
>>> besselyzero(5,100); besselyzero(5,1000); besselyzero(5,10000)
319.6183338562782156235062
3147.086508524556404473186
31421.42392920214673402828
>>> besselyzero(0,100,1); besselyzero(0,1000,1); besselyzero(0,10000,1)
313.3726705426359345050449
3140.807136030340213610065
31415.14112579761578220175
```

---

## 9.3 Hankel functions

### 9.3.1 Hankel function of the first kind

---

Function **hankel1(*n* As mpNum, *x* As mpNum) As mpNum**

---

The function `hankel1` returns the Hankel function of the first kind

**Parameters:**

*n*: A real or complex number.

*x*: A real or complex number.

The Hankel function of the first kind is the complex combination of Bessel functions given by

$$H_n^{(1)}(x) = J_n(x) + iY_n(x). \quad (9.3.1)$$

The Hankel function is generally complex-valued:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hankel1(2, pi)
(0.4854339326315091097054957 - 0.0999007139290278787734903j)
>>> hankel1(3.5, pi)
(0.2340002029630507922628888 - 0.6419643823412927142424049j)
```

---

### 9.3.2 Hankel function of the second kind

---

Function **hankel2(*n* As mpNum, *x* As mpNum) As mpNum**

---

The function `hankel2` returns the Hankel function of the second kind

**Parameters:**

*n*: A real or complex number.

*x*: A real or complex number.

The Hankel function of the second kind is the complex combination of Bessel functions given by

$$H_n^{(2)}(x) = J_n(x) - iY_n(x). \quad (9.3.2)$$

The Hankel function is generally complex-valued:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hankel2(2, pi)
(0.4854339326315091097054957 + 0.0999007139290278787734903j)
>>> hankel2(3.5, pi)
(0.2340002029630507922628888 + 0.6419643823412927142424049j)
```

---

## 9.4 Kelvin functions

### 9.4.1 Kelvin function ber

---

Function **ber(*n* As mpNum, *z* As mpNum) As mpNum**

---

The function **ber** returns the Kelvin function ber

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

The Kelvin function ber returns for real arguments the real part of the Bessel J function of a rotated argument

$$J_n(xe^{3\pi i/4}) = \text{ber}_n(x) + i\text{bei}_n(x). \quad (9.4.1)$$

The imaginary part is given by **bei()**.

Verifying the defining relation:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> n, x = 2, 3.5
>>> ber(n,x)
1.442338852571888752631129
>>> bei(n,x)
-0.948359035324558320217678
>>> besselj(n, x*root(1,8,3))
(1.442338852571888752631129 - 0.948359035324558320217678j)
```

---

The ber and bei functions are also defined by analytic continuation for complex arguments:

---

```
>>> ber(1+j, 2+3j)
(4.675445984756614424069563 - 15.84901771719130765656316j)
>>> bei(1+j, 2+3j)
(15.83886679193707699364398 + 4.684053288183046528703611j)
```

---

### 9.4.2 Kelvin function bei

---

Function **bei(*n* As mpNum, *z* As mpNum) As mpNum**

---

The function **bei** returns the Kelvin function bei

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

The Kelvin function bei returns for real arguments the imaginary part of the Bessel J function of a rotated argument. See **ber()**.

### 9.4.3 Kelvin function ker

---

Function **ker(*n* As mpNum, *z* As mpNum) As mpNum**

---

The function `ker` returns the Kelvin function  $\text{ker}$

**Parameters:**

$n$ : A real or complex number.

$z$ : A real or complex number.

The Kelvin function `ker` returns for real arguments the real part of the (rescaled) Bessel K function of a rotated argument

$$e^{-\pi i/2} K_n(xe^{3\pi i/4}) = \text{ker}_n(x) + i\text{kei}_n(x). \quad (9.4.2)$$

The imaginary part is given by `kei()`.

Verifying the defining relation:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> n, x = 2, 4.5
>>> ker(n,x)
0.02542895201906369640249801
>>> kei(n,x)
-0.02074960467222823237055351
>>> exp(-n*pi*j/2) * besselk(n, x*root(1,8,1))
(0.02542895201906369640249801 - 0.02074960467222823237055351j)
```

---

The `ker` and `kei` functions are also defined by analytic continuation for complex arguments:

---

```
>>> ker(1+j, 3+4j)
(1.586084268115490421090533 - 2.939717517906339193598719j)
>>> kei(1+j, 3+4j)
(-2.940403256319453402690132 - 1.585621643835618941044855j)
```

---

#### 9.4.4 Kelvin function `kei`

---

Function **kei( $n$  As `mpNum`,  $z$  As `mpNum`) As `mpNum`**

---

The function `kei` returns the Kelvin function  $\text{kei}$

**Parameters:**

$n$ : A real or complex number.

$z$ : A real or complex number.

The Kelvin function `kei` returns for real arguments the imaginary part of the (rescaled) Bessel K function of a rotated argument. See `ker()`.

## 9.5 Struve Functions

The Struve functions  $\mathbf{H}_\nu(x)$  and the modified Struve functions  $\mathbf{L}_\nu(x)$  have the power series expansions (see [Abramowitz & Stegun. \(1970\)](#) [1, 12.1.3 and 12.2.1]):

$$\mathbf{H}_\nu(x) = \left(\frac{1}{2}x\right)^{\nu+1} \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{1}{2}x\right)^{2k}}{\Gamma\left(k + \frac{3}{2}\right) \Gamma\left(k + \nu + \frac{3}{2}\right)} \quad (9.5.1)$$

$$\mathbf{L}_\nu(x) = \left(\frac{1}{2}x\right)^{\nu+1} \sum_{k=0}^{\infty} \frac{\left(\frac{1}{2}x\right)^{2k}}{\Gamma\left(k + \frac{3}{2}\right) \Gamma\left(k + \nu + \frac{3}{2}\right)} \quad (9.5.2)$$

### 9.5.1 Struve function H

---

Function **struveh(*n* As mpNum, *z* As mpNum)** As mpNum

---

The function **struveh** returns the Struve function H

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

The Struve function H is defined as

$$\mathbf{H}_n(z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\Gamma\left(k + \frac{3}{2}\right) \Gamma\left(k + n + \frac{3}{2}\right)} \left(\frac{z}{2}\right)^{2k+n+1} \quad (9.5.3)$$

which is a solution to the Struve differential equation

$$z^2 f''(z) + z f'(z) + (z^2 - n^2) f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!} \quad (9.5.4)$$

Examples Evaluation for arbitrary real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> struveh(0, 3.5)
0.3608207733778295024977797
>>> struveh(-1, 10)
-0.255212719726956768034732
>>> struveh(1, -100.5)
0.5819566816797362287502246
>>> struveh(2.5, 100000000000000)
3153915652525200060.308937
>>> struveh(2.5, -100000000000000)
(0.0 - 3153915652525200060.308937j)
>>> struveh(1+j, 1000000+4000000j)
(-3.066421087689197632388731e+1737173 - 1.596619701076529803290973e+1737173j)
```

---

## 9.5.2 Modified Struve function L

---

Function **struvel(*n* As mpNum, *z* As mpNum) As mpNum**

---

The function **struvel** returns the modified Struve function L

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

The modified Struve function  $L_n(z)$  is defined as

$$L_n(z) = -ie^{-n\pi i/2} H_n(iz) \quad (9.5.5)$$

which solves to the modified Struve differential equation

$$z^2 f''(z) + z f'(z) + (z^2 + n^2) f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!} \quad (9.5.6)$$

Examples

Evaluation for arbitrary real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> struvel(0, 3.5)
7.180846515103737996249972
>>> struvel(-1, 10)
2670.994904980850550721511
>>> struvel(1, -100.5)
1.757089288053346261497686e+42
>>> struvel(2.5, 100000000000000)
4.160893281017115450519948e+4342944819025
>>> struvel(2.5, -100000000000000)
(0.0 - 4.160893281017115450519948e+4342944819025j)
>>> struvel(1+j, 700j)
(-0.1721150049480079451246076 + 0.1240770953126831093464055j)
>>> struvel(1+j, 1000000+4000000j)
(-2.973341637511505389128708e+434290 - 5.164633059729968297147448e+434290j)
```

---

## 9.6 Anger-Weber functions

### 9.6.1 Anger function J

---

Function **angerj**(*v* As mpNum, *z* As mpNum) As mpNum

---

The function **angerj** returns the Anger function J

**Parameters:**

*v*: A real or complex number.

*z*: A real or complex number.

The Anger function  $\mathbf{J}_\nu(z)$  is defined as

$$\mathbf{J}_\nu(z) = \frac{1}{\pi} \int_0^\pi \cos(\nu t - z \sin(t)) dt \quad (9.6.1)$$

which is an entire function of both the parameter  $\nu$  and the argument  $z$ . It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z} f'(z) + \left(1 - \frac{\nu^2}{z^2}\right) f(z) = \frac{(z - \nu)}{\pi z^2} \sin(\pi\nu). \quad (9.6.2)$$

Examples

Evaluation for real and complex parameter and argument:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> angerj(2,3)
0.4860912605858910769078311
>>> angerj(-3+4j, 2+5j)
(-5033.358320403384472395612 + 585.8011892476145118551756j)
>>> angerj(3.25, 1e6j)
(4.630743639715893346570743e+434290 - 1.117960409887505906848456e+434291j)
>>> angerj(-1.5, 1e6)
0.0002795719747073879393087011
```

---

### 9.6.2 Weber function E

---

Function **webere**(*v* As mpNum, *z* As mpNum) As mpNum

---

The function **webere** returns the Weber function E

**Parameters:**

*v*: A real or complex number.

*z*: A real or complex number.

The Weber function  $\mathbf{E}_\nu(z)$  is defined as

$$\mathbf{E}_\nu(z) = \frac{1}{\pi} \int_0^\pi \sin(\nu t - z \sin(t)) dt \quad (9.6.3)$$

which is an entire function of both the parameter  $\nu$  and the argument  $z$ . It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z} f'(z) + \left(1 - \frac{\nu^2}{z^2}\right) f(z) = \frac{1}{\pi z^2} (z + \nu + (z - \nu) \cos(\pi\nu)). \quad (9.6.4)$$

Examples

Evaluation for real and complex parameter and argument:

---

```
>>> from mpmath import *
>>> mp.dps = 25; mp.pretty = True
>>> webere(2,3)
-0.1057668973099018425662646
>>> webere(-3+4j, 2+5j)
(-585.8081418209852019290498 - 5033.314488899926921597203j)
>>> webere(3.25, 1e6j)
(-1.117960409887505906848456e+434291 - 4.630743639715893346570743e+434290j)
>>> webere(3.25, 1e6)
-0.00002812518265894315604914453
```

---

## 9.7 Lommel functions

### 9.7.1 First Lommel function s

---

Function **lommels1(*u* As mpNum, *v* As mpNum, *z* As mpNum) As mpNum**

---

The function lommels1 returns the First Lommel functions s

**Parameters:**

- u*: A real or complex number.
- v*: A real or complex number.
- z*: A real or complex number.

The Lommel function  $s_{\mu,\nu}$  or  $s_{\mu,\nu}^{(1)}$  is defined as

$$s_{\mu,\nu} = \frac{z^{\mu+1}}{(\mu - \nu + 1)(\mu + \nu + 1)} {}_1F_2 \left( 1; \frac{\mu - \nu + 3}{2}, \frac{\mu + \nu + 3}{2}; -\frac{z^2}{4} \right) \quad (9.7.1)$$

which solves the inhomogeneous Bessel equation

$$z^2 f''(z) + z f'(z) + (z^2 - \nu^2) f(z) = z^{\mu+1}. \quad (9.7.2)$$

A second solution is given by lommels2().

An integral representation:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> u,v,z = 0.25, 0.125, mpf(0.75)
>>> lommels1(u,v,z)
0.4276243877565150372999126
>>> (bessely(v,z)*quad(lambda t: t**u*besselj(v,t), [0,z]) - \
... besselj(v,z)*quad(lambda t: t**u*bessely(v,t), [0,z]))*(pi/2
0.4276243877565150372999126
```

---

### 9.7.2 Second Lommel function S

---

Function **lommels2(*u* As mpNum, *v* As mpNum, *z* As mpNum) As mpNum**

---

The function lommels2 returns the Second Lommel functions S

**Parameters:**

- u*: A real or complex number.
- v*: A real or complex number.
- z*: A real or complex number.

The second Lommel function or  $S_{\mu,\nu}$  or  $s_{\mu,\nu}^{(2)}$  is defined as

$$S_{\mu,\nu}(z) = s_{\mu,\nu}(z) + 2^{\mu-1} \Gamma\left(\frac{1}{2}(\mu - \nu + 1)\right) \Gamma\left(\frac{1}{2}(\mu + \nu + 1)\right) \times \left[ \sin\left(\frac{1}{2}(\mu - \nu)\pi\right) J_{\nu}(z) - \cos\left(\frac{1}{2}(\mu - \nu)\pi\right) Y_{\nu}(z) \right] \quad (9.7.3)$$

which solves the same differential equation as lommels1().

Verifying the differential equation:

---

```
>>> f = lambda z: lommels2(u,v,z)
>>> z**2*diff(f,z,2) + z*diff(f,z) + (z**2-v**2)*f(z)
0.6495190528383289850727924
>>> z**(u+1)
0.6495190528383289850727924
```

---

## 9.8 Airy and Scorer functions

### 9.8.1 Airy function Ai

---

Function **airyai(z As mpNum, Keywords As String)** As mpNum

---

The function **airyai** returns the Airy function Ai

**Parameters:**

*z*: A real or complex number.

*Keywords*: derivative=0.

The Airy function  $\text{Ai}(z)$  is the solution of the Airy differential equation  $f''(z) - zf(z) = 0$  with initial conditions

$$\text{Ai}(0) = \frac{1}{3^{2/3}\Gamma\left(\frac{2}{3}\right)}; \quad \text{Ai}'(0) = \frac{1}{3^{1/3}\Gamma\left(\frac{1}{3}\right)} \quad (9.8.1)$$

Other common ways of defining the Ai-function include integrals such as

$$\text{Ai}(x) = \frac{1}{\pi} \int_0^\infty \cos\left(\frac{1}{3}t^3 + xt\right) dt, \quad x \in \mathbb{R} \quad (9.8.2)$$

$$\text{Ai}(z) = \frac{\sqrt{3}}{2\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} - \frac{z^3}{3t^3}\right) dt. \quad (9.8.3)$$

The Airy function  $\text{Ai}(x)$  can also be defined as

$$\text{Ai}(x) = \frac{1}{\pi} \sqrt{\frac{x}{3}} K_{1/3}(z), \quad (x > 0) \quad (9.8.4)$$

$$\text{Ai}(x) = \frac{1}{3^{2/3}\Gamma(2/3)}, \quad (x = 0) \quad (9.8.5)$$

$$\text{Ai}(x) = \frac{1}{2}\sqrt{-x} \left( J_{1/3}(z) - \frac{1}{\sqrt{3}} Y_{1/3}(z) \right), \quad (x < 0) \quad (9.8.6)$$

The Ai-function is an entire function with a turning point, behaving roughly like a slowly decaying sine wave for  $z < 0$  and like a rapidly decreasing exponential for  $z > 0$ . A second solution of the Airy differential equation is given by Bi(z)(see **airybi()**).

Optionally, with derivative=alpha, **airyai()** can compute the  $\alpha$ -th order fractional derivative with respect to  $z$ .

For  $\alpha = n = 1, 2, 3, \dots$  this gives the derivative  $\text{Ai}^n(z)$ , and for  $\alpha = -n = -1, -2, -3, \dots$  this gives the  $n$ -fold iterated integral

$$f_0(z) = \text{Ai}(z); \quad f_n(z) = \int_0^z f_{n-1}(t) dt. \quad (9.8.7)$$

The Ai-function has infinitely many zeros, all located along the negative half of the real axis. They can be computed with **airyaizero()**.

Limits and values include:

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> airyai(0); 1/(power(3,'2/3')*gamma('2/3'))
0.3550280538878172392600632
0.3550280538878172392600632
>>> airyai(1)
0.1352924163128814155241474
>>> airyai(-1)
0.5355608832923521187995166
>>> airyai(inf); airyai(-inf)
0.0
0.0

```

---

Evaluation is supported for large magnitudes of the argument:

---

```

>>> airyai(-100)
0.1767533932395528780908311
>>> airyai(100)
2.634482152088184489550553e-291
>>> airyai(50+50j)
(-5.31790195707456404099817e-68 - 1.163588003770709748720107e-67j)
>>> airyai(-50+50j)
(1.041242537363167632587245e+158 + 3.347525544923600321838281e+157j)
>>> airyai(10**10)
1.162235978298741779953693e-289529654602171
>>> airyai(-10**10)
0.0001736206448152818510510181
>>> w = airyai(10**10*(1+j))
>>> w.real
5.711508683721355528322567e-186339621747698
>>> w.imag
1.867245506962312577848166e-186339621747697

```

---

## 9.8.2 Airy function Bi

---

Function **airybi(z As mpNum, Keywords As String)** As mpNum

---

The function **airybi** returns the Airy function Bi

**Parameters:**

*z*: A real or complex number.

*Keywords*: derivative=0.

The Airy function Bi(*z*) is the solution of the Airy differential equation  $f''(z) - zf(z) = 0$  with initial conditions

$$\text{Bi}(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{2}{3}\right)}; \quad \text{Bi}'(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{1}{3}\right)} \quad (9.8.8)$$

The Airy function  $\text{Bi}(x)$ , can also be defined as

$$\text{Bi}(x) = \sqrt{x} \left( \frac{2}{\sqrt{3}} I_{1/3}(z) + \frac{1}{\pi} K_{1/3}(z) \right), \quad (x > 0) \quad (9.8.9)$$

$$\text{Bi}(x) = \frac{1}{3^{1/6} \Gamma(2/3)}, \quad (x = 0) \quad (9.8.10)$$

$$\text{Bi}(x) = -\frac{1}{2} \sqrt{-x} \left( \frac{1}{\sqrt{3}} J_{1/3}(z) + Y_{1/3}(z) \right), \quad (x < 0) \quad (9.8.11)$$

Like the  $\text{Ai}$ -function (see `airyai()`), the  $\text{Bi}$ -function is oscillatory for  $z < 0$ , but it grows rather than decreases for  $z > 0$ .

Optionally, as for `airyai()`, derivatives, integrals and fractional derivatives can be computed with the derivative parameter.

The  $\text{Bi}$ -function has infinitely many zeros along the negative half-axis, as well as complex zeros, which can all be computed with `airybizer()`.

Limits and values include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> airybi(0); 1/(power(3, '1/6')*gamma('2/3'))
0.6149266274460007351509224
0.6149266274460007351509224
>>> airybi(1)
1.207423594952871259436379
>>> airybi(-1)
0.10399738949694461188869
>>> airybi(inf); airybi(-inf)
+inf
0.0
```

---

Evaluation is supported for large magnitudes of the argument:

---

```
>>> airybi(-100)
0.02427388768016013160566747
>>> airybi(100)
6.041223996670201399005265e+288
>>> airybi(50+50j)
(-5.322076267321435669290334e+63 + 1.478450291165243789749427e+65j)
>>> airybi(-50+50j)
(-3.347525544923600321838281e+157 + 1.041242537363167632587245e+158j)
>>> airybi(10**10)
1.369385787943539818688433e+289529654602165
>>> airybi(-10**10)
0.001775656141692932747610973
>>> w = airybi(10**10*(1+j))
>>> w.real
-6.559955931096196875845858e+186339621747689
>>> w.imag
-6.822462726981357180929024e+186339621747690
```

---

### 9.8.3 Zeros of the Airy function Ai

---

Function **airyaizero(k As mpNum, Keywords As String)** As mpNum

---

The function `airyaizero` returns the  $k$ -th zero of the Airy Ai-function

**Parameters:**

$k$ : An integer.

**Keywords:** derivative=0.

Gives the  $k$ -th zero of the Airy Ai-function, i.e. the  $k$ -th number ordered by magnitude for which  $\text{Ai}(a_k) = 0$ .

Optionally, with derivative=1, the corresponding zero  $a'_k$  of the derivative function, i.e.  $\text{Ai}'(a'_k) = 0$ , is computed.

Examples

Some values of :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> airyaizero(1)
-2.338107410459767038489197
>>> airyaizero(2)
-4.087949444130970616636989
>>> airyaizero(3)
-5.520559828095551059129856
>>> airyaizero(1000)
-281.0315196125215528353364
```

---

### 9.8.4 Zeros of the Airy function Bi

---

Function **airybizero(k As mpNum, Keywords As String)** As mpNum

---

The function `airybizero` returns the  $k$ -th zero of the Airy Bi-function

**Parameters:**

$k$ : An integer.

**Keywords:** derivative=0, complex=0.

With complex=False, gives the  $k$ -th real zero of the Airy Bi-function, i.e. the  $k$ -th number ordered by magnitude for which  $\text{Bi}(b_k) = 0$ .

With complex=True, gives the  $k$ -th complex zero in the upper half plane  $\beta_k$ . Also the conjugate  $\bar{\beta}_k$  is a zero.

Optionally, with derivative=1, the corresponding zero  $b'_k$  or  $\beta'_k$  of the derivative function, i.e.  $\text{Bi}'(b'_k) = 0$  or  $\text{Bi}'(\beta'_k) = 0$ , is computed.

Examples

Some values of :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> airybizero(1)
-1.17371322270912792491998
>>> airybizero(2)
```

---

---

```

-3.271093302836352715680228
>>> airybizero(3)
-4.830737841662015932667709
>>> airybizero(1000)
-280.9378112034152401578834

```

---

## 9.8.5 Scorer function Gi

---

Function **scorergi(z As mpNum) As mpNum**

---

The function **scorergi** returns the Scorer function Gi

**Parameter:**

*z*: A real or complex number.

Evaluates the Scorer function

$$Gi(z) = Ai(z) \int_0^z Bi(t)dt + Bi(z) \int_z^\infty Ai(t)dt \quad (9.8.12)$$

which gives a particular solution to the inhomogeneous Airy differential equation  $f''(z) - zf(z) = 1 - \pi$ . Another particular solution is given by the Scorer Hi-function (**scorerhi()**). The two functions are related as  $Gi(z) + Hi(z) = Bi(z)$ .

Examples

Some values and limits:

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> scorergi(0); 1/(power(3, '7/6')*gamma('2/3'))
0.2049755424820002450503075
0.2049755424820002450503075
>>> diff(scorergi, 0); 1/(power(3, '5/6')*gamma('1/3'))
0.1494294524512754526382746
0.1494294524512754526382746
>>> scorergi(+inf); scorergi(-inf)
0.0
0.0
>>> scorergi(1)
0.2352184398104379375986902
>>> scorergi(-1)
-0.1166722172960152826494198

```

---

## 9.8.6 Scorer function Hi

---

Function **scorerhi(z As mpNum) As mpNum**

---

The function **scorerhi** returns the Scorer function Gi

**Parameter:**

*z*: A real or complex number.

Evaluates the second Scorer function

$$Hi(z) = Bi(z) \int_{-\infty}^z Ai(t)dt - Ai(z) \int_{-\infty}^z Bi(t)dt \quad (9.8.13)$$

which gives a particular solution to the inhomogeneous Airy differential equation  $f''(z) - zf(z) = 1 - \pi$ . See also `scorerhi()`.

Examples

Some values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> scorerhi(0); 2/(power(3,'7/6')*gamma('2/3'))
0.4099510849640004901006149
0.4099510849640004901006149
>>> diff(scorerhi,0); 2/(power(3,'5/6')*gamma('1/3'))
0.2988589049025509052765491
0.2988589049025509052765491
>>> scorerhi(+inf); scorerhi(-inf)
+inf
0.0
>>> scorerhi(1)
0.9722051551424333218376886
>>> scorerhi(-1)
0.2206696067929598945381098
```

---

## 9.9 Coulomb wave functions

### 9.9.1 Regular Coulomb wave function

---

Function **coulombf(*l* As mpNum, *eta* As mpNum, *z* As mpNum)** As mpNum

---

The function `coulombf` returns the regular Coulomb wave function

**Parameters:**

*l*: A real or complex number.

*eta*: A real or complex number.

*z*: A real or complex number.

Calculates the regular Coulomb wave function

$$F_l(\eta, z) = C_l(\eta) z^{l+1} e^{iz} {}_1F_1(l + 1 - i\eta, 2l + 2, 2iz) \quad (9.9.1)$$

where the normalization constant  $C_l(\eta)$  is as calculated by `coulombc()`. This function solves the differential equation

$$f''(z) + \left(1 - \frac{2\eta}{z} - \frac{l(l+1)}{z^2}\right) f(z) = 0. \quad (9.9.2)$$

A second linearly independent solution is given by the irregular Coulomb wave function  $G_l(\eta, z)$  (see `coulombg()`) and thus the general solution is

$$f(z) = C_1 F_l(\eta, z) + C_2 G_l(\eta, z) \quad (9.9.3)$$

for arbitrary constants  $C_1, C_2$ . Physically, the Coulomb wave functions give the radial solution to the Schrodinger equation for a point particle in a  $1/z$  potential;  $z$  is then the radius and  $l, \eta$  are quantum numbers.

The Coulomb wave functions with real parameters are defined in Abramowitz & Stegun, section 14. However, all parameters are permitted to be complex in this implementation (see references). Evaluation is supported for arbitrary magnitudes of :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> coulombf(2, 1.5, 3.5)
0.4080998961088761187426445
>>> coulombf(-2, 1.5, 3.5)
0.7103040849492536747533465
>>> coulombf(2, 1.5, '1e-10')
4.143324917492256448770769e-33
>>> coulombf(2, 1.5, 1000)
0.4482623140325567050716179
>>> coulombf(2, 1.5, 10**10)
-0.066804196437694360046619
```

---

Some test case with complex parameters, taken from Michel [2]:

---

```
>>> mp.dps = 15
>>> coulombf(1+0.1j, 50+50j, 100.156)
(-1.02107292320897e+15 - 2.83675545731519e+15j)
```

---

```
>>> coulombg(1+0.1j, 50+50j, 100.156)
(2.83675545731519e+15 - 1.02107292320897e+15j)
>>> coulombf(1e-5j, 10+1e-5j, 0.1+1e-6j)
(4.30566371247811e-14 - 9.03347835361657e-19j)
>>> coulombg(1e-5j, 10+1e-5j, 0.1+1e-6j)
(778709182061.134 + 18418936.2660553j)
```

---

## 9.9.2 Irregular Coulomb wave function

---

Function **coulombg(*l* As mpNum, *eta* As mpNum, *z* As mpNum) As mpNum**

---

The function **coulombg** returns the irregular Coulomb wave function

**Parameters:**

*l*: A real or complex number.

*eta*: A real or complex number.

*z*: A real or complex number.

Calculates the irregular Coulomb wave function

$$G_l(\eta, z) = \frac{F_l(\eta, z) \cos(\chi) - F_{-l-1}(\eta, z)}{\sin(\chi)} \quad (9.9.4)$$

where

$$\chi = \sigma_l - \sigma_{l-1} - (l + 1/2)\pi \quad (9.9.5)$$

and

$$\sigma_l(\eta) = (\ln \Gamma(1 + l + i\eta) - \ln \Gamma(1 + l - i\eta))/(2i) \quad (9.9.6)$$

See **coulombf()** for additional information.

Evaluation is supported for arbitrary magnitudes of :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> coulombg(-2, 1.5, 3.5)
1.380011900612186346255524
>>> coulombg(2, 1.5, 3.5)
1.919153700722748795245926
>>> coulombg(-2, 1.5, '1e-10')
201126715824.7329115106793
>>> coulombg(-2, 1.5, 1000)
0.1802071520691149410425512
>>> coulombg(-2, 1.5, 10**10)
0.652103020061678070929794
```

---

## 9.9.3 Normalizing Gamow constant

---

Function **coulombc(*l* As mpNum, *eta* As mpNum) As mpNum**

---

The function **coulombc** returns the normalizing Gamow constant for Coulomb wave functions

**Parameters:**

*l*: A real or complex number.

*eta*: A real or complex number.

The normalizing Gamow constant for Coulomb wave functions is defined as

$$C_l(\eta) = 2^l \exp(-\pi\eta/2 + [\ln \Gamma(1 + l + i\eta) + \ln \Gamma(1 + l - i\eta)]/2 - \ln \Gamma(2l + 2)) \quad (9.9.7)$$

where the log gamma function with continuous imaginary part away from the negative half axis (see `loggamma()`) is implied.

This function is used internally for the calculation of Coulomb wave functions, and automatically cached to make multiple evaluations with fixed *l*, *eta* fast.

## 9.10 Parabolic cylinder functions

### 9.10.1 Parabolic cylinder function D

---

Function **pcfD**(*n* As mpNum, *z* As mpNum) As mpNum

---

The function pcfD returns the parabolic cylinder function D

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

Gives the parabolic cylinder function in Whittaker's notation  $D_n(z) = U(-n-1/2, z)$  (see pcfu()). It solves the differential equation

$$y'' + \left(n + \frac{1}{2} - \frac{1}{4}z^2\right)y = 0. \quad (9.10.1)$$

and can be represented in terms of Hermite polynomials (see hermite()) as

$$D_n(z) = 2^{-n/2} e^{-z^2/4} H_n\left(\frac{z}{\sqrt{2}}\right) \quad (9.10.2)$$

Examples

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> pcfD(0,0); pcfD(1,0); pcfD(2,0); pcfD(3,0)
1.0
0.0
-1.0
0.0
>>> pcfD(4,0); pcfD(-3,0)
3.0
0.6266570686577501256039413
>>> pcfD('1/2', 2+3j)
(-5.363331161232920734849056 - 3.858877821790010714163487j)
>>> pcfD(2, -10)
1.374906442631438038871515e-9
```

---

### 9.10.2 Parabolic cylinder function U

---

Function **pcfU**(*a* As mpNum, *z* As mpNum) As mpNum

---

The function pcfU returns the parabolic cylinder function U

**Parameters:**

*a*: A real or complex number.

*z*: A real or complex number.

Gives the parabolic cylinder function  $U(a, z)$ , which may be defined for  $\Re(z) > 0$  in terms of the confluent U-function (see hyperu()) by

$$U(a, z) = 2^{-\frac{1}{4} - \frac{a}{2}} e^{-\frac{1}{4}z^2} U\left(\frac{a}{2} + \frac{1}{4}, \frac{1}{2}, \frac{1}{2}z^2\right) \quad (9.10.3)$$

or, for arbitrary  $z$ ,

$$e^{-\frac{1}{4}z^2} U(a, z) = U(a, 0) {}_1F_1\left(-\frac{a}{2} + \frac{1}{4}, \frac{1}{2}, -\frac{1}{2}z^2\right) + U'(a, 0) {}_1F_1\left(-\frac{a}{2} + \frac{3}{4}, \frac{3}{2}, -\frac{1}{2}z^2\right) \quad (9.10.4)$$

Examples

Connection to other functions:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> z = mpf(3)
>>> pcfu(0.5,z)
0.03210358129311151450551963
>>> sqrt(pi/2)*exp(z**2/4)*erfc(z/sqrt(2))
0.03210358129311151450551963
>>> pcfu(0.5,-z)
23.75012332835297233711255
>>> sqrt(pi/2)*exp(z**2/4)*erfc(-z/sqrt(2))
23.75012332835297233711255
>>> pcfu(0.5,-z)
23.75012332835297233711255
>>> sqrt(pi/2)*exp(z**2/4)*erfc(-z/sqrt(2))
23.75012332835297233711255
```

---

### 9.10.3 Parabolic cylinder function V

---

Function **pcfV**(*a* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function **pcfV** returns the parabolic cylinder function V

**Parameters:**

*a*: A real or complex number.

*z*: A real or complex number.

Gives the parabolic cylinder function  $V(a, z)$ , which can be represented in terms of **pcfU()** as

$$V(a, z) = \frac{\Gamma(a + \frac{1}{2})(U(a, -z) - \sin(\pi a)U(a, z))}{\pi} \quad (9.10.5)$$

Examples

Wronskian relation between  $U$  and  $V$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> a, z = 2, 3
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcfV(a,z)
0.7978845608028653558798921
>>> sqrt(2/pi)
0.7978845608028653558798921
```

---

---

```

>>> a, z = 2.5, 3
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcf(v(a,z)
0.7978845608028653558798921
>>> a, z = 0.25, -1
>>> pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcf(v(a,z)
0.7978845608028653558798921
>>> a, z = 2+1j, 2+3j
>>> chop(pcfu(a,z)*diff(pcfv,(a,z),(0,1))-diff(pcfu,(a,z),(0,1))*pcf(v(a,z))
0.7978845608028653558798921

```

---

## 9.10.4 Parabolic cylinder function W

---

Function **pcfW(a As mpNum, z As mpNum)** As mpNum

---

The function **pcfW** returns Computes the parabolic cylinder function W

**Parameters:**

*a*: A real or complex number.

*z*: A real or complex number.

Gives the parabolic cylinder function  $W(a, z)$  defined in (DLMF 12.14).

**Examples**

Value at the origin:

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> a = mpf(0.25)
>>> pcfw(a,0)
0.9722833245718180765617104
>>> power(2,-0.75)*sqrt(abs(gamma(0.25+0.5j*a)/gamma(0.75+0.5j*a)))
0.9722833245718180765617104
>>> diff(pcfw,(a,0),(0,1))
-0.5142533944210078966003624
>>> -power(2,-0.25)*sqrt(abs(gamma(0.75+0.5j*a)/gamma(0.25+0.5j*a)))
-0.5142533944210078966003624

```

---

# Chapter 10

## Orthogonal polynomials

An orthogonal polynomial sequence is a sequence of polynomials  $P_0(x), P_1(x), \dots$  of degree 0, 1,  $\dots$ , which are mutually orthogonal in the sense that

$$\int_S P_n(x) P_m(x) w(x) = \begin{cases} c_n \neq 0 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases} \quad (10.0.1)$$

where  $S$  is some domain (e.g. an interval  $[a, b] \in \mathbb{R}$ ) and  $w(x)$  is a fixed weight function. A sequence of orthogonal polynomials is determined completely by  $w$ ,  $S$ , and a normalization convention (e.g.  $c_n = 1$ ). Applications of orthogonal polynomials include function approximation and solution of differential equations.

Orthogonal polynomials are sometimes defined using the differential equations they satisfy (as functions of  $x$ ) or the recurrence relations they satisfy with respect to the order  $n$ . Other ways of defining orthogonal polynomials include differentiation formulas and generating functions. The standard orthogonal polynomials can also be represented as hypergeometric series (see Hypergeometric functions), more specifically using the Gauss hypergeometric function  ${}_2F_1$  in most cases. The following functions are generally implemented using hypergeometric functions since this is computationally efficient and easily generalizes.

For more information, see the Wikipedia article on orthogonal polynomials.

### 10.1 Legendre functions

#### 10.1.1 Legendre polynomial

---

Function **legendre**(*n* As mpNum, *x* As mpNum) As mpNum

---

The function `legendre` returns the Legendre polynomial  $P_n(x)$

**Parameters:**

*n*: A real or complex number.

*x*: A real or complex number.

The Legendre polynomials are given by the formula

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n \quad (10.1.1)$$

Alternatively, they can be computed recursively using

$$P_0(x) = 1; \quad P_1(x) = x; \quad (n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \quad (10.1.2)$$

A third definition is in terms of the hypergeometric function  ${}_2F_1$ , whereby they can be generalized to arbitrary  $n$ :

$$P_n(x) = {}_2F_1\left(-n, n+1, 1, \frac{1-x}{2}\right) \quad (10.1.3)$$

The Legendre polynomials assume fixed values at the points  $x = -1$  and  $x = 1$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint([legendre(n, 1) for n in range(6)])
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> nprint([legendre(n, -1) for n in range(6)])
[1.0, -1.0, 1.0, -1.0, 1.0, -1.0]
```

---

### 10.1.2 Associated Legendre function of the first kind

---

Function **legenp**(*n* As *mpNum*, *m* As *mpNum*, *z* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function **legenp** returns the (associated) Legendre function of the first kind of degree  $n$  and order  $m$ ,  $P_n^m(z)$ .

**Parameters:**

*n*: A real or complex number.

*m*: A real or complex number.

*z*: A real or complex number.

**Keywords:** type=2.

Calculates the (associated) Legendre function of the first kind of degree  $n$  and order  $m$ ,  $P_n^m(z)$ . Taking  $m = 0$  gives the ordinary Legendre function of the first kind,  $P_n(z)$ . The parameters may be complex numbers.

In terms of the Gauss hypergeometric function, the (associated) Legendre function is defined as

$$P_n^m = \frac{1}{\Gamma(1-m)} \frac{(1+z)^{m/2}}{(1-z)^{m/2}} {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right). \quad (10.1.4)$$

With type=3 instead of type=2, the alternative definition

$$\hat{P}_n^m = \frac{1}{\Gamma(1-m)} \frac{(1+z)^{m/2}}{(z-1)^{m/2}} {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right). \quad (10.1.5)$$

is used. These functions correspond respectively to `LegendreP[n,m,2,z]` and `LegendreP[n,m,3,z]` in Mathematica.

The general solution of the (associated) Legendre differential equation

$$(1-z^2)f''(z) - 2zf'(z) + \left(n(n+1) - \frac{m^2}{1-z^2}\right)f(z) = 0 \quad (10.1.6)$$

is given by  $C_1 P_n^m(z) + C_2 Q_n^m(z)$  for arbitrary constants  $C_1, C_2$ , where  $Q_n^m(z)$  is a Legendre function of the second kind as implemented by `legenq()`.

## Examples

Evaluation for arbitrary parameters and arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> legenp(2, 0, 10); legendre(2, 10)
149.5
149.5
>>> legenp(-2, 0.5, 2.5)
(1.972260393822275434196053 - 1.972260393822275434196053j)
>>> legenp(2+3j, 1-j, -0.5+4j)
(-3.335677248386698208736542 - 5.663270217461022307645625j)
>>> chop(legenp(3, 2, -1.5, type=2))
28.125
>>> chop(legenp(3, 2, -1.5, type=3))
-28.125
```

---

### 10.1.3 Associated Legendre function of the second kind

---

Function **legenq**(*n* As *mpNum*, *m* As *mpNum*, *z* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function `legenq` returns the (associated) Legendre function of the second kind of degree *n* and order *m*,  $Q_n^m(z)$ .

**Parameters:**

*n*: A real or complex number.

*m*: A real or complex number.

*z*: A real or complex number.

*Keywords*: type=2.

Calculates the (associated) Legendre function of the second kind of degree *n* and order *m*,  $Q_n^m(z)$ . Taking *m* = 0 gives the ordinary Legendre function of the second kind,  $Q_n(z)$ . The parameters may complex numbers.

The Legendre functions of the second kind give a second set of solutions to the (associated) Legendre differential equation. (See `legenp()`.) Unlike the Legendre functions of the first kind, they are not polynomials of *z* for integer *n, m* but rational or logarithmic functions with poles at *z* = ±1.

There are various ways to define Legendre functions of the second kind, giving rise to different complex structure. A version can be selected using the `type` keyword argument. The `type=2` and `type=3` functions are given respectively by

$$Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} \left( \cos(\pi m) P_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} P_n^{-m}(z) \right) \quad (10.1.7)$$

$$\hat{Q}_n^m(z) = \frac{\pi}{2 \sin(\pi m)} e^{\pi i m} \left( \hat{P}_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} \hat{P}_n^{-m}(z) \right) \quad (10.1.8)$$

where  $P$  and  $\hat{P}$  are the `type=2` and `type=3` Legendre functions of the first kind. The formulas above should be understood as limits when *m* is an integer.

These functions correspond to `LegendreQ[n,m,2,z]` (or `LegendreQ[n,m,z]`) and `LegendreQ[n,m,3,z]` in Mathematica. The `type=3` function is essentially the same as the function defined in Abramowitz & Stegun (eq. 8.1.3) but with  $(z+1)^{m/2}(z-1)^{m/2}$  instead of  $(z^2-1)^{m/2}$ , giving slightly different branches.

Examples

Evaluation for arbitrary parameters and arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> legenq(2, 0, 0.5)
-0.8186632680417568557122028
>>> legenq(-1.5, -2, 2.5)
(0.6655964618250228714288277 + 0.3937692045497259717762649j)
>>> legenq(2-j, 3+4j, -6+5j)
(-10001.95256487468541686564 - 6011.691337610097577791134j)
```

---

### 10.1.4 Spherical harmonics

---

Function **spherharm**(*l* As *mpNum*, *m* As *mpNum*, *theta* As *mpNum*, *phi* As *mpNum*) As *mpNum*

---

The function `spherharm` returns the spherical harmonic  $Y_l^m(\theta, \phi)$

**Parameters:**

*l*: A real or complex number.

*m*: A real or complex number.

*theta*: A real or complex number.

*phi*: A real or complex number.

Evaluates the spherical harmonic  $Y_l^m(\theta, \phi)$ ,

$$Y_l^m(\theta, \phi) = \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}} P_l^m(\cos(\theta)) e^{im\phi} \quad (10.1.9)$$

where  $P_l^m$  is an associated Legendre function (see `legenp()`).

Here  $\theta \in [0, \pi]$  denotes the polar coordinate (ranging from the north pole to the south pole) and  $\phi \in [0, 2\pi]$  denotes the azimuthal coordinate on a sphere. Care should be used since many different conventions for spherical coordinate variables are used.

Usually spherical harmonics are considered for  $l \in \mathbb{N}$ ,  $m \in \mathbb{Z}$ ,  $|m| \leq l$ . More generally,  $l, m, \theta, \phi$  are permitted to be complex numbers.

Some low-order spherical harmonics with reference values:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> theta = pi/4
>>> phi = pi/3
>>> spherharm(0,0,theta,phi); 0.5*sqrt(1/pi)*expj(0)
(0.2820947917738781434740397 + 0.0j)
(0.2820947917738781434740397 + 0.0j)
>>> spherharm(1,-1,theta,phi); 0.5*sqrt(3/(2*pi))*expj(-phi)*sin(theta)
(0.1221506279757299803965962 - 0.2115710938304086076055298j)
```

```
(0.1221506279757299803965962 - 0.2115710938304086076055298j)
>>> spherharm(1,0,theta,phi); 0.5*sqrt(3/pi)*cos(theta)*expj(0)
(0.3454941494713354792652446 + 0.0j)
(0.3454941494713354792652446 + 0.0j)
>>> spherharm(1,1,theta,phi); -0.5*sqrt(3/(2*pi))*expj(phi)*sin(theta)
(-0.1221506279757299803965962 - 0.2115710938304086076055298j)
(-0.1221506279757299803965962 - 0.2115710938304086076055298j)
```

---

## 10.2 Chebyshev polynomials

### 10.2.1 Chebyshev polynomial of the first kind

---

Function **chebyt(*n* As mpNum, *x* As mpNum) As mpNum**

---

The function `chebyt` returns the Chebyshev polynomial of the first kind  $T_n(x)$

**Parameters:**

*n*: A real or complex number.

*x*: A real or complex number.

The Chebyshev polynomial of the first kind  $T_n(x)$  are defined by the identity

$$T_n(\cos(x)) = \cos(nx). \quad (10.2.1)$$

The  $T_n(x)$  are orthogonal on the interval  $(-1, 1)$ , with respect to the weight function  $w(x) = (1 - x^2)^{-1/2}$ .

For  $0 \leq n \leq 64$  the function evaluates  $T_n(x)$  with the standard recurrence formulas [1, 22.7.4]:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x). \end{aligned} \quad (10.2.2)$$

If  $n > 64$  the following trigonometric and hyperbolic identities [1, 22.3.15]:

$$T_n(x) = \cos(n \arccos(x)), \quad |x| < 1 \quad (10.2.3)$$

$$T_n(x) = \cosh(n \operatorname{arccosh}(x)), \quad |x| > 1 \quad (10.2.4)$$

are used, and the special cases  $|x| = 1$  are handled separately. If  $n < 0$  the function result is  $T_n(x) = T_{-n}(x)$ .

The Chebyshev polynomials of the first kind are a special case of the Jacobi polynomials, and by extension of the hypergeometric function  ${}_2F_1$ . They can thus also be evaluated for nonintegral  $n$ . The Chebyshev polynomials of the first kind are orthogonal on the interval  $[-1, 1]$  with respect to the weight function  $w(x) = 1/\sqrt{1 - x^2}$ :

---

```
>>> f = lambda x: chebyt(m,x)*chebyt(n,x)/sqrt(1-x**2)
>>> m, n = 3, 4
>>> nprint(quad(f, [-1, 1]),1)
0.0
>>> m, n = 4, 4
>>> quad(f, [-1, 1])
1.57079632596448
```

---

### 10.2.2 Chebyshev polynomial of the second kind

---

Function **chebyu(*n* As mpNum, *x* As mpNum) As mpNum**

---

The function `chebyu` returns the Chebyshev polynomial of the second kind  $U_n(x)$

**Parameters:**

$n$ : A real or complex number.

$x$ : A real or complex number.

The Chebyshev polynomial of the second kind  $U_n(x)$  are defined by the identity

$$U_n(\cos(x)) = \frac{\sin((n+1)x)}{\sin(x)} \quad (10.2.5)$$

The  $U_n(x)$  are orthogonal on the interval  $(-1, 1)$ , with respect to the weight function  $w(x) = (1 - x^2)^{1/2}$ .

For  $0 \leq n \leq 64$  the function evaluates  $U_n(x)$  with the standard recurrence formulas [1, 22.7.4]:

$$\begin{aligned} U_0(x) &= 1 \\ U_1(x) &= 2x \\ U_{n+1}(x) &= 2xU_n(x) - U_{n-1}(x). \end{aligned} \quad (10.2.6)$$

If  $n > 64$  the following trigonometric and hyperbolic identities [1, 22.3.15]:

$$U_n(x) = \frac{\sin((n+1) \arccos(x))}{\sin(\arccos(x))}, \quad |x| < 1 \quad (10.2.7)$$

$$U_n(x) = \frac{\sin((n+1) \operatorname{arccosh}(x))}{\sin(\operatorname{arccosh}(x))}, \quad |x| > 1 \quad (10.2.8)$$

are used, and the special cases  $|x| = 1$  are handled separately. If  $n < 0$  the function result are  $U_{-1}(x) = 0$  and  $U_n(x) = -U_{-n-2}(x)$ .

The Chebyshev polynomials of the second kind are a special case of the Jacobi polynomials, and by extension of the hypergeometric function  ${}_2F_1$ . They can thus also be evaluated for nonintegral  $n$ .

The Chebyshev polynomials of the first kind are orthogonal on the interval  $[-1, 1]$  with respect to the weight function  $w(x) = \sqrt{1 - x^2}$ :

---

```
>>> f = lambda x: chebyu(m, x)*chebyu(n, x)*sqrt(1-x**2)
>>> m, n = 3, 4
>>> quad(f, [-1, 1])
0.0
>>> m, n = 4, 4
>>> quad(f, [-1, 1])
1.5707963267949
```

---

## 10.3 Jacobi and Gegenbauer polynomials

### 10.3.1 Jacobi polynomial

---

Function **jacobi(*n* As mpNum, *a* As mpNum, *b* As mpNum, *z* As mpNum)** As mpNum

---

The function **jacobi** returns the Jacobi polynomial  $P_n^{(a,b)}$

**Parameters:**

*n*: A real or complex number.

*a*: A real or complex number.

*b*: A real or complex number.

*z*: A real or complex number.

These functions return  $P_n^{(a,b)}(x)$ , the Jacobi polynomial of degree  $n \geq 0$  with parameters  $(a, b)$ .  $a, b$  should be greater than  $-1$ , and  $a + b$  must not be an integer less than  $-1$ . Jacobi polynomials are orthogonal on the interval  $(-1, 1)$ , with respect to the weight function  $w(x) = (1-x)^a(1+x)^b$ , if  $a, b$  are greater than  $-1$ . The cases  $n \leq 1$  are computed with the explicit formulas

$$P_0^{(a,b)} = 1, \quad 2P_1^{(a,b)} = (a - b) + (a + b + 2)x, \quad (10.3.1)$$

and for  $n > 1$  there are the somewhat complicated recurrence relations from [30] (18.9.1) and (18.9.2):

$$\begin{aligned} P_{n+1}^{(a,b)} &= (A_n x + B_n) P_n^{(a,b)} - C_n P_{n+1}^{(a,b)} & (10.3.2) \\ A_n &= \frac{(2n + a + b + 1)(2n + a + b + 2)}{2(n + 1)(n + a + b + 1)} \\ B_n &= \frac{(a^2 - b^2)(2n + a + b + 1)}{2(n + 1)(n + a + b + 1)(2n + a + b)} \\ C_n &= \frac{(n + a)(n + b)(2n + a + b + 2)}{(n + 1)(n + a + b + 1)(2n + a + b)}. \end{aligned}$$

**jacobi(*n*, *a*, *b*, *x*)** evaluates the Jacobi polynomial  $P_n^{(a,b)}$ . The Jacobi polynomials are a special case of the hypergeometric function  ${}_2F_1$  given by:

$$P_n^{(a,b)} = \binom{n+a}{n} {}_2F_1 \left( -n, 1 + a + b + n, a + 1, \frac{1-x}{2} \right). \quad (10.3.3)$$

Note that this definition generalizes to nonintegral values of  $n$ . When  $n$  is an integer, the hypergeometric series terminates after a finite number of terms, giving a polynomial in  $x$ .

Evaluation of Jacobi polynomials

A special evaluation is  $P_n^{(a,b)} = \binom{n+a}{n}$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> jacobi(4, 0.5, 0.25, 1)
2.4609375
>>> binomial(4+0.5, 4)
2.4609375
```

---

### 10.3.2 Zernike Radial Polynomials

---

Function **ZernikeRadialMpMath(*n* As mpNum, *m* As mpNum, *x* As mpNum) As mpNum**

---

**NOT YET IMPLEMENTED**

---

The function **ZernikeRadialMpMath** returns the Zernike radial polynomials  $R_n^m(r)$ .

**Parameters:**

*n*: An Integer.

*m*: An Integer.

*x*: A real number.

This function returns the Zernike radial polynomials  $R_n^m(r)$  with  $r \geq 0$  and  $n \geq m \geq 0$ ,  $n - m$  even, and zero otherwise. The  $R_n^m(r)$  are special cases of the Jacobi Polynomials

$$R_n^m(r) = (-1)^{(n-m)/2} r^m P_{(n-m)/2}^{(m,0)}(1 - 2r^2). \quad (10.3.4)$$

### 10.3.3 Gegenbauer polynomial

---

Function **gegenbauer(*n* As mpNum, *a* As mpNum, *z* As mpNum) As mpNum**

---

The function **gegenbauer** returns the Gegenbauer polynomial  $C_n^{(a)}(z)$

**Parameters:**

*n*: A real or complex number.

*a*: A real or complex number.

*z*: A real or complex number.

Evaluates the Gegenbauer polynomial, or ultraspherical polynomial,

$$C_n^{(a)}(z) = \binom{n+2a-1}{n} {}_2F_1 \left( -n, n+2a; a + \frac{1}{2}, \frac{1}{2}(1-z) \right). \quad (10.3.5)$$

When *n* is a nonnegative integer, this formula gives a polynomial in *z* of degree *n*, but all parameters are permitted to be complex numbers. With *a* = 1/2, the Gegenbauer polynomial reduces to a Legendre polynomial.

These functions return  $C_n^{(a)}(x)$ , the Gegenbauer (ultraspherical) polynomial of degree *n* with parameter *a*. The degree *n* must be non-negative; *a* should be  $> -1/2$ . The Gegenbauer polynomials are orthogonal on the interval  $(-1, 1)$ , with respect to the weight function  $w(x) = (1 - x^2)^{a-1/2}$ . If *a*  $\neq 0$  the function uses the standard recurrence formulas [1, 22.7.3]:

$$\begin{aligned} C_0^{(a)}(x) &= 1 \\ C_1^{(a)}(x) &= 2ax \\ nC_n^{(a)}(x) &= 2(n+a-1)x C_{n-1}^{(a)}(x) - (n+2a-2) C_{n-2}^{(a)}(x). \end{aligned} \quad (10.3.6)$$

For *a* = 0 the result can be expressed in Chebyshev polynomials:

$$C_0^{(0)}(x) = 1, \quad C_n^{(0)}(x) = 2/n T_n(x). \quad (10.3.7)$$

Evaluation for arbitrary arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> gegenbauer(3, 0.5, -10)
-2485.0
>>> gegenbauer(1000, 10, 100)
3.012757178975667428359374e+2322
>>> gegenbauer(2+3j, -0.75, -1000j)
(-5038991.358609026523401901 + 9414549.285447104177860806j)
```

---

## 10.4 Hermite and Laguerre polynomials

### 10.4.1 Hermite polynomials

---

Function **hermite(*n* As mpNum, *z* As mpNum) As mpNum**

---

The function `hermite` returns the Hermite polynomial  $H_n(z)$

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

Evaluates the Hermite polynomial  $H_n(z)$ , which may be defined using the recurrence

$$H_0(z) = 1; \quad H_1(z) = 2z; \quad H_{n+1} = 2zH_n(z) - 2nH_{n-1}(z). \quad (10.4.1)$$

The  $H_n$  are orthogonal on the interval  $(-\infty, \infty)$ , with respect to the weight function  $w(x) = e^{-x^2}$ . They are computed with the standard recurrence formulas [1, 22.7.13]:

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= 2x \\ H_n(x) &= 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x). \end{aligned} \quad (10.4.2)$$

The Hermite polynomials are orthogonal on  $(-\infty, \infty)$  with respect to the weight  $e^{-z^2}$ . More generally, allowing arbitrary complex values of  $n$ , the Hermite function  $H_n(z)$  is defined as

$$H_n(z) = (2z)^n {}_2F_0\left(-\frac{n}{2}, \frac{1-n}{2}, -\frac{1}{z^2}\right) \quad (10.4.3)$$

for  $\Re z > 0$ , or generally

$$H_n(z) = 2^n \sqrt{\pi} \left( \frac{1}{\Gamma(\frac{1-n}{2})} {}_1F_1\left(-\frac{n}{2}, \frac{1}{2}, z^2\right) - \frac{2z}{\Gamma(-\frac{n}{2})} {}_1F_1\left(-\frac{1-n}{2}, \frac{3}{2}, z^2\right) \right) \quad (10.4.4)$$

Evaluation for arbitrary arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hermite(0, 10)
1.0
>>> hermite(1, 10); hermite(2, 10)
20.0
398.0
>>> hermite(10000, 2)
4.950440066552087387515653e+19334
>>> hermite(3, -10**8)
-7999999999999998800000000.0
>>> hermite(-3, -10**8)
1.675159751729877682920301e+4342944819032534
>>> hermite(2+3j, -1+2j)
(-0.07652130602993513389421901 - 0.1084662449961914580276007j)
```

---

### 10.4.2 Laguerre polynomials

---

Function **laguerre(*n* As mpNum, *a* As mpNum, *z* As mpNum) As mpNum**

---

The function `laguerre` returns the generalized Laguerre polynomial  $L_n^{\alpha}(z)$

**Parameters:**

*n*: A real or complex number.

*a*: A real or complex number.

*z*: A real or complex number.

These functions return  $L_n^{(a)}(x)$ , the generalized Laguerre polynomials of degree  $n \geq 0$  with parameter  $a$ ;  $x \geq 0$  and  $a > -1$  are the standard ranges. These polynomials are orthogonal on the interval  $(0, \infty)$ , with respect to the weight function  $w(x) = e^{-x}x^a$ .

If  $x < 0$  and  $a > -1$  the function tries to avoid inaccuracies and computes the result with KummerâŽs confluent hypergeometric function, see Abramowitz and Stegun[1, 22.5.34]

$$L_n^{(a)}(x) = \binom{n+a}{n} M(-n, a+1, x), \quad (10.4.5)$$

otherwise the standard recurrence formulas are used:

$$\begin{aligned} L_0^{(a)}(x) &= 1 \\ L_1^{(a)}(x) &= -x + 1 + a \\ nL_n^{(a)}(x) &= (2n + a - 1 - x)L_{n-1}^{(a)}(x) - (n + a - 1)L_{n-2}^{(a)}(x). \end{aligned} \quad (10.4.6)$$

Gives the generalized (associated) Laguerre polynomial, defined by

$$L_n^{\alpha}(z) = \frac{\Gamma(n + b + 1)}{\Gamma(b + 1)\Gamma(n + 1)} {}_1F_1(-n, a + 1, z). \quad (10.4.7)$$

With  $a = 0$  and  $n$  a nonnegative integer, this reduces to an ordinary Laguerre polynomial, the sequence of which begins

$$L_0(z) = 1, \quad L_1(z) = 1 - z, \quad L_2(z) = z^2 - 2z + 1, \dots \quad (10.4.8)$$

The Laguerre polynomials are orthogonal with respect to the weight  $z^a e^{-z}$  on  $[0, \infty)$ .

Evaluation for arbitrary arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> laguerre(5, 0, 0.25)
0.037263997395833333333333333
>>> laguerre(1+j, 0.5, 2+3j)
(4.474921610704496808379097 - 11.02058050372068958069241j)
>>> laguerre(2, 0, 10000)
49980001.0
>>> laguerre(2.5, 0, 10000)
-9.327764910194842158583189e+4328
```

---

### 10.4.3 Laguerre Polynomials

---

Function **LaguerreLMpMath(*n* As mpNum, *x* As mpNum) As mpNum**

---

**NOT YET IMPLEMENTED**

---

The function `LaguerreLMpMath` returns  $L_n(x)$ , the Laguerre polynomials of degree  $n \geq 0$ .

**Parameters:**

*n*: An Integer.

*x*: A real number.

This function returns  $L_n(x)$ , the Laguerre polynomials of degree  $n \geq 0$ . The Laguerre polynomials are just special cases of the generalized Laguerre polynomials

$$L_n(x) = L_n^{(0)}(x). \quad (10.4.9)$$

### 10.4.4 Associated Laguerre Polynomials

---

Function **AssociatedLaguerreMpMath(*n* As mpNum, *m* As mpNum, *x* As mpNum) As mpNum**

---

**NOT YET IMPLEMENTED**

---

The function `AssociatedLaguerreMpMath` returns  $L_n^m(x)$ , the associated Laguerre polynomials of degree  $n \geq 0$  and order  $m \geq 0$ .

**Parameters:**

*n*: An Integer.

*m*: An Integer.

*x*: A real number.

This function returns  $L_n^m(x)$ , the associated Laguerre polynomials of degree  $n \geq 0$  and order  $m \geq 0$ , defined as

$$L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x). \quad (10.4.10)$$

The  $L_n^m(x)$  are computed using the generalized Laguerre polynomials

$$L_n^m(x) = L_n^{(m)}(x). \quad (10.4.11)$$

# Chapter 11

## Hypergeometric functions

The functions listed in Exponential integrals and error functions, Bessel functions and related functions and Orthogonal polynomials, and many other functions as well, are merely particular instances of the generalized hypergeometric function  ${}_pF_q$ . The functions listed in the following section enable efficient direct evaluation of the underlying hypergeometric series, as well as linear combinations, limits with respect to parameters, and analytic continuations thereof. Extensions to twodimensional series are also provided. See also the basic or q-analog of the hypergeometric series in q-functions.

For convenience, most of the hypergeometric series of low order are provided as standalone functions. They can equivalently be evaluated using `hyper()`. As will be demonstrated in the respective docstrings, all the `hyp#f#` functions implement analytic continuations and/or asymptotic expansions with respect to the argument  $z$ , thereby permitting evaluation for anywhere in the complex plane. Functions of higher degree can be computed via `hyper()`, but generally only in rapidly convergent instances.

Most hypergeometric and hypergeometric-derived functions accept optional keyword arguments to specify options for `hypercomb()` or `hyper()`. Some useful options are `maxprec`, `maxterms`, `zeroprec`, `accurate_small`, `hmag`, `force_series`, `asymp_tol` and `eliminate`. These options give control over what to do in case of slow convergence, extreme loss of accuracy or evaluation at zeros (these two cases cannot generally be distinguished from each other automatically), and singular parameter combinations.

For alternative implementations, see e.g. [Pearson \(2009\)](#), [Muller \(2001\)](#) or [Forrey \(1997\)](#).

## 11.1 Confluent Hypergeometric Limit Function

### 11.1.1 Confluent Hypergeometric Limit Function

---

Function **hyp0f1(a As mpNum, z As mpNum)** As mpNum

---

The function `hyp0f1` returns the hypergeometric function  ${}_0F_1$

**Parameters:**

*a*: A real or complex number.

*z*: A real or complex number.

Gives the hypergeometric function  ${}_0F_1$ , sometimes known as the confluent limit function, defined as

$${}_0F_1(a, z) = \sum_{k=0}^{\infty} \frac{1}{(a)_k} \frac{z^k}{k!}. \quad (11.1.1)$$

This function satisfies the differential equation  $zf''(z) + af'(z) = f(z)$ , and is related to the Bessel function of the first kind (see `besselj()`).

This function returns the confluent hypergeometric limit function  ${}_0F_1(b; x)$ , defined for  $b \neq 0, -1, -2, -3, \dots$ , by the series

$${}_0F_1(b; x) = {}_0F_1(-; b; x) = \sum_{n=0}^{\infty} \frac{x^n}{(b)_n n!} \quad (11.1.2)$$

where  $(a)_k$  is the Pochammer symbol (see section 7.3)

The function is calculated by treating  ${}_0F_1(b; 0) = 1$  as special case, and otherwise using the following relations to Bessel functions:

$${}_0F_1(b; x) = \Gamma(b) (+x)^{(1-b)/2} I_{b-1} (2\sqrt{+x}), \quad x > 0, \quad (11.1.3)$$

$${}_0F_1(b; x) = \Gamma(b) (-x)^{(1-b)/2} J_{b-1} (2\sqrt{-x}), \quad x < 0, \quad (11.1.4)$$

`hyp0f1(a,z)` is equivalent to `hyper([], [a], z)`; see documentation for `hyper()` for more information.

Examples

Evaluation for arbitrary arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp0f1(2, 0.25)
1.130318207984970054415392
>>> hyp0f1((1,2), 1234567)
6.27287187546220705604627e+964
>>> hyp0f1(3+4j, 1000000j)
(3.905169561300910030267132e+606 + 3.807708544441684513934213e+606j)
```

---

Evaluation is supported for arbitrarily large values of *z*, using asymptotic expansions:

---

```
>>> hyp0f1(1, 10**50)
2.131705322874965310390701e+8685889638065036553022565
>>> hyp0f1(1, -10**50)
1.115945364792025420300208e-13
```

---

Verifying the differential equation:

---

```
>>> a = 2.5
>>> f = lambda z: hyp0f1(a,z)
>>> for z in [0, 10, 3+4j]:
... chop(z*diff(f,z,2) + a*diff(f,z) - f(z))
...
0.0
0.0
0.0
```

---

### 11.1.2 Regularized Confluent Hypergeometric Limit Function

---

Function **Hypergeometric0F1RegularizedMpMath**(*b* As mpNum, *x* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **Hypergeometric0F1RegularizedMpMath** returns the regularized confluent hypergeometric limit function  ${}_0\tilde{F}_1(b; x)$ .

**Parameters:**

*b*: A real number.

*x*: A real number.

The regularized confluent hypergeometric limit function  ${}_0\tilde{F}_1(b; x)$  for unrestricted *b* is defined by [30, 15.1.2]

$${}_0\tilde{F}_1(b; x) = \frac{1}{\Gamma(b)} {}_0F_1(b; x), \quad (b \neq 0, -1, -2, \dots) \quad (11.1.5)$$

and by the corresponding limit if  $b = 0, -1, -2, \dots, = -n$ , with the value

$${}_0\tilde{F}_1(-n; x) = x^{n+1} {}_0F_1(n+2; x), \quad n \in \mathbb{N} \quad (11.1.6)$$

where  $\Gamma(\cdot)$  is the Gamma function (see section ??).

## 11.2 Kummer's Confluent Hypergeometric Functions and related functions

### 11.2.1 Kummer's Confluent Hypergeometric Function of the first kind

---

Function **hyp1f1(a As mpNum, b As mpNum, z As mpNum) As mpNum**

---

The function `hyp1f1` returns the confluent hypergeometric function of the first kind  ${}_1F_1(a, b; z)$

**Parameters:**

*a*: A real or complex number.

*b*: A real or complex number.

*z*: A real or complex number.

The confluent hypergeometric function of the first kind is defined as

$${}_1F_1(a, b; z) = \sum_{k=0}^{\infty} \frac{(a)_k}{(b)_k} \cdot \frac{z^k}{k!} \quad (11.2.1)$$

also known as Kummer's function and sometimes denoted by  $M(a, b; z)$ . This function gives one solution to the confluent (Kummer's) differential equation

$$zf''(z) + (b - z)f'(z) - af(z) = 0. \quad (11.2.2)$$

A second solution is given by the *U* function; see `hyperu()`. Solutions are also given in an alternate form by the Whittaker functions (`whitm()`, `whitw()`).

This function returns the Kummer's confluent hypergeometric function  ${}_1F_1(a, b; x)$ , defined by the series

$${}_1F_1(a, b; z) = M(a, b; z) = \sum_{n=0}^{\infty} \frac{(a)_n}{(b)_n} \cdot \frac{z^n}{n!} \quad (11.2.3)$$

where  $(a)_k$  is the Pochammer symbol (see section 7.3)

The function has the following integral representation

$${}_1F_1(a, b; z) = B(a, b - a)^{-1} \int_0^1 e^{zt} t^{a-1} (1 - t)^{b-a-1} dt, \quad \Re b > \Re a > 0 \quad (11.2.4)$$

The following closed-form approximation based on a Laplace approximation has been derived by [Butler & Wood \(2002\)](#):

$${}_1F_1(a, b; z) \approx \frac{G_1(a, b; z)}{G_1(a, b; 0)}, \quad \text{where} \quad (11.2.5)$$

$$G_1(a, b, c; z) = w^{-1/2} y^a (1 - y)^{b-a} e^x y,$$

$$w = a(1 - y)^2 + (b - a)y^2$$

$$y = [(x - b) + \sqrt{(x - b)^2 + 4ax}]/2x, \text{ if } x \neq 0, \text{ and } y = a/b \text{ otherwise.}$$

`hyp1f1(a,b,z)` is equivalent to `hyper([a],[b],z)`; see documentation for `hyper()` for more information. Parameters may be complex:

---

```
>>> hyp1f1(2+3j, -1+j, 10j)
(261.8977905181045142673351 + 160.8930312845682213562172j)
```

---

Arbitrarily large values of are supported:

---

```
>>> hyp1f1(3, 4, 10**20)
3.890569218254486878220752e+43429448190325182745
>>> hyp1f1(3, 4, -10**20)
6.0e-60
>>> hyp1f1(3, 4, 10**20*j)
(-1.935753855797342532571597e-20 - 2.291911213325184901239155e-20j)
```

---

Verifying the differential equation:

---

```
>>> a, b = 1.5, 2
>>> f = lambda z: hyp1f1(a,b,z)
>>> for z in [0, -10, 3, 3+4j]:
...   chop(z*diff(f,z,2) + (b-z)*diff(f,z) - a*f(z))
...
0.0
0.0
0.0
0.0
```

---

## 11.2.2 Kummer's Regularized Confluent Hypergeometric Function

---

Function **Hypergeometric1F1RegularizedMpMath**(*a* As mpNum, *b* As mpNum, *z* As mpNum)

---

**NOT YET IMPLEMENTED**

---

The function **Hypergeometric1F1RegularizedMpMath** returns Kummer's regularized confluent hypergeometric function  ${}_1F_1(a; b; z)$ .

**Parameters:**

*a*: A real number.

*b*: A real number.

*z*: A real number.

This function returns the Kummer's regularized confluent hypergeometric function  ${}_1\tilde{F}_1(a; b; z)$  for unrestricted *b*, defined by [30, 15.1.2]

$${}_1\tilde{F}_1(a; b; z) = \frac{1}{\Gamma(b)} {}_1F_1(a; b; z) = M(a, b; z) = \frac{1}{\Gamma(b)} M(a; b; z), \quad (b \neq 0, -1, -2, \dots) \quad (11.2.6)$$

and by the corresponding limit if  $b = 0, -1, -2, \dots, = -n$ , with the value

$${}_1\tilde{F}_1(a; b; z) = \frac{(a)_{n+1}}{(n+1)!} x^{n+1} {}_1F_1(a+n+1; n+2; z), \quad n \in \mathbb{N} \quad (11.2.7)$$

where  $\Gamma(\cdot)$  is the Gamma function (see section ??) and  $(a)_k$  is the Pochammer symbol (see section 7.3)

The function has the following integral representation

$${}_1\tilde{F}_1(a, b; -m; z) = \frac{1}{\Gamma(a)\Gamma(b-a)} \int_0^1 e^{zt} t^{a-1} (1-t)^{b-a-1} dt, \quad \Re b > \Re a > 0 \quad (11.2.8)$$

### 11.2.3 Kummer's Confluent Hypergeometric Function of the second kind

---

Function **hyperu**(*a* As mpNum, *b* As mpNum, *z* As mpNum) As mpNum

---

The function `hyperu` returns the Tricomi confluent hypergeometric function  $U$

**Parameters:**

- a*: A real or complex number.
- b*: A real or complex number.
- z*: A real or complex number.

The Kummer or confluent hypergeometric function of the second kind is also known as the Tricomi confluent hypergeometric function,  $U$ . This function gives a second linearly independent solution to the confluent hypergeometric differential equation (the first is provided by  ${}_1F_1$  - see `hyp1f1()`).

This function returns the Tricomi's confluent hypergeometric function  $U(a, b; x)$  for  $x > 0$  and  $b \neq 0, \pm 1 \pm 2, \dots$ , defined by

$$U(a, b; x) = \frac{\Gamma(1-b)}{\Gamma(1+a-b)} M(a, b; c; z) + \frac{\Gamma(1-b)}{\Gamma(a)} x^{1-b} M(1+a-b, 2-b; x) \quad (11.2.9)$$

where  $\Gamma(\cdot)$  is the Gamma function (see section ??).

Examples

Evaluation for arbitrary complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hyperu(2,3,4)
0.0625
>>> hyperu(0.25, 5, 1000)
0.1779949416140579573763523
>>> hyperu(0.25, 5, -1000)
(0.1256256609322773150118907 - 0.1256256609322773150118907j)
```

---

### 11.2.4 Hypergeometric Function 2F0

---

Function **hyp2f0**(*a* As mpNum, *b* As mpNum, *z* As mpNum) As mpNum

---

The function `hyp2f0` returns the hypergeometric function  ${}_2F_0$

**Parameters:**

- a*: A real or complex number.
- b*: A real or complex number.
- z*: A real or complex number.

The hypergeometric function  ${}_2F_0$  is defined formally by the series

$$_2F_0(a, b; z) = \sum_{n=0}^{\infty} (a)_n (b)_n \frac{z^n}{n!} \quad (11.2.10)$$

This series usually does not converge. For small enough  $z$ , it can be viewed as an asymptotic series that may be summed directly with an appropriate truncation. When this is not the case, `hyp2f0()` gives a regularized sum, or equivalently, it uses a representation in terms of the hypergeometric U function [1]. The series also converges when either  $a$  or  $b$  is a nonpositive integer, as it then terminates into a polynomial after  $-a$  or  $-b$  terms.

Examples

Evaluation is supported for arbitrary complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp2f0((2,3), 1.25, -100)
0.07095851870980052763312791
>>> hyp2f0((2,3), 1.25, 100)
(-0.03254379032170590665041131 + 0.07269254613282301012735797j)
>>> hyp2f0(-0.75, 1-j, 4j)
(-0.3579987031082732264862155 - 3.052951783922142735255881j)
```

---

Even with real arguments, the regularized value of  $2F0$  is often complex-valued, but the imaginary part decreases exponentially as  $z \rightarrow 0$ . In the following example, the first call uses complex evaluation while the second has a small enough  $z$  to evaluate using the direct series and thus the returned value is strictly real (this should be taken to indicate that the imaginary part is less than  $\text{eps}$ ):

---

```
>>> mp.dps = 15
>>> hyp2f0(1.5, 0.5, 0.05)
(1.04166637647907 + 8.34584913683906e-8j)
>>> hyp2f0(1.5, 0.5, 0.0005)
1.00037535207621
```

---

The imaginary part can be retrieved by increasing the working precision:

---

```
>>> mp.dps = 80
>>> nprint(hyp2f0(1.5, 0.5, 0.009).imag)
1.23828e-46
```

---

## 11.3 Whittaker functions M and W

### 11.3.1 Whittaker function M

---

Function **whitm**(*k* As *mpNum*, *m* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function `whitm` returns the Whittaker function  $M$

**Parameters:**

*k*: A real or complex number.

*m*: A real or complex number.

*z*: A real or complex number.

The Whittaker function  $M(k, m, z)$  gives a solution to the Whittaker differential equation

$$\frac{d^2}{dz^2} + \left( -\frac{1}{4} + \frac{k}{z} + \frac{\frac{1}{4} - m^2}{z^2} \right) f = 0. \quad (11.3.1)$$

A second solution is given by `whitw()`.

The Whittaker functions are defined in Abramowitz & Stegun, section 13. They are alternate forms of the confluent hypergeometric functions  ${}_1F_1$  and  $U$ :

$$M(k, m, z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} {}_1F_1\left(\frac{1}{2} + m - k, 1 + 2m, z\right) \quad (11.3.2)$$

$$W(k, m, z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} U\left(\frac{1}{2} + m - k, 1 + 2m, z\right) \quad (11.3.3)$$

Examples

Evaluation for arbitrary real and complex arguments is supported:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> whitm(1, 1, 1)
0.7302596799460411820509668
>>> whitm(1, 1, -1)
(0.0 - 1.417977827655098025684246j)
>>> whitm(j, j/2, 2+3j)
(3.245477713363581112736478 - 0.822879187542699127327782j)
>>> whitm(2, 3, 100000)
4.303985255686378497193063e+21707
```

---

### 11.3.2 Whittaker function W

---

Function **whitw**(*k* As *mpNum*, *m* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function `whitw` returns the Whittaker function  $W$

**Parameters:**

*k*: A real or complex number.

*m*: A real or complex number.

*z*: A real or complex number.

The Whittaker function  $W(k, m, z)$  gives a solution to the Whittaker differential equation. See `whitw()`.

## Examples

Evaluation for arbitrary real and complex arguments is supported:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> whitw(1, 1, 1)
1.19532063107581155661012
>>> whitw(1, 1, -1)
(-0.9424875979222187313924639 - 0.2607738054097702293308689j)
>>> whitw(j, j/2, 2+3j)
(0.1782899315111033879430369 - 0.01609578360403649340169406j)
>>> whitw(2, 3, 100000)
1.887705114889527446891274e-21705
>>> whitw(-1, -1, 100)
1.905250692824046162462058e-24
```

---

## 11.4 Gauss Hypergeometric Function

### 11.4.1 Gauss Hypergeometric Function

---

Function **hyp2f1**(*a* As mpNum, *b* As mpNum, *c* As mpNum, *z* As mpNum) As mpNum

---

The function `hyp2f1` returns the square of *z*.

**Parameters:**

- a*: A real or complex number.
- b*: A real or complex number.
- c*: A real or complex number.
- z*: A real or complex number.

The Gauss hypergeometric function  ${}_2F_1$  (often simply referred to as *the* hypergeometric function), defined for  $|z| < 1$  by the series

$${}_2F_1(a, b; c; z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \cdot \frac{z^k}{k!} \quad (11.4.1)$$

and for  $|z| \geq 1$  by analytic continuation, with a branch cut on  $1, \infty$  when necessary.

Special cases of this function include many of the orthogonal polynomials as well as the incomplete beta function and other functions. Properties of the Gauss hypergeometric function are documented comprehensively in many references, for example Abramowitz & Stegun, section 15.

The implementation supports the analytic continuation as well as evaluation close to the unit circle where  $|z| \approx 1$ . The syntax `hyp2f1(a,b,c,z)` is equivalent to `hyper([a,b],[c],z)`.

This function returns the Gauss hypergeometric function  ${}_2F_1(a, b; c; x)$ , defined for  $|x| < 1$  by the series

$${}_2F_1(a, b; c; x) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \cdot \frac{x^k}{k!} \quad (11.4.2)$$

where  $(a)_k$  is the Pochammer symbol (see section 7.3)

The function has the following integral representation

$${}_2F_1(a, b; c; z) = B(a, c - a)^{-1} \int_0^1 \frac{t^{b-1} (1-t)^{c-b-1}}{(1-zt)^a} dt, \quad \Re c > \Re b > 0 \quad (11.4.3)$$

where  $B(\cdot, \cdot)$  is the Beta function (see section ??)

The following closed-form approximation based on a Laplace approximation has been derived by [Butler & Wood \(2002\)](#):

$${}_2F_1(a, b; c; z) \approx \frac{G_2(a, b, c; z)}{G_2(a, b, c; 0)}, \quad \text{where} \quad (11.4.4)$$

$$G_2(a, b, c; z) = w^{-1/2} y^a (1-y)^{c-a} (1-xy)^{-b},$$

$$w = a(1-y)^2 + (c-a)y^2 - bx^2y^2(1-y)^2/(1-xy)^2$$

$$y = [\tau + \sqrt{\tau^2 - 4ax(c-b)}]/[2x(b-c)], \text{ if } x \neq 0, \text{ and } y = a/c \text{ otherwise.}$$

$$\tau = x(b-a) - c.$$

## Examples

Evaluation with inside, outside and on the unit circle, for fixed parameters:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp2f1(2, (1,2), 4, 0.75)
1.303703703703703703704
>>> hyp2f1(2, (1,2), 4, -1.75)
0.7431290566046919177853916
>>> hyp2f1(2, (1,2), 4, 1.75)
(1.418075801749271137026239 - 1.114976146679907015775102j)
>>> hyp2f1(2, (1,2), 4, 1)
1.6
>>> hyp2f1(2, (1,2), 4, -1)
0.8235498012182875315037882
>>> hyp2f1(2, (1,2), 4, j)
(0.9144026291433065674259078 + 0.2050415770437884900574923j)
>>> hyp2f1(2, (1,2), 4, 2+j)
(0.9274013540258103029011549 + 0.7455257875808100868984496j)
>>> hyp2f1(2, (1,2), 4, 0.25j)
(0.9931169055799728251931672 + 0.06154836525312066938147793j)
```

---

Evaluation with complex parameter values:

---

```
>>> hyp2f1(1+j, 0.75, 10j, 1+5j)
(0.8834833319713479923389638 + 0.7053886880648105068343509j)
```

---

Evaluation with  $z = 1$  :

---

```
>>> hyp2f1(-2.5, 3.5, 1.5, 1)
0.0
>>> hyp2f1(-2.5, 3, 4, 1)
0.06926406926406926406926407
>>> hyp2f1(2, 3, 4, 1)
+inf
```

---

Arbitrarily large values of are supported:

---

```
>>> hyp2f1((-1,3), 1.75, 4, '1e100')
(7.883714220959876246415651e+32 + 1.365499358305579597618785e+33j)
>>> hyp2f1((-1,3), 1.75, 4, '1e1000000')
(7.883714220959876246415651e+333332 + 1.365499358305579597618785e+333333j)
>>> hyp2f1((-1,3), 1.75, 4, '1e1000000j')
(1.365499358305579597618785e+333333 - 7.883714220959876246415651e+333332j)
```

---

Verifying the differential equation:

---

```
>>> f = lambda z: hyp2f1(a,b,c,z)
>>> chop(z*(1-z)*diff(f,z,2) + (c-(a+b+1)*z)*diff(f,z) - a*b*f(z))
0.0
```

---

### 11.4.2 Gauss Regularized Hypergeometric Function

---

Function **Hypergeometric2F1RegularizedMpMath**(*a* As mpNum, *b* As mpNum, *c* As mpNum, *z* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function `Hypergeometric2F1RegularizedMpMath` returns the regularized Gauss hypergeometric function  ${}_2F_1(a, b; c; z)$ .

**Parameters:**

*a*: A real number.

*b*: A real number.

*c*: A real number.

*z*: A real number.

The regularized Gauss hypergeometric function  ${}_2\tilde{F}_1(a, b; c; z)$  for unrestricted *c*, is defined by [30, 15.1.2]

$${}_2\tilde{F}_1(a, b; c; z) = \frac{1}{\Gamma(c)} {}_2F_1(a, b; c; z) = \mathbf{F}(a, b; c; z), \quad (c \neq 0, -1, -2, \dots) \quad (11.4.5)$$

and by the corresponding limit if  $c = 0, -1, -2, \dots, = -m$ , with the value

$${}_2\tilde{F}_1(a, b; -m; z) = \frac{(a)_{m+1}(b)_{m+1}}{(m+1)!} x^{m+1} {}_2F_1(a+m+1, a+m+1; m+2; z) \quad (11.4.6)$$

where  $\Gamma(\cdot)$  is the Gamma function (see section ??) and  $(a)_k$  is the Pochammer symbol (see section 7.3)

The function has the following integral representation

$${}_2\tilde{F}_1(a, b; c; z) = \frac{1}{\Gamma(b)\Gamma(c-b)} \int_0^1 \frac{t^{b-1}(1-t)^{c-b-1}}{(1-zt)^a} dt, \quad \Re c > \Re b > 0 \quad (11.4.7)$$

## 11.5 Additional Hypergeometric Functions

### 11.5.1 Hypergeometric Function 1F2

---

Function **hyp1f2(a1 As mpNum, b1 As mpNum, b2 As mpNum, z As mpNum) As mpNum**

---

The function hyp1f2 returns the the hypergeometric function  ${}_1F_2(a_1; b_1, b_2; z)$

**Parameters:**

*a1*: A real or complex number.

*b1*: A real or complex number.

*b2*: A real or complex number.

*z*: A real or complex number.

Gives the hypergeometric function  ${}_1F_2(a_1; b_1, b_2; z)$ . The call hyp1f2(a1,b1,b2,z) is equivalent to hyper([a1],[b1,b2],z).

Evaluation works for complex and arbitrarily large arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> a, b, c = 1.5, (-1,3), 2.25
>>> hyp1f2(a, b, c, 10**20)
-1.159388148811981535941434e+8685889639
>>> hyp1f2(a, b, c, -10**20)
-12.60262607892655945795907
>>> hyp1f2(a, b, c, 10**20*j)
(4.237220401382240876065501e+6141851464 - 2.950930337531768015892987e+6141851464j)
>>> hyp1f2(2+3j, -2j, 0.5j, 10-20j)
(135881.9905586966432662004 - 86681.95885418079535738828j)
```

---

### 11.5.2 Hypergeometric Function 2F2

---

Function **hyp2f2(a1 As mpNum, a2 As mpNum, b1 As mpNum, b2 As mpNum, z As mpNum) As mpNum**

---

The function hyp2f2 returns the hypergeometric function  ${}_2F_2(a_1, a_2; b_1, b_2; z)$ .

**Parameters:**

*a1*: A real or complex number.

*a2*: A real or complex number.

*b1*: A real or complex number.

*b2*: A real or complex number.

*z*: A real or complex number.

Gives the hypergeometric function  ${}_2F_2(a_1, a_2; b_1, b_2; z)$ . The call hyp2f2(a1,a2,b1,b2,z) is equivalent to hyper([a1,a2],[b1,b2],z).

Evaluation works for complex and arbitrarily large arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> a, b, c, d = 1.5, (-1,3), 2.25, 4
```

---

```

>>> hyp2f2(a, b, c, d, 10**20)
-5.275758229007902299823821e+43429448190325182663
>>> hyp2f2(a, b, c, d, -10**20)
2561445.079983207701073448
>>> hyp2f2(a, b, c, d, 10**20*j)
(2218276.509664121194836667 - 1280722.539991603850462856j)
>>> hyp2f2(2+3j, -2j, 0.5j, 4j, 10-20j)
(80500.68321405666957342788 - 20346.82752982813540993502j)

```

---

### 11.5.3 Hypergeometric Function 2F3

---

Function **hyp2f3**(*a1* As *mpNum*, *a2* As *mpNum*, *b1* As *mpNum*, *b2* As *mpNum*, *b3* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function `hyp2f3` returns the hypergeometric function  ${}_2F_3(a_1, a_2; b_1, b_2, b_3; z)$ .

**Parameters:**

*a1*: A real or complex number.  
*a2*: A real or complex number.  
*b1*: A real or complex number.  
*b2*: A real or complex number.  
*b3*: A real or complex number.  
*z*: A real or complex number.

Gives the hypergeometric function  ${}_2F_3(a_1, a_2; b_1, b_2, b_3; z)$ . The call `hyp2f3(a1,a2,b1,b2,b3,z)` is equivalent to `hyper([a1,a2],[b1,b2,b3],z)`.

Evaluation works for complex and arbitrarily large arguments:

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> a1,a2,b1,b2,b3 = 1.5, (-1,3), 2.25, 4, (1,5)
>>> hyp2f3(a1,a2,b1,b2,b3,10**20)
-4.169178177065714963568963e+8685889590
>>> hyp2f3(a1,a2,b1,b2,b3,-10**20)
7064472.587757755088178629
>>> hyp2f3(a1,a2,b1,b2,b3,10**20*j)
(-5.163368465314934589818543e+6141851415 + 1.783578125755972803440364e+6141851416j)
>>> hyp2f3(2+3j, -2j, 0.5j, 4j, -1-j, 10-20j)
(-2280.938956687033150740228 + 13620.97336609573659199632j)
>>> hyp2f3(2+3j, -2j, 0.5j, 4j, -1-j, 10000000-20000000j)
(4.849835186175096516193e+3504 - 3.365981529122220091353633e+3504j)

```

---

### 11.5.4 Hypergeometric Function 3F2

---

Function **hyp3f2**(*a1* As *mpNum*, *a2* As *mpNum*, *a3* As *mpNum*, *b1* As *mpNum*, *b2* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function `hyp3f2` returns hypergeometric function  ${}_3F_2$ .

**Parameters:**

- $a_1$ : A real or complex number.
- $a_2$ : A real or complex number.
- $a_3$ : A real or complex number.
- $b_1$ : A real or complex number.
- $b_2$ : A real or complex number.
- $z$ : A real or complex number.

Gives the hypergeometric function  ${}_3F_2$ , defined for  $|z| < 1$  as

$${}_3F_2(a_1, a_2, a_3; b_1, b_2; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k (a_3)_k}{(b_1)_k (b_2)_k} \frac{z^k}{k!}, \quad (11.5.1)$$

and for  $|z| \geq 1$  by analytic continuation. The analytic structure of this function is similar to that of  ${}_2F_1$ , generally with a singularity at  $z = 1$  and a branch cut on  $(1, \infty)$ .

Evaluation is supported inside, on, and outside the circle of convergence  $|z| = 1$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> hyp3f2(1,2,3,4,5,0.25)
1.083533123380934241548707
>>> hyp3f2(1,2+2j,3,4,5,-10+10j)
(0.1574651066006004632914361 - 0.03194209021885226400892963j)
>>> hyp3f2(1,2,3,4,5,-10)
0.3071141169208772603266489
>>> hyp3f2(1,2,3,4,5,10)
(-0.4857045320523947050581423 - 0.5988311440454888436888028j)
>>> hyp3f2(0.25,1,1,2,1.5,1)
1.157370995096772047567631
>>> (8-pi-2*ln2)/3
1.157370995096772047567631
>>> hyp3f2(1+j,0.5j,2,1,-2j,-1)
(1.74518490615029486475959 + 0.1454701525056682297614029j)
>>> hyp3f2(1+j,0.5j,2,1,-2j,sqrt(j))
(0.9829816481834277511138055 - 0.4059040020276937085081127j)
>>> hyp3f2(-3,2,1,-5,4,1)
1.41
>>> hyp3f2(-3,2,1,-5,4,2)
2.12
```

---

Evaluation very close to the unit circle:

---

```
>>> hyp3f2(1,2,3,4,5,'1.0001')
(1.564877796743282766872279 - 3.76821518787438186031973e-11j)
>>> hyp3f2(1,2,3,4,5,'1+0.0001j')
(1.564747153061671573212831 + 0.000130575750366084557648482j)
>>> hyp3f2(1,2,3,4,5,'0.9999')
1.564616644881686134983664
>>> hyp3f2(1,2,3,4,5,'-0.9999')
0.7823896253461678060196207
```

---

## 11.6 Generalized hypergeometric functions

### 11.6.1 Generalized hypergeometric function $pFq$

---

Function **hyper(*as* As mpNum, *bs* As mpNum, *z* As mpNum) As mpNum**

---

The function `hyper` returns the generalized hypergeometric function  $pF_q$

**Parameters:**

*as*: list of real or complex numbers.

*bs*: list of real or complex numbers.

*z*: A real or complex number.

Evaluates the generalized hypergeometric function

$${}_pF_q(a_1, \dots, a_p; b_1, \dots, b_q; z) = \sum_{n=0}^{\infty} \frac{(a_1)_n (a_2)_n \dots (a_p)_n}{(b_1)_n (b_2)_n \dots (b_q)_n} \frac{z^n}{n!}, \quad (11.6.1)$$

where  $(x)_n$  denotes the rising factorial (see `rf()`).

The parameters lists *a\_s* and *b\_s* may contain integers, real numbers, complex numbers, as well as exact fractions given in the form of tuples  $(p, q)$ . `hyper()` is optimized to handle integers and fractions more efficiently than arbitrary floating-point parameters (since rational parameters are by far the most common).

The parameters can be any combination of integers, fractions, floats and complex numbers:

---

```
>>> a, b, c, d, e = 1, (-1,2), pi, 3+4j, (2,3)
>>> x = 0.2j
>>> hyper([a,b],[c,d,e],x)
(0.9923571616434024810831887 - 0.005753848733883879742993122j)
>>> b, e = -0.5, mpf(2)/3
>>> fn = lambda n: rf(a,n)*rf(b,n)/rf(c,n)/rf(d,n)/rf(e,n)*x**n/fac(n)
>>> nsum(fn, [0, inf])
(0.9923571616434024810831887 - 0.005753848733883879742993122j)
```

---

If any  $b_k$  is a nonpositive integer, the function is undefined (unless the series terminates before the division by zero occurs):

---

```
>>> hyper([1,1,1,-3],[-2,5],1)
Traceback (most recent call last):
...
ZeroDivisionError: pole in hypergeometric series
>>> hyper([1,1,1,-1],[-2,5],1)
1.1
```

---

Except for polynomial cases, the radius of convergence  $R$  of the hypergeometric series is either  $R = \infty$  (if  $p \leq q$ ),  $R = 1$  (if  $p = q + 1$ ), or  $R = 0$  (if  $p > q + 1$ ).

The analytic continuations of the functions with  $p = q + 1$ , i.e.  ${}_2F_1$ ,  ${}_3F_2$ ,  ${}_4F_3$ , etc, are all implemented and therefore these functions can be evaluated for  $|z| \geq 1$ . The shortcuts `hyp2f1()`, `hyp3f2()` are available to handle the most common cases (see their documentation), but functions of higher degree are also supported via `hyper()`:

---

```
>>> hyper([1,2,3,4], [5,6,7], 1) # 4F3 at finite-valued branch point
1.141783505526870731311423
```

---

```
>>> hyper([4,5,6,7], [1,2,3], 1) # 4F3 at pole
+inf
>>> hyper([1,2,3,4,5], [6,7,8,9], 10) # 5F4
(1.543998916527972259717257 - 0.5876309929580408028816365j)
>>> hyper([1,2,3,4,5,6], [7,8,9,10,11], 1j) # 6F5
(0.9996565821853579063502466 + 0.0129721075905630604445669j)
```

---

Please note that, as currently implemented, evaluation of  ${}_pF_{p-1}$  with  $p \geq 3$  may be slow or inaccurate when  $|z - 1|$  is small, for some parameter values.

When  $p > q + 1$ , `hyper` computes the (iterated) Borel sum of the divergent series. For  ${}_2F_0$  the Borel sum has an analytic solution and can be computed efficiently (see `hyp2f0()`). For higher degrees, the functions is evaluated first by attempting to sum it directly as an asymptotic series (this only works for tiny  $|z|$ ), and then by evaluating the Borel regularized sum using numerical integration. Except for special parameter combinations, this can be extremely slow.

---

```
>>> hyper([1,1], [], 0.5) # regularization of 2F0
(1.340965419580146562086448 + 0.8503366631752726568782447j)
>>> hyper([1,1,1,1], [1], 0.5) # regularization of 4F1
(1.108287213689475145830699 + 0.5327107430640678181200491j)
```

---

With the following magnitude of argument, the asymptotic series for  ${}_3F_1$  gives only a few digits. Using Borel summation, `hyper` can produce a value with full accuracy:

---

```
>>> mp.dps = 15
>>> hyper([2,0.5,4], [5.25], '0.08', force_series=True)
Traceback (most recent call last):
...
NoConvergence: Hypergeometric series converges too slowly. Try increasing maxterms.
>>> hyper([2,0.5,4], [5.25], '0.08', asymp_tol=1e-4)
1.0725535790737
>>> hyper([2,0.5,4], [5.25], '0.08')
(1.07269542893559 + 5.54668863216891e-5j)
>>> hyper([2,0.5,4], [5.25], '-0.08', asymp_tol=1e-4)
0.946344925484879
>>> hyper([2,0.5,4], [5.25], '-0.08')
0.946312503737771
>>> mp.dps = 25
>>> hyper([2,0.5,4], [5.25], '-0.08')
0.9463125037377662296700858
```

---

Note that with the positive  $z$  value, there is a complex part in the correct result, which falls below the tolerance of the asymptotic series.

## 11.6.2 Weighted combination of hypergeometric functions

---

Function **hypercomb**(*f* As *mpFunction*, *params* As *mpNum*, *z* As *mpNum*, **Keywords** As String) As *mpNum*

---

The function `hypercomb` returns a weighted combination of hypergeometric functions

**Parameters:**

*f*: a real or function.

*params*: list of real or complex numbers.

*z*: A real or complex number.

*Keywords*: `discardknownzeros=True`.

Computes a weighted combination of hypergeometric functions

$$\sum_{r=1}^N \left[ \prod_{k=1}^{l_r} (w_{r,k})^{c_{r,k}} \frac{\prod_{k=1}^{m_r} \Gamma(\alpha_{r,k})}{\prod_{k=1}^{n_r} \Gamma(\beta_{r,k})} {}^{p_r}F_{q_r}(a_{r,1}, \dots, a_{r,p}; b_{r,1}, \dots, b_{r,q}; z_r) \right] \quad (11.6.2)$$

Typically the parameters are linear combinations of a small set of base parameters; `hypercomb()` permits computing a correct value in the case that some of the  $\alpha$ ,  $\beta$ ,  $b$  turn out to be nonpositive integers, or if division by zero occurs for some  $w^c$ , assuming that there are opposing singularities that cancel out. The limit is computed by evaluating the function with the base parameters perturbed, at a higher working precision.

The first argument should be a function that takes the perturbable base parameters `params` as input and returns tuples `(w, c, alpha, beta, a, b, z)`, where the coefficients `w`, `c`, gamma factors `alpha`, `beta`, and hypergeometric coefficients `a`, `b` each should be lists of numbers, and `z` should be a single number.

Examples

The following evaluates

$$(a-1) \frac{\Gamma(a-3)}{\Gamma(a-4)} {}^1F_1(a, a-1, z) = e^z(a-4)(a+z-1) \quad (11.6.3)$$

with  $a = 1, z = 3$ . There is a zero factor, two gamma function poles, and the  $1F_1$  function is singular; all singularities cancel out to give a finite value:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> hypercomb(lambda a: [([a-1], [1], [a-3], [a-4], [a], [a-1], 3)], [1])
-180.769832308689
>>> -9*exp(3)
-180.769832308689
```

---

## 11.7 Meijer G-function

---

Function **meijerg**(*as* As mpNum, *bs* As mpNum, *z* As mpNum, **Keywords** As String) As mpNum

---

The function `meijerg` returns the Meijer G-function

**Parameters:**

*as*: list of real or complex numbers.

*bs*: list of real or complex numbers.

*z*: A real or complex number.

**Keywords**: *r*=1, *series*=1.

Evaluates the Meijer G-function, defined as

$$G_{p,q}^{m,n}\left(z; r \left| \begin{matrix} a_1, \dots, a_n, a_{n+1}, \dots, a_p \\ b_1, \dots, b_m, b_{m+1}, \dots, b_q \end{matrix} \right. \right) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j + s) \prod_{j=1}^n \Gamma(1 - a_j - s)}{\prod_{j=n+1}^p \Gamma(a_j + s) \prod_{j=m+1}^q \Gamma(1 - b_j - s)} z^{-s/r} ds \quad (11.7.1)$$

for an appropriate choice of the contour *L* (see references).

There are *p* elements *a<sub>j</sub>*. The argument *a\_s* should be a pair of lists, the first containing the *n* elements *a<sub>1</sub>, ..., a<sub>n</sub>* and the second containing the *p - n* elements *a<sub>n+1</sub>, ..., a<sub>p</sub>*.

There are *q* elements *a<sub>j</sub>*. The argument *b\_s* should be a pair of lists, the first containing the *m* elements *b<sub>1</sub>, ..., b<sub>m</sub>* and the second containing the *q - m* elements *b<sub>m+1</sub>, ..., b<sub>q</sub>*.

The implicit tuple (*m, n, p, q*) constitutes the order or degree of the Meijer G-function, and is determined by the lengths of the coefficient vectors. Confusingly, the indices in this tuple appear in a different order from the coefficients, but this notation is standard. The many examples given below should hopefully clear up any potential confusion.

The Meijer G-function is evaluated as a combination of hypergeometric series. There are two versions of the function, which can be selected with the optional *series* argument.

*series*=1 uses a sum of *m*  ${}_pF_{q-1}$  functions of *z*

*series*=2 uses a sum of *n*  ${}_qF_{p-1}$  functions of *1/z*

The default series is chosen based on the degree and  $|z|$  in order to be consistent with Mathematica's. This definition of the Meijer G-function has a discontinuity at  $|z| = 1$  for some orders, which can be avoided by explicitly specifying a series.

Keyword arguments are forwarded to `hypercomb()`.

Many standard functions are special cases of the Meijer G-function (possibly rescaled and/or with branch cut corrections). We define some test parameters:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> a = mpf(0.75)
>>> b = mpf(1.5)
>>> z = mpf(2.25)
```

---

The exponential function:

$$e^z = G_{0,1}^{1,0}\left(-z \left| \begin{matrix} 1 \\ 1 \end{matrix} \right. \right) \quad (11.7.2)$$

---

```
>>> meijerg([],[], [[0],[]], -z)
9.487735836358525720550369
>>> exp(z)
9.487735836358525720550369
```

---

The natural logarithm

$$\log(1+z) = G_{2,2}^{1,2} \left( -z \left| \begin{matrix} a_1, a_2 \\ b_1, b_2 \end{matrix} \right. \right) \quad (11.7.3)$$


---

```
>>> meijerg([[1,1],[]], [[1],[0]], z)
1.178654996341646117219023
>>> log(1+z)
1.178654996341646117219023
```

---

A rational function

$$\frac{z}{z+1} = G_{2,2}^{1,2} \left( -z \left| \begin{matrix} a_1, a_2 \\ b_1, b_2 \end{matrix} \right. \right) \quad (11.7.4)$$


---

```
>>> meijerg([[1,1],[]], [[1],[1]], z)
0.6923076923076923076923077
>>> z/(z+1)
0.6923076923076923076923077
```

---

The sine and cosine functions:

$$\frac{1}{\sqrt{\pi}} \sin(2\sqrt{z}) = G_{0,2}^{1,0} \left( z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (11.7.5)$$

$$\frac{1}{\sqrt{\pi}} \cos(2\sqrt{z}) = G_{0,2}^{1,0} \left( z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (11.7.6)$$


---

```
>>> meijerg([],[], [[0.5],[0]], (z/2)**2)
0.4389807929218676682296453
>>> sin(z)/sqrt(pi)
0.4389807929218676682296453
>>> meijerg([],[], [[0],[0.5]], (z/2)**2)
-0.3544090145996275423331762
>>> cos(z)/sqrt(pi)
-0.3544090145996275423331762
```

---

Bessel functions:

$$J_\alpha(2\sqrt{z}) = G_{0,2}^{1,0} \left( z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (11.7.7)$$

$$Y_\alpha(2\sqrt{z}) = G_{1,3}^{2,0} \left( z \left| \begin{matrix} a_1 \\ b_1, b_2, b_3 \end{matrix} \right. \right) \quad (11.7.8)$$

$$(-z)^{\alpha/2} z^{-\alpha/2} I_\alpha(2\sqrt{z}) = G_{0,2}^{2,0} \left( -z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (11.7.9)$$

$$2K_\alpha(2\sqrt{z}) = G_{0,2}^{2,0} \left( z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (11.7.10)$$

As the example with the Bessel I function shows, a branch factor is required for some arguments when inverting the square root.

---

```

>>> meijerg([],[], [[a/2], [-a/2]], (z/2)**2)
0.5059425789597154858527264
>>> besselj(a,z)
0.5059425789597154858527264
>>> meijerg([], [(-a-1)/2], [[a/2, -a/2], [(-a-1)/2]], (z/2)**2)
0.1853868950066556941442559
>>> bessely(a, z)
0.1853868950066556941442559
>>> meijerg([],[], [[a/2], [-a/2]], -(z/2)**2)
(0.8685913322427653875717476 + 2.096964974460199200551738j)
>>> (-z)**(a/2) / z**(a/2) * besseli(a, z)
(0.8685913322427653875717476 + 2.096964974460199200551738j)
>>> 0.5*meijerg([],[], [[a/2, -a/2], []], (z/2)**2)
0.09334163695597828403796071
>>> besselk(a,z)
0.09334163695597828403796071

```

---

Error functions:

$$\sqrt{\pi} z^{2(\alpha-1)} \operatorname{erfc}(z) = G_{1,2}^{2,0} \left( z, \frac{1}{2} \middle| \begin{matrix} a_1 \\ b_1, b_2 \end{matrix} \right) \quad (11.7.11)$$


---

```

>>> meijerg([], [a], [[a-1, a-0.5], []], z, 0.5)
0.00172839843123091957468712
>>> sqrt(pi) * z**(2*a-2) * erfc(z)
0.00172839843123091957468712

```

---

A Meijer G-function of higher degree, (1,1,2,3):

```

>>> meijerg([[a], [b]], [[a], [b, a-1]], z)
1.55984467443050210115617
>>> sin((b-a)*pi)/pi*(exp(z)-1)*z**(a-1)
1.55984467443050210115617

```

---

A Meijer G-function of still higher degree, (4,1,2,4), that can be expanded as a messy combination of exponential integrals:

```

>>> meijerg([[a], [2*b-a]], [[b, a, b-0.5, -1-a+2*b], []], z)
0.3323667133658557271898061
>>> chop(4*(a-b+1)*sqrt(pi)*gamma(2*b-2*a)*z**a*\n... expint(2*b-2*a, -2*sqrt(-z))*expint(2*b-2*a, 2*sqrt(-z)))
0.3323667133658557271898061

```

---

In the following case, different series give different values:

```

>>> chop(meijerg([[1], [0.25]], [[3], [0.5]], -2))
-0.06417628097442437076207337
>>> meijerg([[1], [0.25]], [[3], [0.5]], -2, series=1)
0.1428699426155117511873047
>>> chop(meijerg([[1], [0.25]], [[3], [0.5]], -2, series=2))
-0.06417628097442437076207337

```

---

## 11.8 Bilateral hypergeometric series

---

Function **bihyper(as As mpNum, bs As mpNum, z As mpNum, Keywords As String) As mpNum**

---

The function bihyper returns the bilateral hypergeometric series

**Parameters:**

*as*: list of real or complex numbers.

*bs*: list of real or complex numbers.

*z*: A real or complex number.

*Keywords*: r=1, series=1.

Evaluates the bilateral hypergeometric series

$${}_A H_B(a_1; \dots, a_A; b_1, \dots, b_B) = \sum_{n=-\infty}^{\infty} \frac{(a_1)_n \dots (a_A)_n}{(b_1)_n \dots (b_B)_n} z^n \quad (11.8.1)$$

where, for direct convergence,  $A = B$  and  $|z| = 1$ , although a regularized sum exists more generally by considering the bilateral series as a sum of two ordinary hypergeometric functions. In order for the series to make sense, none of the parameters may be integers.

Examples

The value of  ${}_2H_2$  at  $z = 1$  is given by Dougall's formula:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> a,b,c,d = 0.5, 1.5, 2.25, 3.25
>>> bihyper([a,b],[c,d],1)
-14.49118026212345786148847
>>> gammaproduct([c,d,1-a,1-b,c+d-a-b-1],[c-a,d-a,c-b,d-b])
-14.49118026212345786148847
```

---

The regularized function  ${}_1H_0$  can be expressed as the sum of one  ${}_2F_0$  function and one  ${}_1F_1$  function:

---

```
>>> a = mpf(0.25)
>>> z = mpf(0.75)
>>> bihyper([a],[],z)
(0.2454393389657273841385582 + 0.2454393389657273841385582j)
>>> hyper([a,1],[],z) + (hyper([1],[1-a],-1/z)-1)
(0.2454393389657273841385582 + 0.2454393389657273841385582j)
>>> hyper([a,1],[],z) + hyper([1],[2-a],-1/z)/z/(a-1)
(0.2454393389657273841385582 + 0.2454393389657273841385582j)
```

---

## 11.9 Hypergeometric functions of two variables

### 11.9.1 Generalized 2D hypergeometric series

---

Function **hyper2d(*a* As mpNum, *b* As mpNum, *x* As mpNum, *y* As mpNum) As mpNum**

---

The function `hyper2d` returns the sum the generalized 2D hypergeometric series

**Parameters:**

*a*: A real or complex number.

*b*: A real or complex number.

*x*: A real or complex number.

*y*: A real or complex number.

The sum of the generalized 2D hypergeometric series is calculated as

$$\sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{P((a), m, n)}{Q((b))} \frac{x^m y^n}{m!, n!} \quad (11.9.1)$$

where  $(a) = (a_1, \dots, a_r)$ ,  $(b) = (b_1, \dots, b_r)$  and where  $P$  and  $Q$  are products of rising factorials such as  $(a_j)_n$  or  $(a_j)_{m+n}$ .  $P$  and  $Q$  are specified in the form of dicts, with the  $m$  and  $n$  dependence as keys and parameter lists as values. The supported rising factorials are given in the following table (note that only a few are supported in  $Q$ ):

Key	Rising factorial	$Q$
”m”	$(a_j)_m$	Yes
”n”	$(a_j)_n$	Yes
”m+n”	$(a_j)_{m+n}$	Yes
”m-n”	$(a_j)_{m-n}$	No
”n-m”	$(a_j)_{n-m}$	No
”2m+n”	$(a_j)_{2m+n}$	No
”2m-n”	$(a_j)_{2m-n}$	No
”2n-m”	$(a_j)_{2n-m}$	No

For example, the Appell F1 and F4 functions

$$F_1 = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b)_m (c)_n}{(d)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (11.9.2)$$

$$F_4 = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b)_{m+n}}{(d)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (11.9.3)$$

can be represented respectively as

---

`hyper2d({'m+n': [a], 'm': [b], 'n': [c]}, {'m+n': [d]}, x, y)`

---

`hyper2d({'m+n': [a, b]}, {'m': [c], 'n': [d]}, x, y)`

---

More generally, `hyper2d()` can evaluate any of the 34 distinct convergent secondorder (generalized Gaussian) hypergeometric series enumerated by Horn, as well as the Kampe de Feriet function. The series is computed by rewriting it so that the inner series (i.e. the series containing  $n$  and  $y$ ) has the form of an ordinary generalized hypergeometric series and thereby can be evaluated

efficiently using `hyper()`. If possible, manually swapping  $x$  and  $y$  and the corresponding parameters can sometimes give better results.

#### Examples

Two separable cases: a product of two geometric series, and a product of two Gaussian hypergeometric functions:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> x, y = mpf(0.25), mpf(0.5)
>>> hyper2d({'m':1, 'n':1}, {}, x,y)
2.66666666666666666666666666667
>>> 1/(1-x)/(1-y)
2.66666666666666666666666666667
>>> hyper2d({'m':[1,2], 'n':[3,4]}, {'m':[5], 'n':[6]}, x,y)
4.164358531238938319669856
>>> hyp2f1(1,2,5,x)*hyp2f1(3,4,6,y)
4.164358531238938319669856
```

---

### 11.9.2 Appell F1 hypergeometric function

---

Function `appellf1(a As mpNum, b1 As mpNum, b2 As mpNum, c As mpNum, x As mpNum, y As mpNum)` As mpNum

---

The function `appellf1` returns the Appell F1 hypergeometric function of two variables.

#### Parameters:

*a*: A real or complex number.  
*b1*: A real or complex number.  
*b2*: A real or complex number.  
*c*: A real or complex number.  
*x*: A real or complex number.  
*y*: A real or complex number.

Gives the Appell F1 hypergeometric function of two variables,

$$F_1(a_1, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b_1)_m (b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (11.9.4)$$

This series is only generally convergent when  $|x| < 1$  and  $|y| < 1$ , although `appellf1()` can evaluate an analytic continuation with respect to either variable, and sometimes both.

#### Examples

Evaluation is supported for real and complex parameters:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf1(1,0,0.5,1,0.5,0.25)
1.154700538379251529018298
>>> appellf1(1,1+j,0.5,1,0.5,0.5j)
(1.138403860350148085179415 + 1.510544741058517621110615j)
```

---

### 11.9.3 Appell F2 hypergeometric function

---

Function **appellf2(*a* As mpNum, *b1* As mpNum, *b2* As mpNum, *c1* As mpNum, *c2* As mpNum, *x* As mpNum, *y* As mpNum) As mpNum**

---

The function **appellf2** returns the Appell F2 hypergeometric function of two variables.

**Parameters:**

*a*: A real or complex number.  
*b1*: A real or complex number.  
*b2*: A real or complex number.  
*c1*: A real or complex number.  
*c2*: A real or complex number.  
*x*: A real or complex number.  
*y*: A real or complex number.

Gives the Appell F2 hypergeometric function of two variables

$$F_2(a_1, b_1, b_2, c_1, c_2, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b_1)_m (b_2)_n}{(c_1)_m (c_2)_n} \frac{x^m y^n}{m! n!} \quad (11.9.5)$$

The series is generally absolutely convergent for  $|x| + |y| < 1$ .

Examples

Evaluation is supported for real and complex parameters:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf2(1,2,3,4,5,0.25,0.125)
1.257417193533135344785602
>>> appellf2(1,-3,-4,2,3,2,3)
-42.8
>>> appellf2(0.5,0.25,-0.25,2,3,0.25j,0.25)
(0.9880539519421899867041719 + 0.01497616165031102661476978j)
>>> chop(appellf2(1,1+j,1-j,3j,-3j,0.25,0.25))
1.201311219287411337955192
>>> appellf2(1,1,1,4,6,0.125,16)
(-0.09455532250274744282125152 - 0.7647282253046207836769297j)
```

---

### 11.9.4 Appell F3 hypergeometric function

---

Function **appellf3(*a1* As mpNum, *a2* As mpNum, *b1* As mpNum, *b2* As mpNum, *c* As mpNum, *x* As mpNum, *y* As mpNum) As mpNum**

---

The function **appellf3** returns the Appell F3 hypergeometric function of two variables.

**Parameters:**

*a1*: A real or complex number.  
*a2*: A real or complex number.  
*b1*: A real or complex number.  
*b2*: A real or complex number.  
*c*: A real or complex number.

*x*: A real or complex number.

*y*: A real or complex number.

Gives the Appell F3 hypergeometric function of two variables

$$F_3(a_1, a_2, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a_1)_m (a_2)_n (b_1)_m (b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (11.9.6)$$

The series is generally absolutely convergent for  $|x| < 1, |y| < 1$ .

Examples

Evaluation for various parameters and variables:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf3(1,2,3,4,5,0.5,0.25)
2.221557778107438938158705
>>> appellf3(1,2,3,4,5,6,0); hyp2f1(1,3,5,6)
(-0.5189554589089861284537389 - 0.1454441043328607980769742j)
(-0.5189554589089861284537389 - 0.1454441043328607980769742j)
>>> appellf3(1,-2,-3,1,1,4,6)
-17.4
>>> appellf3(1,2,-3,1,1,4,6)
(17.7876136773677356641825 + 19.54768762233649126154534j)
>>> appellf3(1,2,-3,1,1,6,4)
(85.02054175067929402953645 + 148.4402528821177305173599j)
>>> chop(appellf3(1+j,2,1-j,2,3,0.25,0.25))
1.719992169545200286696007
```

---

### 11.9.5 Appell F4 hypergeometric function

---

Function **appellf4**(*a* As *mpNum*, *b* As *mpNum*, *c1* As *mpNum*, *c2* As *mpNum*, *x* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function **appellf4** returns the Appell F4 hypergeometric function of two variables.

**Parameters:**

*a*: A real or complex number.

*b*: A real or complex number.

*c1*: A real or complex number.

*c2*: A real or complex number.

*x*: A real or complex number.

*y*: A real or complex number.

Gives the Appell F4 hypergeometric function of two variables

$$F_4(a, b, c_1, c_2, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (a_2)_n (b)_{m+n}}{(c_1)_m (c_2)_n} \frac{x^m y^n}{m!, n!} \quad (11.9.7)$$

The series is generally absolutely convergent for  $\sqrt{|x|} + \sqrt{|y|} < 1$ .

Examples

Evaluation for various parameters and variables:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> appellf4(1,1,2,2,0.25,0.125)
1.286182069079718313546608
>>> appellf4(-2,-3,4,5,4,5)
34.8
>>> appellf4(5,4,2,3,0.25j,-0.125j)
(-0.2585967215437846642163352 + 2.436102233553582711818743j)
```

---

# Chapter 12

## Elliptic functions

Elliptic functions historically comprise the elliptic integrals and their inverses, and originate from the problem of computing the arc length of an ellipse. From a more modern point of view, an elliptic function is defined as a doubly periodic function, i.e. a function which satisfies

$$f(z + 2\omega_1) = f(z + 2\omega_2) = f(z) \quad (12.0.1)$$

for some half-periods  $\omega_1, \omega_2$  with  $\Im[\omega_1/\omega_2] > 0$ . The canonical elliptic functions are the Jacobi elliptic functions. More broadly, this section includes quasi-doubly periodic functions (such as the Jacobi theta functions) and other functions useful in the study of elliptic functions.

Many different conventions for the arguments of elliptic functions are in use. It is even standard to use different parametrizations for different functions in the same text or software (and mpFormulaPy is no exception). The usual parameters are the elliptic nome  $q$ , which usually must satisfy  $|q| < 1$ ; the elliptic parameter  $m$  (an arbitrary complex number); the elliptic modulus  $k$  (an arbitrary complex number); and the half-period ratio  $\tau$ , which usually must satisfy  $\Im[\tau] > 0$ . These quantities can be expressed in terms of each other using the following relations:

$$m = k^2; \quad \tau = i \frac{K(1-m)}{K(m)}; \quad q = e^{i\pi\tau}; \quad k = \frac{\vartheta_2^4(q)}{\vartheta_2^4(q)} \quad (12.0.2)$$

In addition, an alternative definition is used for the nome in number theory, which we here denote by  $\bar{q}$ :

$$\bar{q} = q^2 = e^{2i\pi\tau} \quad (12.0.3)$$

### 12.1 Elliptic arguments

For convenience, mpFormulaPy provides functions to convert between the various parameters (`qfrom()`, `mfrom()`, `kfrom()`, `taufrom()`, `qbarfrom()`).

---

#### Function `qfrom(Keywords As String) As mpNum`

---

The function `qfrom` returns the elliptic nome  $q$ .

**Parameter:**

*Keywords:* `m=x`; `k=x`; `tau=x`; `qbar=x`.

Returns the elliptic nome  $q$ , given any of  $m, k, \tau, \bar{q}$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> qfrom(q=0.25)
0.25
>>> qfrom(m=mfrom(q=0.25))
0.25
>>> qfrom(k=kfrom(q=0.25))
0.25
>>> qfrom(tau=taufrom(q=0.25))
(0.25 + 0.0j)
>>> qfrom(qbar=qbarfrom(q=0.25))
0.25
```

---



---

### Function **qbarfrom**(*Keywords As String*) As mpNum

---

The function **qbarfrom** returns the number-theoretic nome  $\bar{q}$ .

**Parameter:**

*Keywords:* m=x; k=x; tau=x; q=x.

Returns the number-theoretic nome  $\bar{q}$ , given any of  $q, m, k, \tau$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> qbarfrom(qbar=0.25)
0.25
>>> qbarfrom(q=qfrom(qbar=0.25))
0.25
>>> qbarfrom(m=extraprec(20)(mfrom)(qbar=0.25)) # ill-conditioned
0.25
>>> qbarfrom(k=extraprec(20)(kfrom)(qbar=0.25)) # ill-conditioned
0.25
>>> qbarfrom(tau=taufrom(qbar=0.25))
(0.25 + 0.0j)
```

---



---

### Function **mfrom**(*Keywords As String*) As mpNum

---

The function **mfrom** returns the elliptic parameter  $m$ .

**Parameter:**

*Keywords:* k=x; tau=x; q=x; qbar=x.

Returns the elliptic parameter  $m$ , given any of  $q, k, \tau, \bar{q}$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> mfrom(m=0.25)
0.25
>>> mfrom(q=qfrom(m=0.25))
0.25
```

---

```
>>> mfrom(k=kfrom(m=0.25))
0.25
>>> mfrom(tau=taufrom(m=0.25))
(0.25 + 0.0j)
>>> mfrom(qbar=qbarfrom(m=0.25))
0.25
```

---



---

### Function **kfrom**(*Keywords As String*) As mpNum

---

The function **kfrom** returns the elliptic modulus  $k$ .

**Parameter:**

*Keywords:* m=x; tau=x; q=x; qbar=x.

Returns the elliptic modulus  $k$ , given any of  $q, m, \tau, \bar{q}$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> kfrom(k=0.25)
0.25
>>> kfrom(m=mfrom(k=0.25))
0.25
>>> kfrom(q=qfrom(k=0.25))
0.25
>>> kfrom(tau=taufrom(k=0.25))
(0.25 + 0.0j)
>>> kfrom(qbar=qbarfrom(k=0.25))
0.25
```

---



---

### Function **taufrom**(*Keywords As String*) As mpNum

---

The function **taufrom** returns the elliptic half-period ratio  $\tau$ .

**Parameter:**

*Keywords:* m=x; k=x; q=x; qbar=x.

Returns the elliptic half-period ratio  $\tau$ , given any of  $q, m, k, \bar{q}$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> taufrom(tau=0.5j)
(0.0 + 0.5j)
>>> taufrom(q=qfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(m=mfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(k=kfrom(tau=0.5j))
(0.0 + 0.5j)
>>> taufrom(qbar=qbarfrom(tau=0.5j))
(0.0 + 0.5j)
```

---

## 12.2 Legendre elliptic integrals

### 12.2.1 Complete elliptic integral of the first kind

---

Function **ellipk(*m* As mpNum)** As mpNum

---

The function **ellipk** returns the complete elliptic integral of the first kind,  $K(m)$ .

**Parameter:**

*m*: A real or complex number.

The complete elliptic integral of the first kind,  $K(m)$  is defined by

$$K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \frac{\pi}{2} {}_2F_1\left(\frac{1}{2}, \frac{1}{2}, 1, m\right) \quad (12.2.1)$$

Note that the argument is the parameter  $m = k^2$ , not the modulus  $k$  which is sometimes used.  
Values and limits include:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipk(0)
1.570796326794896619231322
>>> ellipk(inf)
(0.0 + 0.0j)
>>> ellipk(-inf)
0.0
>>> ellipk(1)
+inf
>>> ellipk(-1)
1.31102877714605990523242
>>> ellipk(2)
(1.31102877714605990523242 - 1.31102877714605990523242j)
```

---

Evaluation is supported for arbitrary complex *m*:

---

```
>>> ellipk(3+4j)
(0.9111955638049650086562171 + 0.6313342832413452438845091j)
```

---

### 12.2.2 Incomplete elliptic integral of the first kind

---

Function **ellipf(*phi* As mpNum, *m* As mpNum)** As mpNum

---

The function **ellipf** returns the Legendre incomplete elliptic integral of the first kind  $F(\phi, m)$ .

**Parameters:**

*phi*: A real or complex number.

*m*: A real or complex number.

The Legendre incomplete elliptic integral of the first kind is defined as

$$F(\phi, m) = \int_0^{\phi} \frac{dt}{\sqrt{1 - m \sin^2 t}} \quad (12.2.2)$$

or equivalently

$$F(\phi, m) = \int_0^{\sin \phi} \frac{dt}{\sqrt{1-t^2}\sqrt{1-mt^2}} \quad (12.2.3)$$

The function reduces to a complete elliptic integral of the first kind (see `ellipk()`) when  $\phi = \pi/2$ ; that is,  $F(\pi/2) = K(m)$ .

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside  $-\pi/2 \leq \Re(\phi) \leq \pi/2$ , the function extends quasi-periodically as

$$F(\phi + n\pi, m) = 2nK(m) + F(\phi, m), n \in \mathbb{Z}. \quad (12.2.4)$$

Basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipf(0,1)
0.0
>>> ellipf(0,0)
0.0
>>> ellipf(1,0); ellipf(2+3j,0)
1.0
(2.0 + 3.0j)
>>> ellipf(1,1); log(sec(1)+tan(1))
1.226191170883517070813061
1.226191170883517070813061
>>> ellipf(pi/2, -0.5); ellipk(-0.5)
1.415737208425956198892166
1.415737208425956198892166
>>> ellipf(pi/2+eps, 1); ellipf(-pi/2-eps, 1)
+inf
+inf
>>> ellipf(1.5, 1)
3.340677542798311003320813
```

---

Evaluation is supported for arbitrary complex  $m$ :

---

```
>>> ellipf(3j, 0.5)
(0.0 + 1.713602407841590234804143j)
>>> ellipf(3+4j, 5-6j)
(1.269131241950351323305741 - 0.3561052815014558335412538j)
>>> z,m = 2+3j, 1.25
>>> k = 1011
>>> ellipf(z+pi*k,m); ellipf(z,m) + 2*k*ellipk(m)
(4086.184383622179764082821 - 3003.003538923749396546871j)
(4086.184383622179764082821 - 3003.003538923749396546871j)
```

---

### 12.2.3 Complete elliptic integral of the second kind

---

Function **ellipe(*m* As mpNum) As mpNum**

---

The function `ellipe` returns the Legendre complete elliptic integral of the second kind  $E(m)$ .

**Parameter:**

*m*: A real or complex number.

The Legendre complete elliptic integral of the second kind is defined by

$$E(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \frac{\pi}{2} {}_2F_1\left(\frac{1}{2}, -\frac{1}{2}, 1, m\right) \quad (12.2.5)$$


---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipe(0)
1.570796326794896619231322
>>> ellipe(1)
1.0
>>> ellipe(-1)
1.910098894513856008952381
>>> ellipe(2)
(0.5990701173677961037199612 + 0.5990701173677961037199612j)
>>> ellipe(inf)
(0.0 + +infj)
>>> ellipe(-inf)
+inf
```

---

Evaluation is supported for arbitrary complex *m*:

```
>>> ellipe(0.5+0.25j)
(1.360868682163129682716687 - 0.1238733442561786843557315j)
>>> ellipe(3+4j)
(1.499553520933346954333612 - 1.577879007912758274533309j)
```

---

### 12.2.4 Incomplete elliptic integral of the second kind

---

Function **ellipef(*phi* As mpNum, *m* As mpNum) As mpNum**

---

The function `ellipef` returns the Legendre incomplete elliptic integral of the second kind  $E(\phi, m)$ .

**Parameters:**

*phi*: A real or complex number.

*m*: A real or complex number.

The incomplete elliptic integral of the second kind

$$E(\phi, m) = \int_0^{\phi} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}} dt \quad (12.2.6)$$

The incomplete integral reduces to a complete integral when  $\phi = \pi/2$ ; that is,  $E(\pi/2, m) = E(m)$ .

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside  $-\pi/2 \leq \Re(\phi) \leq \pi/2$ , the function extends quasi-periodically as

$$E(\phi + n\pi, m) = 2nE(m) + F(\phi, m), n \in \mathbb{Z}. \quad (12.2.7)$$

Basic values and limits:

---

```
>>> ellipe(0,1)
0.0
>>> ellipe(0,0)
0.0
>>> ellipe(1,0)
1.0
>>> ellipe(2+3j,0)
(2.0 + 3.0j)
>>> ellipe(1,1); sin(1)
0.8414709848078965066525023
0.8414709848078965066525023
>>> ellipe(pi/2, -0.5); ellipe(-0.5)
1.751771275694817862026502
1.751771275694817862026502
>>> ellipe(pi/2, 1); ellipe(-pi/2, 1)
1.0
-1.0
>>> ellipe(1.5, 1)
0.9974949866040544309417234
```

---

Evaluation is supported for arbitrary complex  $m$ :

---

```
>>> ellipe(0.5+0.25j)
>>> ellipe(3j, 0.5)
(0.0 + 7.551991234890371873502105j)
>>> ellipe(3+4j, 5-6j)
(24.15299022574220502424466 + 75.2503670480325997418156j)
>>> k = 35
>>> z,m = 2+3j, 1.25
>>> ellipe(z+pi*k,m); ellipe(z,m) + 2*k*ellipe(m)
(48.30138799412005235090766 + 17.47255216721987688224357j)
(48.30138799412005235090766 + 17.47255216721987688224357j)
```

---

## 12.2.5 Complete elliptic integral of the third kind

---

Function **ellippi(*n* As mpNum, *m* As mpNum) As mpNum**

---

The function **ellippi** returns the complete elliptic integral of the third kind  $\Pi(n, m)$ .

**Parameters:**

*n*: A real or complex number.

*m*: A real or complex number.

The complete elliptic integral of the third kind is defined as

$$\Pi(n, m) = \Pi(n; \pi/2, m).$$

Basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> ellippi(0,-5); ellipk(-5)
0.9555039270640439337379334
0.9555039270640439337379334
>>> ellippi(inf,2)
0.0
>>> ellippi(2,inf)
0.0
>>> abs(ellippi(1,5))
+inf
>>> abs(ellippi(0.25,1))
+inf
```

---

### 12.2.6 Incomplete elliptic integral of the third kind

---

Function **ellippif**(*n* As mpNum, *phi* As mpNum, *m* As mpNum) As mpNum

---

The function **ellippif** returns the Legendre incomplete elliptic integral of the third kind  $\Pi(n; \phi, m)$ .

**Parameters:**

*n*: A real or complex number.

*phi*: A real or complex number.

*m*: A real or complex number.

The Legendre incomplete elliptic integral of the third kind is defined as

$$\Pi(n; \phi, m) = \int_0^\phi \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{dt}{(1 - nt^2) \sqrt{1 - t^2} \sqrt{1 - mt^2}} \quad (12.2.8)$$

In the defining integral, it is assumed that the principal branch of the square root is taken and that the path of integration avoids crossing any branch cuts. Outside  $-\pi/2 \leq \Re(\phi) \leq \pi/2$ , the function extends quasi-periodically as

$$\Pi(n, \phi + k\pi, m) = 2k\Pi(n, m) + \Pi(n, \phi, m), k \in \mathbb{Z}. \quad (12.2.9)$$

Basic values and limits:

---

```
>>> ellippi(0.25,-0.5); ellippi(0.25,pi/2,-0.5)
1.622944760954741603710555
1.622944760954741603710555
>>> ellippi(1,0,1)
0.0
>>> ellippi(inf,0,1)
0.0
>>> ellippi(0,0.25,0.5); ellipf(0.25,0.5)
0.2513040086544925794134591
0.2513040086544925794134591
>>> ellippi(1,1,1); (log(sec(1)+tan(1))+sec(1)*tan(1))/2
2.054332933256248668692452
```

```
2.054332933256248668692452
>>> ellippi(0.25, 53*pi/2, 0.75); 53*ellippi(0.25,0.75)
135.240868757890840755058
135.240868757890840755058
>>> ellippi(0.5,pi/4,0.5); 2*ellipe(pi/4,0.5)-1/sqrt(3)
0.9190227391656969903987269
0.9190227391656969903987269
```

---

Complex arguments are supported:

```
>>> ellippi(0.5, 5+6j-2*pi, -7-8j)
(-0.3612856620076747660410167 + 0.5217735339984807829755815j)
```

---

## 12.3 Carlson symmetric elliptic integrals

The Carlson style elliptic integrals are a complete alternative group to the classical Legendre style integrals. They are symmetric and the numerical calculation is usually performed by duplication as described in [Carlson & Gustafson \(1994\)](#) and [Carlson \(1995\)](#).

### 12.3.1 Symmetric elliptic integral of the first kind, RF

---

Function **elliprf(x As mpNum, y As mpNum, z As mpNum) As mpNum**

---

The function `elliprf` returns the Carlson symmetric elliptic integral of the first kind.

**Parameters:**

- x*: A real or complex number.
- y*: A real or complex number.
- z*: A real or complex number.

The Carlson symmetric elliptic integral of the first kind is given by

$$R_F(x, y, z) = \frac{1}{2} \int_0^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}} \quad (12.3.1)$$

which is defined for  $x, y, z \in (-\infty, 0)$ , and with at most one of  $x, y, z$  being zero.

For real  $x, y, z \geq 0$ , the principal square root is taken in the integrand. For complex  $x, y, z$ , the principal square root is taken as  $t \rightarrow \infty$  and as  $t \rightarrow 0$  non-principal branches are chosen as necessary so as to make the integrand continuous.

Basic values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprf(0,1,1); pi/2
1.570796326794896619231322
1.570796326794896619231322
>>> elliprf(0,1,inf)
0.0
>>> elliprf(1,1,1)
1.0
>>> elliprf(2,2,2)**2
0.5
>>> elliprf(1,0,0); elliprf(0,0,1); elliprf(0,1,0); elliprf(0,0,0)
+inf
+inf
+inf
+inf
```

---

With the following arguments, the square root in the integrand becomes discontinuous at  $t = 1/2$  if the principal branch is used. To obtain the right value,  $-\sqrt{r}$  must be taken instead of  $\sqrt{r}$  on  $t \in (0, 1/2)$ :

---

```
>>> x,y,z = j-1,j,0
>>> elliprf(x,y,z)
(0.7961258658423391329305694 - 1.213856669836495986430094j)
```

---

```
>>> -q(f, [0,0.5]) + q(f, [0.5,inf])
(0.7961258658423391329305694 - 1.213856669836495986430094j)
```

---

### 12.3.2 Degenerate Carlson symmetric elliptic integral of the first kind, $R_C$

---

Function **elliprc**(*x* As *mpNum*, *y* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function **elliprc** returns the degenerate Carlson symmetric elliptic integral of the first kind.

**Parameters:**

*x*: A real or complex number.

*y*: A real or complex number.

**Keywords:** *pv*=True.

The degenerate Carlson symmetric elliptic integral of the first kind is given by

$$R_C(x, y) = R_F(x, y, y) = \frac{1}{2} \int_0^\infty \frac{dt}{(t+y)\sqrt{(t+x)}} \quad (12.3.2)$$

If  $y \in (-\infty, 0)$ , either a value defined by continuity, or with *pv*=True the Cauchy principal value, can be computed.

If  $x \geq 0, y > 0$ , the value can be expressed in terms of elementary functions as

$$R_C(x, y) = \begin{cases} \frac{1}{\sqrt{y-x}} \cos^{-1} \left( \sqrt{x/y} \right), & x < y \\ \frac{1}{\sqrt{y}}, & x = y \\ \frac{1}{\sqrt{y-x}} \cosh^{-1} \left( \sqrt{x/y} \right), & x > y \end{cases} \quad (12.3.3)$$

Examples

Some special values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprc(1,2)*4; elliprc(0,1)*2; +pi
3.141592653589793238462643
3.141592653589793238462643
3.141592653589793238462643
>>> elliprc(1,0)
+inf
>>> elliprc(5,5)**2
0.2
>>> elliprc(1,inf); elliprc(inf,1); elliprc(inf,inf)
0.0
0.0
0.0
```

---

Comparing with numerical integration:

---

```
>>> q = extradps(25)(quad)
>>> elliprc(2, -3, pv=True)
0.3333969101113672670749334
```

---

```
>>> elliprc(2, -3, pv=False)
(0.3333969101113672670749334 + 0.7024814731040726393156375j)
>>> 0.5*q(lambda t: 1/(sqrt(t+2)*(t-3)), [0,3-j,6,inf])
(0.3333969101113672670749334 + 0.7024814731040726393156375j)
```

---

### 12.3.3 Symmetric elliptic integral of the third kind, $R_J$

---

Function **elliprj(*x* As mpNum, *y* As mpNum, *z* As mpNum, *p* As mpNum) As mpNum**

---

The function `elliprj` returns the Carlson symmetric elliptic integral of the third kind.

**Parameters:**

*x*: A real or complex number.  
*y*: A real or complex number.  
*z*: A real or complex number.  
*p*: A real or complex number.

The Carlson symmetric elliptic integral of the third kind is given by

$$R_J(x, y, z, p) = \frac{3}{2} \int_0^{\infty} \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}} \quad (12.3.4)$$

Like `elliprf()`, the branch of the square root in the integrand is defined so as to be continuous along the path of integration for complex values of the arguments.

**Examples**

Some values and limits:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprj(1,1,1,1)
1.0
>>> elliprj(2,2,2,2); 1/(2*sqrt(2))
0.3535533905932737622004222
0.3535533905932737622004222
>>> elliprj(0,1,2,2)
1.067937989667395702268688
>>> 3*(2*gamma('5/4')**2-pi**2/gamma('1/4')**2)/(sqrt(2*pi))
1.067937989667395702268688
>>> elliprj(0,1,1,2); 3*pi*(2-sqrt(2))/4
1.380226776765915172432054
1.380226776765915172432054
>>> elliprj(1,3,2,0); elliprj(0,1,1,0); elliprj(0,0,0,0)
+inf
+inf
+inf
>>> elliprj(1,inf,1,0); elliprj(1,1,1,inf)
0.0
0.0
>>> chop(elliprj(1+j, 1-j, 1, 1))
0.8505007163686739432927844
```

---

Comparing with numerical integration:

---

```
>>> elliprj(1,2,3,4)
0.2398480997495677621758617
>>> f = lambda t: 1/((t+4)*sqrt((t+1)*(t+2)*(t+3)))
>>> 1.5*quad(f, [0,inf])
0.2398480997495677621758617
>>> elliprj(1,2+1j,3,4-2j)
(0.216888906014633498739952 + 0.04081912627366673332369512j)
>>> f = lambda t: 1/((t+4-2j)*sqrt((t+1)*(t+2+1j)*(t+3)))
>>> 1.5*quad(f, [0,inf])
(0.216888906014633498739952 + 0.04081912627366673332369511j)
```

---

### 12.3.4 Symmetric elliptic integral of the second kind, RD

---

Function **elliprd**(*x* As *mpNum*, *y* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function **elliprd** returns the Carlson symmetric elliptic integral of the second kind.

**Parameters:**

- x*: A real or complex number.
- y*: A real or complex number.
- z*: A real or complex number.

Evaluates the degenerate Carlson symmetric elliptic integral of the third kind or Carlson elliptic integral of the second kind  $R_D(x, y, z) = R_j(x, y, z, z)$ .

See **elliprj()** for additional information.

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprd(1,2,3)
0.2904602810289906442326534
>>> elliprj(1,2,3,3)
0.2904602810289906442326534
```

---

### 12.3.5 Completely symmetric elliptic integral of the second kind, RG

---

Function **elliprg**(*x* As *mpNum*, *y* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function **elliprg** returns the Carlson completely symmetric elliptic integral of the second kind.

**Parameters:**

- x*: A real or complex number.
- y*: A real or complex number.
- z*: A real or complex number.

The Carlson completely symmetric elliptic integral of the second kind is defined as

$$R_G(x, y, z) = \frac{1}{4} \int_0^\infty \frac{t}{\sqrt{(t+x)(t+y)(t+z)}} \left( \frac{x}{t+x} + \frac{y}{t+y} + \frac{z}{t+z} \right) dt \quad (12.3.5)$$

Evaluation for real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> elliprg(0,1,1)*4; +pi
3.141592653589793238462643
3.141592653589793238462643
>>> elliprg(0,0.5,1)
0.6753219405238377512600874
>>> chop(elliprg(1+j, 1-j, 2))
1.172431327676416604532822
```

---

## 12.4 Jacobi theta functions

---

Function **jtheta(*n* As mpNum, *z* As mpNum, *q* As mpNum, **Keywords** As String) As mpNum**

---

The function jtheta returns the Jacobi theta function  $\vartheta_n(z, q)$ .

**Parameters:**

*n*: An integer, where  $n = 1, 2, 3, 4$ .

*z*: A real or complex number.

*q*: A real or complex number.

**Keywords**: derivative=0.

The Jacobi theta function  $\vartheta_n(z, q)$ , where  $n = 1, 2, 3, 4$ , is defined by the infinite series:

$$\vartheta_1(z, q) = 2q^{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin(2n+1)z \quad (12.4.1)$$

$$\vartheta_2(z, q) = 2q^{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos(2n+1)z \quad (12.4.2)$$

$$\vartheta_3(z, q) = 1 + 2 \sum_{n=0}^{\infty} q^{n^2} \cos(2nz) \quad (12.4.3)$$

$$\vartheta_4(z, q) = 1 + 2 \sum_{n=0}^{\infty} (-1)^n q^{n^2} \cos(2nz) \quad (12.4.4)$$

The theta functions are functions of two variables:

*z* is the argument, an arbitrary real or complex number

*q* is the nome, which must be a real or complex number in the unit disk (i.e.  $|q| < 1$ ). For  $|q| \ll 1$ , the series converge very quickly, so the Jacobi theta functions can efficiently be evaluated to high precision.

The compact notations  $\vartheta_n(q) = \vartheta_n(0, q)$  and  $\vartheta_n = \vartheta_n(0, q)$  are also frequently encountered. Finally, Jacobi theta functions are frequently considered as functions of the half-period ratio  $\tau$  and then usually denoted by  $\vartheta_n(z|\tau)$ .

Optionally, jtheta(*n*, *z*, *q*, derivative=*d*) with  $d > 0$  computes a *d*-th derivative with respect to *z*. Examples and basic properties

Considered as functions of *z*, the Jacobi theta functions may be viewed as generalizations of the ordinary trigonometric functions cos and sin. They are periodic functions:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> jtheta(1, 0.25, '0.2')
0.2945120798627300045053104
>>> jtheta(1, 0.25 + 2*pi, '0.2')
0.2945120798627300045053104
```

---

jtheta() supports arbitrary precision evaluation and complex arguments:

```
>>> mp.dps = 50
>>> jtheta(4, sqrt(2), 0.5)
2.0549510717571539127004115835148878097035750653737
>>> mp.dps = 25
```

```
>>> jtheta(4, 1+2j, (1+j)/5)
(7.180331760146805926356634 - 1.634292858119162417301683j)
```

---

## 12.5 Jacobi elliptic functions

These procedures return the Jacobi elliptic functions  $\text{sn}$ ,  $\text{cn}$ ,  $\text{dn}$  for argument  $x$  and complementary parameter  $m_c$ . A convenient implicit definition of the functions is

$$x = \int_0^{\text{sn}} \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}}, \quad \text{sn}^2 + \text{cn}^2 = 1, \quad k^2\text{sn}^2 + \text{cn}^2 = 1 \quad (12.5.1)$$

with  $k^2 = 1 - m_c$ . There are a lot of equivalent definition of the Jacobi elliptic functions, e.g. with the Jacobi amplitude function (see e.g. [Olver et al. \(2010\)](#) [30, 22.16.11/12])

$$\begin{aligned} \text{sn}(x, k) &= \sin(\text{am}(x, k)), \\ \text{cn}(x, k) &= \cos(\text{am}(x, k)), \end{aligned}$$

or with Jacobi theta functions (cf. [\[Olver et al. \(2010\), 22.2\]](#)).

---

Function **ellipfun**(*kind* As String, *u* As mpNum, *m* As mpNum, **Keywords** As String) As mpNum

---

The function **ellipfun** returns any of the Jacobi elliptic functions.

### Parameters:

*kind*: A function identifier.

*u*: A real or complex number.

*m*: A real or complex number.

**Keywords**: *q*=None, *k*=None, *tau*=None.

Computes any of the Jacobi elliptic functions, defined in terms of Jacobi theta functions as

$$\text{sn}(u, m) = \frac{\vartheta_3(0, q)\vartheta_1(t, q)}{\vartheta_2(0, q)\vartheta_4(t, q)} \quad (12.5.2)$$

$$\text{cn}(u, m) = \frac{\vartheta_4(0, q)\vartheta_2(t, q)}{\vartheta_2(0, q)\vartheta_4(t, q)} \quad (12.5.3)$$

$$\text{dn}(u, m) = \frac{\vartheta_4(0, q)\vartheta_3(t, q)}{\vartheta_3(0, q)\vartheta_4(t, q)} \quad (12.5.4)$$

or more generally computes a ratio of two such functions. Here  $t = u/\vartheta_3(0, q)^2$ , and  $q = q(m)$  denotes the nome (see `nome()`). Optionally, you can specify the nome directly instead of by passing *q*=*value*, or you can directly specify the elliptic parameter with *k*=*value*.

The first argument should be a two-character string specifying the function using any combination of 's', 'c', 'd', 'n'. These letters respectively denote the basic functions  $\text{sn}(u, m)$ ,  $\text{cn}(u, m)$ ,  $\text{dn}(u, m)$ , and 1. The identifier specifies the ratio of two such functions. For example, 'ns' identifies the function

$$\text{cd}(u, m) = \frac{1}{\text{sn}(u, m)} \quad (12.5.5)$$

and 'cd' identifies the function

$$\text{ns}(u, m) = \frac{\text{cn}(u, m)}{\text{dn}(u, m)} \quad (12.5.6)$$

If called with only the first argument, a function object evaluating the chosen function for given arguments is returned.

Examples

Basic evaluation

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> ellipfun('cd', 3.5, 0.5)
-0.9891101840595543931308394
>>> ellipfun('cd', 3.5, q=0.25)
0.07111979240214668158441418
```

---

## 12.6 Klein j-invariant

---

Function **kleinj**(*tau* As mpNum) As mpNum

---

The function `kleinj` returns the Klein j-invariant.

**Parameter:**

*tau*: A real or complex number.

The Klein j-invariant is a modular function defined for  $\tau$  in the upper half-plane as

$$J(\tau) = \frac{g_2^3(\tau)}{g_2^3(\tau) - 27g_3^2(\tau)} \quad (12.6.1)$$

where  $g_2$  and  $g_3$  are the modular invariants of the Weierstrass elliptic function,

$$g_2(\tau) = 60 \sum_{(m,n) \in \mathbb{Z}^2 \setminus (0,0)} (m\tau + n)^{-4} \quad (12.6.2)$$

$$g_3(\tau) = 140 \sum_{(m,n) \in \mathbb{Z}^2 \setminus (0,0)} (m\tau + n)^{-6} \quad (12.6.3)$$

An alternative, common notation is that of the j-function  $j(\tau) = 1728J(\tau)$ .

Examples

Verifying the functional equation  $J(\tau) = J(\tau + 1) = J(-\tau^{-1})$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> tau = 0.625+0.75*j
>>> tau = 0.625+0.75*j
>>> kleinj(tau)
(-0.1507492166511182267125242 + 0.07595948379084571927228948j)
>>> kleinj(tau+1)
(-0.1507492166511182267125242 + 0.07595948379084571927228948j)
>>> kleinj(-1/tau)
(-0.1507492166511182267125242 + 0.07595948379084571927228946j)
```

---

# Chapter 13

## Zeta functions, L-series and polylogarithms

This section includes the Riemann zeta functions and associated functions pertaining to analytic number theory.

### 13.1 Riemann and Hurwitz zeta functions

---

Function **zeta**(*s* As mpNum, **Keywords** As String) As mpNum

---

The function **zeta** returns the Riemann zeta function

**Parameters:**

*s*: A real or complex number.

**Keywords**: derivative=0.

---

Function **hurwitz**(*s* As mpNum, *a* As mpNum, **Keywords** As String) As mpNum

---

The function **hurwitz** returns the Hurwitz zeta function

**Parameters:**

*s*: A real or complex number.

*a*: A real or complex number.

**Keywords**: derivative=0.

Computes the Riemann zeta function or the Hurwitz zeta function.

$$\zeta(s) = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \dots \quad (13.1.1)$$

or, with  $a \neq 1$ , the more general Hurwitz zeta function

$$\zeta(s, a) = \sum_{k=0}^{\infty} \frac{1}{(a+k)^s}. \quad (13.1.2)$$

Optionally, **zeta**(*s*, *a*, *n*) computes the *n*-th derivative with respect to *s*,

$$\zeta^{(n)}(s, a) = (-1)^n \sum_{k=0}^{\infty} \frac{\log^n(a+k)}{(a+k)^s}. \quad (13.1.3)$$

Although these series only converge for  $\Re(s) > 1$ , the Riemann and Hurwitz zeta functions are defined through analytic continuation for arbitrary complex  $s \neq 1$  ( $s = 1$  is a pole).

The implementation uses three algorithms: the Borwein algorithm for the Riemann zeta function when  $s$  is close to the real line; the Riemann-Siegel formula for the Riemann zeta function when  $s$  is large imaginary, and Euler-Maclaurin summation in all other cases. The reflection formula for  $\Re(s) < 0$  is implemented in some cases. The algorithm can be chosen with method = 'borwein', method='riemann-siegel' or method = 'euler-maclaurin'.

The parameter  $a$  is usually a rational number  $a = p/q$ , and may be specified as such by passing an integer tuple  $(p, q)$ . Evaluation is supported for arbitrary complex  $a$ , but may be slow and/or inaccurate when  $\Re(s) < 0$  for nonrational  $a$  or when computing derivatives.

Examples

Some values of the Riemann zeta function:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> zeta(2); pi**2 / 6
1.644934066848226436472415
1.644934066848226436472415
>>> zeta(0)
-0.5
>>> zeta(-1)
-0.08333333333333333333333333
>>> zeta(-2)
0.0
```

---

Evaluation is supported for complex  $s$  and  $a$ :

---

```
>>> zeta(-3+4j)
(-0.03373057338827757067584698 + 0.2774499251557093745297677j)
>>> zeta(2+3j, -1+j)
(389.6841230140842816370741 + 295.2674610150305334025962j)
```

---

Some values of the Hurwitz zeta function:

---

```
>>> zeta(2, 3); -5./4 + pi**2/6
0.3949340668482264364724152
0.3949340668482264364724152
>>> zeta(2, (3,4)); pi**2 - 8*catalan
2.541879647671606498397663
2.541879647671606498397663
```

---

## 13.2 Dirichlet L-series

### 13.2.1 Dirichlet eta function

---

Function **altzeta(s As mpNum) As mpNum**

---

The function `altzeta` returns the Dirichlet eta function,  $\eta(s)$

**Parameter:**

*s*: A real or complex number.

The Dirichlet eta function,  $\eta(s)$  is also known as the alternating zeta function. This function is defined in analogy with the Riemann zeta function as providing the sum of the alternating series

$$\eta(s) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k^s} = 1 - \frac{1}{2^s} + \frac{1}{3^s} - \frac{1}{4^s} + \dots \quad (13.2.1)$$

The eta function, unlike the Riemann zeta function, is an entire function, having a finite value for all complex *s*. The special case  $\eta(1) = \log(2)$  gives the value of the alternating harmonic series.

The alternating zeta function may be expressed using the Riemann zeta function as

$\eta(s) = (1 - 2^{1-s})\zeta(s)$ . It can also be expressed in terms of the Hurwitz zeta function, for example using `dirichlet()` (see documentation for that function).

Examples

Some special values are:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> altzeta(1)
0.693147180559945
>>> altzeta(0)
0.5
>>> altzeta(-1)
0.25
>>> altzeta(-2)
0.0
```

---

### 13.2.2 Dirichlet $\eta(s) - 1$

---

Function **DirichletEtam1MpMath(x As mpNum) As mpNum**

---

**NOT YET IMPLEMENTED**

---

The function `DirichletEtam1MpMath` returns the Dirichlet function  $\eta(s) - 1$ .

**Parameter:**

*x*: A real number.

This function returns the Dirichlet function  $\eta(s) - 1$ .

### 13.2.3 Dirichlet Beta Function

---

Function **DirichletBetaMpMath(s As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function `DirichletBetaMpMath` returns the Dirichlet function  $\beta(s)$ .

**Parameter:**

*s*: A real number.

This function returns the Dirichlet function  $\beta(s)$ , defined for  $s > 0$  as

$$\beta(s) = \sum_{n=1}^{\infty} \frac{(-1)^n}{(2n+1)^s} = 2^{-s} \Phi\left(-1, s, \frac{1}{2}\right) \quad (13.2.2)$$

### 13.2.4 Dirichlet Lambda Function

---

Function **DirichletLambdaMpMath(s As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function `DirichletLambdaMpMath` returns the Dirichlet function  $\beta(s)$ .

**Parameter:**

*s*: A real number.

This function returns the Dirichlet function  $\lambda(s)$ , defined for  $s > 0$  as

$$\lambda(s) = \sum_{n=0}^{\infty} (2n+1)^s \quad (13.2.3)$$

and by analytic continuation for  $s < 1$ . The function is calculated as

$$\lambda(s) = (1 - 2^{-s})\eta(s). \quad (13.2.4)$$

### 13.2.5 Dirichlet L-function

---

Function **dirichlet(s As mpNum, chi As mpNum, Keywords As mpNum) As mpNum**

---

The function `dirichlet` returns the Dirichlet L-function

**Parameters:**

*s*: A real or complex number.

*chi*: A periodic sequence.

*Keywords*: derivative=0.

The Dirichlet L-function is defined as

$$L(s, \chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s} \quad (13.2.5)$$

where  $\chi$  is a periodic sequence of length  $q$  which should be supplied in the form of a list  $[\chi(0), \chi(1), \dots, \chi(q-1)]$ . Strictly,  $\chi$  should be a Dirichlet character, but any periodic sequence will work.

For example, `dirichlet(s, [1])` gives the ordinary Riemann zeta function and `dirichlet(s, [-1,1])` gives the alternating zeta function (Dirichlet eta function).

Also the derivative with respect to (currently only a first derivative) can be evaluated.

Examples

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> dirichlet(3, [1]); zeta(3)
1.202056903159594285399738
1.202056903159594285399738
>>> dirichlet(1, [1])
+inf
```

---

### 13.3 Stieltjes constants

---

Function **stieltjes(*n* As mpNum, *a* As mpNum) As mpNum**

---

The function `stieltjes` returns the  $n$ -th Stieltjes constant

**Parameters:**

*n*: A real or complex number.

*a*: A real or complex number.

For a nonnegative integer  $n$ , `stieltjes(n)` computes the  $n$ -th Stieltjes constant  $\gamma_n$ , defined as the  $n$ -th coefficient in the Laurent series expansion of the Riemann zeta function around the pole at  $s = 1$ . That is, we have:

$$\zeta(s) = \frac{1}{s-1} \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n (s-1)^n \quad (13.3.1)$$

More generally, `stieltjes(n, a)` gives the corresponding coefficient  $\gamma_n(a)$  for the Hurwitz zeta function  $\zeta(s, a)$  (with  $\gamma_n = \gamma_n(1)$ ).

`stieltjes()` numerically evaluates the integral in the following representation due to Ainsworth, Howell and Coffey [1], [2]:

$$\gamma_n(a) = \frac{\log^n a}{2a} \frac{\log^{n+1}(a)}{n+1} + \frac{2}{a} \Re \int_0^\infty \frac{(x/a - i) \log^n(a - ix)}{(1 + x^2/a^2)(e^{2\pi x} - 1)} dx \quad (13.3.2)$$

Examples

The zeroth Stieltjes constant is just Euler's constant  $\gamma$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> stieltjes(0)
0.577215664901533
```

---

Some more values are:

---

```
>>> stieltjes(1)
-0.0728158454836767
>>> stieltjes(10)
0.000205332814909065
>>> stieltjes(30)
0.00355772885557316
>>> stieltjes(1000)
-1.57095384420474e+486
>>> stieltjes(2000)
2.680424678918e+1109
>>> stieltjes(1, 2.5)
-0.23747539175716
```

---

## 13.4 Zeta function zeros

These functions are used for the study of the Riemann zeta function in the critical strip.

---

### Function `zetazero(n As mpNum, Keywords As String) As mpNum`

---

The function `zetazero` returns the  $n$ -th nontrivial zero of  $\zeta(s)$  on the critical line

**Parameters:**

*n*: An integer.

*Keywords*: `verbose=False`.

Computes the  $n$ -th nontrivial zero of  $\zeta(s)$  on the critical line, i.e. returns an approximation of the  $n$ -th largest complex number  $s = \frac{1}{2} + ti$  for which  $\zeta(s) = 0$ .

Equivalently, the imaginary part  $t$  is a zero of the Z-function (`siegelz()`).

Examples

The first few zeros:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> zetazero(1)
(0.5 + 14.13472514173469379045725j)
>>> zetazero(2)
(0.5 + 21.02203963877155499262848j)
>>> zetazero(20)
(0.5 + 77.14484006887480537268266j)
```

---

Verifying that the values are zeros:

---

```
>>> for n in range(1,5):
...     s = zetazero(n)
...     chop(zeta(s)), chop(siegelz(s.imag))
...
(0.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
```

---



---

### Function `nzeros(t As mpNum) As mpNum`

---

The function `nzeros` returns the number of zeros of the Riemann zeta function in  $(0, 1) \times (0, t)$ , usually denoted by  $N(t)$ .

**Parameter:**

*t*: An integer.

Examples

The first zero has imaginary part between 14 and 15:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> nzeros(14)
0
>>> nzeros(15)
```

```
1
>>> zetazero(1)
(0.5 + 14.1347251417347j)
```

---

## 13.5 Riemann-Siegel Z function and related functions

### 13.5.1 Riemann-Siegel Z

---

Function **siegelz(*t* As mpNum)** As mpNum

---

The function `siegelz` returns the Riemann-Siegel Z function

**Parameter:**

*t*: A real or complex number.

The Riemann-Siegel Z function is defined as

$$Z(t) = e^{i\theta(t)} \zeta(1/2 + it) \quad (13.5.1)$$

where  $\zeta(s)$  is the Riemann zeta function (`zeta()`) and where  $\theta(t)$  denotes the Riemann-Siegel theta function (see `siegeltheta()`).

Evaluation is supported for real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> siegelz(1)
-0.7363054628673177346778998
>>> siegelz(3+4j)
(-0.1852895764366314976003936 - 0.2773099198055652246992479j)
```

---

The first four derivatives are supported, using the optional derivative keyword argument:

---

```
>>> siegelz(1234567, derivative=3)
56.89689348495089294249178
>>> diff(siegelz, 1234567, n=3)
56.89689348495089294249178
```

---

### 13.5.2 Riemann-Siegel theta function

---

Function **siegeltheta(*t* As mpNum)** As mpNum

---

The function `siegeltheta` returns the Riemann-Siegel theta function

**Parameter:**

*t*: A real or complex number.

The Riemann-Siegel theta function is defined as

$$\theta(t) = \frac{\log \Gamma(\frac{1+2it}{4}) - \log \Gamma(\frac{1-2it}{4})}{2i} - \frac{\log \pi}{2} t. \quad (13.5.2)$$

The Riemann-Siegel theta function is important in providing the phase factor for the Zfunction (see `siegelz()`). Evaluation is supported for real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> siegeltheta(0)
0.0
```

---

---

```
>>> siegeltheta(inf)
+inf
>>> siegeltheta(-inf)
-inf
>>> siegeltheta(1)
-1.767547952812290388302216
>>> siegeltheta(10+0.25j)
(-3.068638039426838572528867 + 0.05804937947429712998395177j)
```

---

Arbitrary derivatives may be computed with derivative = k

---

```
>>> siegeltheta(1234, derivative=2)
0.0004051864079114053109473741
>>> diff(siegeltheta, 1234, n=2)
0.0004051864079114053109473741
```

---

### 13.5.3 Gram point (Riemann-Siegel Z function)

---

Function **grampoint**(*n* As mpNum) As mpNum

---

The function **grampoint** returns the *n*-th Gram point  $g_n$ , defined as the solution to the equation  $\theta(g_n) = \pi n$  where  $\theta(t)$  is the Riemann-Siegel theta function

**Parameter:**

*n*: A real or complex number.

The first few Gram points are:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> grampoint(0)
17.84559954041086081682634
>>> grampoint(1)
23.17028270124630927899664
>>> grampoint(2)
27.67018221781633796093849
>>> grampoint(3)
31.71797995476405317955149
```

---

### 13.5.4 Backlunds function

---

Function **backlunds**(*t* As mpNum) As mpNum

---

The function **backlunds** returns the function  $S(t) = \arg\zeta\left(\frac{1}{2} + it\right)/\pi$ .

**Parameter:**

*t*: A real or complex number.

See Titchmarsh Section 9.3 for details of the definition.

Examples

---

```
>>> from mpFormulaPy import *
```

```
>>> mp.dps = 15; mp.pretty = True
>>> backlunds(217.3)
0.16302205431184
```

---

## 13.6 Lerch transcendent and related functions

---

Function **lerchphi(z As mpNum, s As mpNum, a As mpNum)** As mpNum

---

The function `lerchphi` returns the Lerch transcendent

**Parameters:**

*z*: A real or complex number.

*s*: A real or complex number.

*a*: A real or complex number.

The Lerch transcendent, defined for  $|z| < 1$  and  $\Re a > 0$ , is given by

$$\Phi(z, s, a) = \sum_{k=0}^{\infty} \frac{z^k}{(a+k)^s} \quad (13.6.1)$$

and generally by the recurrence  $\Phi(z, s, a) = z\Phi(z, s, a+1) + a^{-s}$  along with the integral representation valid for  $\Re a > 0$

$$\Phi(z, s, a) = \frac{1}{2a^s} + \int_0^{\infty} \frac{z^t}{(a+t)^s} dt - 2 \int_0^{\infty} \frac{\sin(t \log z) - s \arctan(t/a)}{(a^2 + t^2)^{s/2} (e^{2\pi t} - 1)} dt. \quad (13.6.2)$$

The Lerch transcendent generalizes the Hurwitz zeta function `zeta()` ( $z = 1$ ) and the polylogarithm `polylog()` ( $a = 1$ ).

Examples

Several evaluations in terms of simpler functions:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> lerchphi(-1,2,0.5); 4*catalan
3.663862376708876060218414
3.663862376708876060218414
>>> diff(lerchphi, (-1,-2,1), (0,1,0)); 7*zeta(3)/(4*pi**2)
0.2131391994087528954617607
0.2131391994087528954617607
>>> lerchphi(-4,1,1); log(5)/4
0.4023594781085250936501898
0.4023594781085250936501898
>>> lerchphi(-3+2j,1,0.5); 2*atanh(sqrt(-3+2j))/sqrt(-3+2j)
(1.142423447120257137774002 + 0.2118232380980201350495795j)
(1.142423447120257137774002 + 0.2118232380980201350495795j)
```

---

Evaluation works for complex arguments and  $|z| \geq 1$ :

---

```
>>> lerchphi(1+2j, 3-j, 4+2j)
(0.002025009957009908600539469 + 0.003327897536813558807438089j)
>>> lerchphi(-2,2,-2.5)
-12.28676272353094275265944
>>> lerchphi(10,10,10)
(-4.462130727102185701817349e-11 + 1.575172198981096218823481e-12j)
>>> lerchphi(10,10,-10.5)
(112658784011940.5605789002 + 498113185.5756221777743631j)
```

---

### 13.6.1 Fermi-Dirac integrals of integer order

---

Function **FermiDiracIntMpMath(x As mpNum, n As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function FermiDiracIntMpMath returns the complete Fermi-Dirac integrals  $F_n(x)$  of integer order.

**Parameters:**

*x*: A real number.

*n*: An Integer.

This function returns the complete Fermi-Dirac integrals  $F_n(x)$  of integer order. They are defined for real orders  $s > -1$  by

$$F_s(x) = \frac{1}{\Gamma(s+1)} \int_0^\infty \frac{t^s}{e^{t-x} + 1} dt \quad (13.6.3)$$

and by analytic continuation for  $s \leq -1$  using polylogarithms

$$F_s(x) = -\text{Li}_{s+1}(-e^x) = e^x \Phi(-e^x, s+1, 1). \quad (13.6.4)$$

### 13.6.2 Fermi-Dirac integral $F_{-1/2}(x)$

---

Function **FermiDiracPHalfMpMath(s As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function FermiDiracPHalfMpMath returns the complete Fermi-Dirac integral  $F_{-1/2}(x)$ .

**Parameter:**

*s*: A real number.

### 13.6.3 Fermi-Dirac integral $F_{1/2}(x)$

---

Function **FermiDiracHalfMpMath(s As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function FermiDiracHalfMpMath returns the complete Fermi-Dirac integral  $F_{1/2}(x)$ .

**Parameter:**

*s*: A real number.

### 13.6.4 Fermi-Dirac integral $F_{3/2}(x)$

---

Function **FermiDirac3HalfMpMath(s As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function FermiDirac3HalfMpMath returns the complete Fermi-Dirac integral  $F_{3/2}(x)$ .

**Parameter:**

*s*: A real number.

### 13.6.5 Legendre Chi-Function

---

Function **LegendreChiMpMath**(*s* As mpNum, *x* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **LegendreChiMpMath** returns the Legendre Chi-Function function  $\chi_s(x)$ .

**Parameters:**

*s*: A real number.

*x*: A real number.

This function calculates the Legendre Chi-Function function  $\chi_s(x)$  defined for  $s \geq 0, |x| \leq 1$  by

$$\chi_s(x) = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)^s}. \quad (13.6.5)$$

The function can be expressed as

$$\chi_s(x) = 2^{-s} x \Phi\left(x^2, s, \frac{1}{2}\right) = \frac{1}{2} (\text{Li}_s(x) - \text{Li}_s(-x)). \quad (13.6.6)$$

For large  $s > 22.8$  the function adds up to three terms of the sum, for  $s = 0$  or  $s = 1$  the *Li*<sub>*s*</sub> relation is used, otherwise the result is computed with Lerch's transcendent.

### 13.6.6 Inverse Tangent Integral

---

Function **InverseTangentMpMath**(*x* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **InverseTangentMpMath** returns the inverse-tangent integral.

**Parameter:**

*x*: A real number.

This function returns the inverse-tangent integral

$$\text{Ti}_2(x) = \int_0^x \frac{\arctan(t)}{t} dt. = \frac{1}{4} x \Phi\left(-x^2, 2, \frac{1}{2}\right) \quad (13.6.7)$$

For  $x > 1$  the relation

$$\text{Ti}_2(x) = \text{Ti}_2\left(\frac{1}{x}\right) + \frac{\pi}{2} \ln(x) \quad (13.6.8)$$

is used, and for  $x < 0$  the result is  $\text{Ti}_2(x) = -\text{Ti}_2(-x)$ .

## 13.7 Polylogarithms and Clausen functions

### 13.7.1 Polylogarithm

---

Function **polylog(s As mpNum, z As mpNum)** As mpNum

---

The function `polylog` returns the polylogarithm

**Parameters:**

*s*: A real or complex number.

*z*: A real or complex number.

The polylogarithm is defined by the sum

$$\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}. \quad (13.7.1)$$

This series is convergent only for  $|z| < 1$ , so elsewhere the analytic continuation is implied.

The polylogarithm should not be confused with the logarithmic integral (also denoted by `Li` or `li`), which is implemented as `li()`.

Examples

The polylogarithm satisfies a huge number of functional identities. A sample of polylogarithm evaluations is shown below:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> polylog(1,0.5), log(2)
(0.693147180559945, 0.693147180559945)
>>> polylog(2,0.5), (pi**2-6*log(2)**2)/12
(0.582240526465012, 0.582240526465012)
>>> polylog(2,-phi), -log(phi)**2-pi**2/10
(-1.21852526068613, -1.21852526068613)
>>> polylog(3,0.5), 7*zeta(3)/8-pi**2*log(2)/12+log(2)**3/6
(0.53721319360804, 0.53721319360804)
```

---

`polylog()` can evaluate the analytic continuation of the polylogarithm when *s* is an integer:

---

```
>>> polylog(2, 10)
(0.536301287357863 - 7.23378441241546j)
>>> polylog(2, -10)
-4.1982778868581
>>> polylog(2, 10j)
(-3.05968879432873 + 3.71678149306807j)
>>> polylog(-2, 10)
-0.150891632373114
>>> polylog(-2, -10)
0.067618332081142
>>> polylog(-2, 10j)
(0.0384353698579347 + 0.0912451798066779j)
```

---

### 13.7.2 Dilogarithm Function

---

Function **dilog**(*x* As *mpNum*) As *mpNum*

---

The function `dilog` returns the dilogarithm function  $\text{Li}_2(x)$ .

**Parameter:**

*x*: A real number.

The dilogarithm function is defined as

$$\text{dilog}(x) = \Re \text{Li}_2(x) = -\Re \int_0^x \frac{\ln(1-t)}{t} dt. \quad (13.7.2)$$

Note that there is some confusion about the naming: some authors and/or computer algebra systems use  $\text{dilog}(x) = \text{Li}_2(1-x)$  and then call  $\text{Li}_2(x)$  Spence function/integral or similar.

### 13.7.3 Debye Functions

---

Function **DebyeMpMath**(*n* As *mpNum*, *x* As *mpNum*) As *mpNum*

---

**NOT YET IMPLEMENTED**

---

The function `DebyeMpMath` returns the Debye function of order *n*.

**Parameters:**

*n*: An Integer.

*x*: A real number.

This routine returns the Debye functions

$$D_n(x) = \frac{n}{x^n} \int_0^x \frac{t^n}{e^t - 1} dt \quad (n > 0, x \geq 0). \quad (13.7.3)$$

$$D_k(x) = \frac{k}{x^{k+1}} \left[ (-1)^k k! \zeta(k+1) + \sum_{m=0}^k (-1)^{k-m+1} \frac{k!}{m!} x^m \text{Li}_{k-m+1}(e^x) \right] - \frac{k}{k+1}, \quad (13.7.4)$$

where  $\text{Li}_s$  denotes the polylogarithm (see [Dubinov & Dubinova \(2008\)](#)).

### 13.7.4 Clausen sine function

---

Function **clsinlog**(*s* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function `clsinlog` returns the Clausen sine function

**Parameters:**

*s*: A real or complex number.

*z*: A real or complex number.

The Clausen sine function is defined formally by the series

$$\text{Cl}_s(z) = \sum_{k=1}^{\infty} \frac{\sin(kz)}{k^s}. \quad (13.7.5)$$

The special case  $\text{Cl}_2(z)$  (i.e.  $\text{clsin}(2, z)$ ) is the classical 'Clausen function'. More generally, the Clausen function is defined for complex  $s$  and  $z$ , even when the series does not converge. The Clausen function is related to the polylogarithm ( $\text{polylog}()$ ) as

$$\text{Cl}_s(z) = \frac{1}{2i} [\text{Li}_s(e^{iz}) - \text{Li}_s(e^{-iz})] \quad (13.7.6)$$

$$\text{Cl}_s(z) = \Im [\text{Li}_s(e^{iz})], \quad (s, z \in \mathbb{R}), \quad (13.7.7)$$

and this representation can be taken to provide the analytic continuation of the series. The complementary function  $\text{clcos}()$  gives the corresponding cosine sum.

Examples

Evaluation for arbitrarily chosen  $s$  and  $z$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> s, z = 3, 4
>>> clsin(s, z); nsum(lambda k: sin(z*k)/k**s, [1,inf])
-0.6533010136329338746275795
-0.6533010136329338746275795
```

---

The classical Clausen function  $\text{Cl}_s(z)$  gives the value of the integral  $\int_0^\theta -\ln(2\sin(x/2))dx$  for  $0 < \theta < 2\pi$ :

---

```
>>> cl2 = lambda t: clsin(2, t)
>>> cl2(3.5)
-0.2465045302347694216534255
>>> -quad(lambda x: ln(2*sin(0.5*x)), [0, 3.5])
-0.2465045302347694216534255
```

---

### 13.7.5 Clausen cosine function

---

Function **clcos(s As mpNum, z As mpNum)** As mpNum

---

The function **clcos** returns the Clausen cosine function

**Parameters:**

**s:** A real or complex number.

**z:** A real or complex number.

The Clausen cosine function is defined formally by the series

$$\widetilde{\text{Cl}}_s(z) = \sum_{k=1}^{\infty} \frac{\cos(kz)}{k^s}. \quad (13.7.8)$$

This function is complementary to the Clausen sine function  $\text{clsin}()$ . In terms of the polylogarithm,

$$\widetilde{\text{Cl}}_s(z) = \frac{1}{2} [\text{Li}_s(e^{iz}) - \text{Li}_s(e^{-iz})] \quad (13.7.9)$$

$$\widetilde{\text{Cl}}_s(z) = \Re [\text{Li}_s(e^{iz})], \quad (s, z \in \mathbb{R}), \quad (13.7.10)$$

Examples

Evaluation for arbitrarily chosen  $s$  and  $z$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> s, z = 3, 4
>>> clcos(s, z); nsum(lambda k: cos(z*k)/k**s, [1,inf])
-0.6518926267198991308332759
-0.6518926267198991308332759
```

---

### 13.7.6 Polyexponential function

---

Function **polyexp(s As mpNum, z As mpNum)** As mpNum

---

The function `polyexp` returns the polyexponential function

**Parameters:**

`s`: A real or complex number.

`z`: A real or complex number.

The polyexponential function is defined for arbitrary complex  $s, z$  by the series

$$E_s(z) = \sum_{k=1}^{\infty} \frac{k^s}{k!} z^k. \quad (13.7.11)$$

$E_s(z)$  is constructed from the exponential function analogously to how the polylogarithm is constructed from the ordinary logarithm; as a function of  $s$  (with  $z$  fixed),  $E_s$  is an L-series. It is an entire function of both  $s$  and  $z$ .

The polyexponential function provides a generalization of the Bell polynomials  $B_n(x)$  (see `bell()`) to noninteger orders  $n$ . In terms of the Bell polynomials,

$$E_s(z) = e^z B_s(z) - \text{sinc}(\pi s). \quad (13.7.12)$$

Note that  $B_n(x)$  and  $e^{-x}E_n(x)$  are identical if  $n$  is a nonzero integer, but not otherwise. In particular, they differ at  $n = 0$ .

Examples

Evaluating a series:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> nsum(lambda k: sqrt(k)/fac(k), [1,inf])
2.101755547733791780315904
>>> polyexp(0.5,1)
2.101755547733791780315904
```

---

Evaluation for arbitrary arguments:

---

```
>>> polyexp(-3-4j, 2.5+2j)
(2.351660261190434618268706 + 1.202966666673054671364215j)
```

---

## 13.8 Zeta function variants

### 13.8.1 Prime zeta function

---

Function **primezeta**(*s* As *mpNum*) As *mpNum*

---

The function `primezeta` returns the prime zeta function.

**Parameter:**

*s*: A real or complex number.

The prime zeta function is defined in analogy with the Riemann zeta function (`zeta()`) as

$$P(s) = \sum_p \frac{1}{p^s} \quad (13.8.1)$$

where the sum is taken over all prime numbers *p*. Although this sum only converges for  $\Re(s) > 1$ , the function is defined by analytic continuation in the half-plane  $\Re(s) > 0$ .

Examples

Arbitrary-precision evaluation for real and complex arguments is supported:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 30; mp.pretty = True
>>> primezeta(2)
0.452247420041065498506543364832
>>> primezeta(pi)
0.15483752698840284272036497397
>>> mp.dps = 50
>>> primezeta(3)
0.17476263929944353642311331466570670097541212192615
>>> mp.dps = 20
>>> primezeta(3+4j)
(-0.12085382601645763295 - 0.013370403397787023602j)
```

---

The analytic continuation to  $0 < \Re(s) \leq 1$  is implemented. In this strip the function exhibits very complex behavior; on the unit interval, it has poles at  $1/n$  for every squarefree integer *n*:

---

```
>>> primezeta(0.5) # Pole at s = 1/2
(-inf + 3.1415926535897932385j)
>>> primezeta(0.25)
(-1.0416106801757269036 + 0.52359877559829887308j)
>>> primezeta(0.5+10j)
(0.54892423556409790529 + 0.45626803423487934264j)
```

---

### 13.8.2 Secondary zeta function

---

Function **secondzeta**(*s* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function `secondzeta` returns the secondary zeta function

**Parameters:**

*s*: A real or complex number.

**Keywords:** a=0.015, error=False.

The secondary zeta function  $Z(s)$  is defined for  $\Re(s) > 1$  by

$$Z(s) = \sum_{n=1}^{\infty} \frac{1}{\tau_n^s} \quad (13.8.2)$$

where  $\frac{1}{2} + i\tau_n$  runs through the zeros of  $\zeta(s)$  with imaginary part positive.

$Z(s)$  extends to a meromorphic function on  $\mathbb{C}$  with a double pole at  $s = 1$  and simple poles at the points  $-2n$  for  $n = 0, 1, 2, \dots$

Examples

---

```
>>> from mpFormulaPy import *
>>> mp.pretty = True; mp.dps = 15
>>> secondzeta(2)
0.023104993115419
>>> xi = lambda s: 0.5*s*(s-1)*pi**(-0.5*s)*gamma(0.5*s)*zeta(s)
>>> Xi = lambda t: xi(0.5+t*j)
>>> -0.5*diff(Xi,0,n=2)/Xi(0)
(0.023104993115419 + 0.0j)
```

---

We may ask for an approximate error value:

---

```
>>> secondzeta(0.5+100j, error=True)
((-0.216272011276718 - 0.844952708937228j), 2.22044604925031e-16)
```

---

# Chapter 14

## Number-theoretical, combinatorial and integer functions

For factorial-type functions, including binomial coefficients, double factorials, etc., see the separate section Factorials and gamma functions.

### 14.1 Fibonacci numbers

---

#### Function **fibonacci**(*n* As mpNum, *Keywords* As String) As mpNum

---

The function fibonacci returns the  $n$ -th Fibonacci number,  $F(n)$

**Parameters:**

*n*: A real or complex number.

*Keywords*: derivative=0.

---

#### Function **fib**(*n* As mpNum, *Keywords* As String) As mpNum

---

The function fib returns the  $n$ -th Fibonacci number,  $F(n)$

**Parameters:**

*n*: A real or complex number.

*Keywords*: derivative=0.

The Fibonacci numbers are defined by the recurrence  $F(n) = F(n-1) + F(n-2)$  with the initial values  $F(0) = 0$ ,  $F(1) = 1$ . fibonacci() extends this definition to arbitrary real and complex arguments using the formula

$$F(z) = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}} \quad (14.1.1)$$

where  $\phi$  is the golden ratio. fibonacci() also uses this continuous formula to compute  $F(n)$  for extremely large  $n$ , where calculating the exact integer would be wasteful.

For convenience, fib() is available as an alias for fibonacci().

Basic examples

Some small Fibonacci numbers are:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
```

---

```
>>> for i in range(10):
...     print(fibonacci(i))
...
0.0
1.0
1.0
2.0
3.0
5.0
8.0
13.0
21.0
34.0
>>> fibonacci(50)
12586269025.0
```

---

`fibonacci()` can compute approximate Fibonacci numbers of stupendous size:

---

```
>>> mp.dps = 15
>>> fibonacci(10**25)
3.49052338550226e+2089876402499787337692720
```

---

The extended Fibonacci function is an analytic function. The property  $F(z) = F(z-1) + F(z-2)$  holds for arbitrary  $z$ :

---

```
>>> mp.dps = 15
>>> fib(pi)
2.1170270579161
>>> fib(pi-1) + fib(pi-2)
2.1170270579161
>>> fib(3+4j)
(-5248.51130728372 - 14195.962288353j)
>>> fib(2+4j) + fib(1+4j)
(-5248.51130728372 - 14195.962288353j)
```

---

## 14.2 Bernoulli numbers and polynomials

### 14.2.1 Bernoulli numbers

---

#### Function **bernoulli(*n* As mpNum) As mpNum**

---

The function `bernoulli` returns the  $n$ th Bernoulli number,  $B_n$ , for any integer  $n > 0$

**Parameter:**

$n$ : An integer

The Bernoulli numbers  $B_n$  are defined by their generating function

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}, \quad |t| < 2\pi. \quad (14.2.1)$$

If  $n < 0$  or if  $n > 2$  is odd, the result is 0, and  $B_1 = -1/2$ . If  $n \leq 120$  the function value is taken from a pre-calculated table. For large  $n$  the asymptotic approximation [30, 24.11.1]

$$(-1)^{n+1} B_{2n} \approx \frac{2(2n)!}{(2\pi)^{2n}}, \quad (14.2.2)$$

gives an asymptotic recursion formula

$$B_{2n+2} \approx -\frac{(2n+1)(2n+2)}{(2\pi)^2} B_{2n}, \quad (14.2.3)$$

which is used for computing  $B_n$  for  $120 < n \leq 2312$  from a pre-calculated table of values  $B_{32k+128}$  ( $0 \leq k \leq 68$ ). The average iteration count is 4, and the maximum relative error of 4.5 eps occurs for  $n = 878$ .

Computes the  $n$ th Bernoulli number,  $B_n$ , for any integer  $n > 0$ . The Bernoulli numbers are rational numbers, but this function returns a floating-point approximation. To obtain an exact fraction, use `bernfrac()` instead.

For small  $n$  ( $n < 3000$ ) `bernoulli()` uses a recurrence formula due to Ramanujan. All results in this range are cached, so sequential computation of small Bernoulli numbers is guaranteed to be fast. For larger  $n$ ,  $B_n$  is evaluated in terms of the Riemann zeta function

Examples

Numerical values of the first few Bernoulli numbers:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(15):
...     print("%s %s" % (n, bernoulli(n)))
...
0 1.0
1 -0.5
2 0.166666666666667
3 0.0
4 -0.0333333333333333
5 0.0
6 0.0238095238095238
7 0.0
8 -0.0333333333333333
9 0.0
```

---

```
10 0.0757575757575758
11 0.0
12 -0.253113553113553
13 0.0
14 1.166666666666667
```

---

Bernoulli numbers can be approximated with arbitrary precision:

---

```
>>> mp.dps = 50
>>> bernoulli(100)
-2.8382249570693706959264156336481764738284680928013e+78
```

---

Arbitrarily large are supported:

---

```
>>> mp.dps = 15
>>> bernoulli(10**20 + 2)
3.09136296657021e+1876752564973863312327
```

---



---

### Function **bernfrac(*n* As mpNum) As mpNum**

---

The function `bernfrac` returns a tuple of integers  $(p, q)$  such that  $p/q = B_n$  exactly, where  $B_n$  denotes the  $n$ -th Bernoulli number.

**Parameter:**

*n*: An integer

The fraction is always reduced to lowest terms. Note that for  $n > 1$  and  $n$  odd,  $B_n = 0$ , and  $(0, 1)$  is returned.

`bernoulli()` computes a floating-point approximation directly, without computing the exact fraction first. This is much faster for large  $n$ .

`bernfrac()` works by computing the value of  $B_n$  numerically and then using the von Staudt-Clausesen theorem [1] to reconstruct the exact fraction. For large  $n$ , this is significantly faster than computing  $B_1, B_2, \dots, B_n$  recursively with exact arithmetic.

The implementation has been tested for  $n = 10^m$  up to  $m = 6$ . In practice, `bernfrac()` appears to be about three times slower than the specialized program `calcbn.exe` [2]

Examples

The first few Bernoulli numbers are exactly:

---

```
>>> from mpFormulaPy import *
>>> for n in range(15):
...     p, q = bernfrac(n)
...     print("%s %s/%s" % (n, p, q))
...
0 1/1
1 -1/2
2 1/6
3 0/1
4 -1/30
5 0/1
```

```

6 1/42
7 0/1
8 -1/30
9 0/1
10 5/66
11 0/1
12 -691/2730
13 0/1
14 7/6

```

---

This function works for arbitrarily large  $n$ :

```

>>> p, q = bernfrac(10**4)
>>> print(q)
2338224387510
>>> print(len(str(p)))
27692
>>> mp.dps = 15
>>> print(mpf(p) / q)
-9.04942396360948e+27677
>>> print(bernoulli(10**4))
-9.04942396360948e+27677

```

---

## 14.2.2 Bernoulli polynomials

---

Function **bernpoly( $n$  As mpNum,  $z$  As mpNum)** As mpNum

---

The function `bernpoly` returns the Bernoulli polynomial  $B_n(z)$

**Parameters:**

$n$ : A real or complex number.

$z$ : A real or complex number.

The Bernoulli polynomials  $B_n(x)$  of degree  $n \geq 0$  are defined by the generating function [30, 24.2.3]

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!}, \quad |t| < 2\pi. \quad (14.2.4)$$

or the simple explicit representation [30, 24.2.5]

$$B_n(x) = \sum_{n=0}^{\infty} \binom{n}{k} B_k(x) x^{n-k}. \quad (14.2.5)$$

The first few Bernoulli polynomials are:

---

```

>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(6):
... nprint(chop(taylor(lambda x: bernpoly(n,x), 0, n)))
...
[1.0]
[-0.5, 1.0]

```

```
[0.166667, -1.0, 1.0]
[0.0, 0.5, -1.5, 1.0]
[-0.0333333, 0.0, 1.0, -2.0, 1.0]
[0.0, -0.166667, 0.0, 1.66667, -2.5, 1.0]
```

---

Evaluation is accurate for large and small  $z$ :

```
>>> mp.dps = 25
>>> bernpoly(100, 0.5)
2.838224957069370695926416e+78
>>> bernpoly(1000, 10.5)
5.318704469415522036482914e+1769
```

---

## 14.3 Euler numbers and polynomials

### 14.3.1 Euler numbers

---

#### Function **eulernum(*n* As mpNum) As mpNum**

---

The function `eulernum` returns the  $n$ -th Euler number

**Parameter:**

*n*: An integer

The  $n$ -th Euler number is defined as the  $n$ -th derivative of  $\text{sech}(t) = 1/\cosh(t)$  evaluated at  $t = 0$ . Equivalently, the Euler numbers give the coefficients of the Taylor series

$$\text{sech}(t) = \sum_{n=0}^{\infty} \frac{E_n}{n!} t^n. \quad (14.3.1)$$

The Euler numbers are closely related to Bernoulli numbers and Bernoulli polynomials. They can also be evaluated in terms of Euler polynomials (see `eulerpoly()`) as  $E_n = 2^n E_n(1/2)$ .

Examples

Euler numbers grow very rapidly. `eulernum()` efficiently computes numerical approximations for large indices:

---

```
>>> eulernum(50)
-6.053285248188621896314384e+54
>>> eulernum(1000)
3.887561841253070615257336e+2371
>>> eulernum(10**20)
4.346791453661149089338186e+1936958564106659551331
```

---

Pass `exact=True` to obtain exact values of Euler numbers as integers:

---

```
>>> print(eulernum(50, exact=True))
-6053285248188621896314383785111649088103498225146815121
>>> print(eulernum(200, exact=True) % 10**10)
1925859625
>>> eulernum(1001, exact=True)
0
```

---

### 14.3.2 Euler polynomials

---

#### Function **eulerpoly(*n* As mpNum, *z* As mpNum) As mpNum**

---

The function `eulerpoly` returns the Euler polynomial  $E_n(z)$

**Parameters:**

*n*: A real or complex number.

*z*: A real or complex number.

The Euler polynomial  $E_n(z)$  is defined by the generating function representation

$$\frac{2e^{zt}}{e^t + 1} = \sum_{n=0}^{\infty} E_n(z) \frac{t^n}{n!}. \quad (14.3.2)$$

The Euler polynomials may also be represented in terms of Bernoulli polynomials (see `bernpoly()`) using various formulas, for example

$$En(z) = \frac{2}{n+1} \left( B_n(z) - 2^{n+2} B_n(z/2) \right) \quad (14.3.3)$$

Special values include the Euler numbers  $E_n = 2^{n+1} E_n(1/2)$  (see `eulernum()`).

Examples

Evaluation for arbitrary  $z$ :

---

```
>>> eulerpoly(2,3)
6.0
>>> eulerpoly(5,4)
423.5
>>> eulerpoly(35, 11111111112)
3.994957561486776072734601e+351
>>> eulerpoly(4, 10+20j)
(-47990.0 - 235980.0j)
>>> eulerpoly(2, '-3.5e-5')
0.000035001225
>>> eulerpoly(3, 0.5)
0.0
>>> eulerpoly(55, -10**80)
-1.0e+4400
>>> eulerpoly(5, -inf)
-inf
>>> eulerpoly(6, -inf)
+inf
```

---

## 14.4 Bell numbers and polynomials

---

Function **bell(*n* As mpNum, *x* As mpNum) As mpNum**

---

The function **bell** returns the Bell polynomial  $B_n(x)$

**Parameters:**

*n*: A non-negative integer.

*x*: A real or complex number.

The Bell polynomial  $B_n(x)$  are feinde for  $n > 0$ . The first few are

$$B_0(x) = 1; \quad B_1(x) = x; \quad B_2(x) = x^2 + x; \quad B_3(x) = x^3 + 3x^2 + x. \quad (14.4.1)$$

If  $x = 1$  or **bell()** is called with only one argument, it gives the  $n$ -th Bell number  $B_n$ , which is the number of partitions of a set with  $n$  elements. By setting the precision to at least  $\log_{10} B_n$  digits, **bell()** provides fast calculation of exact Bell numbers.

In general, **bell()** computes

$$B_n(x) = e^{-x} (\text{sinc}(\pi n) + E_n(x)) \quad (14.4.2)$$

where  $E_n(x)$  is the generalized exponential function implemented by **polyexp()**. This is an extension of Dobinski's formula [1], where the modification is the sinc term ensuring that  $B_n(x)$  is continuous in  $n$  ; **bell()** can thus be evaluated, differentiated, etc for arbitrary complex arguments.

Examples

Simple evaluations:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> bell(0, 2.5)
1.0
>>> bell(1, 2.5)
2.5
>>> bell(2, 2.5)
8.75
```

---

Evaluation for arbitrary complex arguments:

---

```
>>> bell(5.75+1j, 2-3j)
(-10767.71345136587098445143 - 15449.55065599872579097221j)
```

---

## 14.5 Stirling numbers

### 14.5.1 Stirling number of the first kind

---

Function **stirling1**(*n* As mpNum, *k* As mpNum, **Keywords** As String) As mpNum

---

The function **stirling1** returns the Stirling number of the first kind  $s(n, k)$

**Parameters:**

*n*: A real or complex number.

*k*: A real or complex number.

**Keywords**: exact=False.

The Stirling number of the first kind  $s(n, k)$  is defined by

$$x(x-1)(x-2)\cdots(x-n+1) = \sum_{k=0}^n s(n, k)x^k. \quad (14.5.1)$$

The value is computed using an integer recurrence. The implementation is not optimized for approximating large values quickly.

Examples

Comparing with the generating function:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> taylor(lambda x: ff(x, 5), 0, 5)
[0.0, 24.0, -50.0, 35.0, -10.0, 1.0]
>>> [stirling1(5, k) for k in range(6)]
[0.0, 24.0, -50.0, 35.0, -10.0, 1.0]
```

---

Pass exact=True to obtain exact values of Stirling numbers as integers:

---

```
>>> stirling1(42, 5)
-2.864498971768501633736628e+50
>>> print stirling1(42, 5, exact=True)
-286449897176850163373662803014001546235808317440000
```

---

### 14.5.2 Stirling number of the second kind

---

Function **stirling2**(*n* As mpNum, *k* As mpNum, **Keywords** As String) As mpNum

---

The function **stirling2** returns the Stirling number of the second kind  $s(n, k)$

**Parameters:**

*n*: A real or complex number.

*k*: A real or complex number.

**Keywords**: exact=False.

The Stirling number of the second kind  $S(n, k)$  is defined by

$$x^n = \sum_{k=0}^n S(n, k)x(x-1)(x-2)\cdots(x-k+1). \quad (14.5.2)$$

The value is computed using integer arithmetic to evaluate a power sum. The implementation is not optimized for approximating large values quickly.

Examples

Comparing with the generating function:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> taylor(lambda x: sum(stirling2(5,k) * ff(x,k) for k in range(
[0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
```

---

Pass exact=True to obtain exact values of Stirling numbers as integers:

---

```
>>> stirling2(52, 10)
2.641822121003543906807485e+45
>>> print stirling2(52, 10, exact=True)
2641822121003543906807485307053638921722527655
```

---

## 14.6 Prime counting functions

### 14.6.1 Exact prime counting function

---

#### Function **primepi(*x* As mpNum) As mpNum**

---

The function primepi returns the prime counting function

**Parameter:**

*x*: A real number

The prime counting function,  $\pi(x)$ , gives the number of primes less than or equal to  $x$ . The argument  $x$  may be fractional.

The prime counting function is very expensive to evaluate precisely for large  $x$ , and the present implementation is not optimized in any way. For numerical approximation of the prime counting function, it is better to use primepi2() or riemannr().

Some values of the prime counting function:

---

```
>>> from mpFormulaPy import *
>>> [primepi(k) for k in range(20)]
[0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7, 8]
>>> primepi(3.5)
2
>>> primepi(100000)
9592
```

---

### 14.6.2 Prime counting function interval

---

#### Function **primepi2(*x* As mpNum) As mpNum**

---

The function primepi2 returns an interval (as an mpi instance) providing bounds for the value of the prime counting function  $\pi(x)$

**Parameter:**

*x*: A real number

For small  $x$ , primepi2() returns an exact interval based on the output of primepi(). For  $x > 2656$ , a loose interval based on Schoenfeld's inequality

$$|\pi(x)| - \text{li}(x) < \frac{\sqrt{x} \log x}{8\pi} \quad (14.6.1)$$

is returned. This estimate is rigorous assuming the truth of the Riemann hypothesis, and can be computed very quickly.

Examples

Exact values of the prime counting function for small  $x$ :

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> iv.dps = 15; iv.pretty = True
>>> primepi2(10)
[4.0, 4.0]
```

---

```
>>> primepi2(100)
[25.0, 25.0]
>>> primepi2(1000)
[168.0, 168.0]
```

---

Loose intervals are generated for moderately large  $x$ :

---

```
>>> primepi2(10000), primepi(10000)
([1209.0, 1283.0], 1229)
>>> primepi2(50000), primepi(50000)
([5070.0, 5263.0], 5133)
```

---

As  $x$  increases, the absolute error gets worse while the relative error improves. The exact value of  $\pi(10^{23})$  is 1925320391606803968923, and primepi2() gives 9 significant digits:

---

```
>>> p = primepi2(10**23)
>>> p
[1.9253203909477020467e+21, 1.925320392280406229e+21]
>>> mpf(p.delta) / mpf(p.a)
6.9219865355293e-10
```

---

A more precise, nonrigorous estimate for  $\pi(x)$  can be obtained using the Riemann R function (riemannr()). For large enough  $x$ , the value returned by primepi2() essentially amounts to a small perturbation of the value returned by riemannr():

---

```
>>> primepi2(10**100)
[4.3619719871407024816e+97, 4.3619719871407032404e+97]
>>> riemannr(10**100)
4.3619719871407e+97
```

---

### 14.6.3 Riemann R function

---

Function **riemannr( $x$  As mpNum)** As mpNum

---

The function **riemannr** returns the Riemann R function, a smooth approximation of the prime counting function  $\pi(x)$

**Parameter:**

$x$ : A real number

The Riemann R function gives a fast numerical approximation useful e.g. to roughly estimate the number of primes in a given interval (see primepi()).

The Riemann R function is computed using the rapidly convergent Gram series,

$$R(x) = 1 + \sum_{k=1}^{\infty} \frac{\log^k x}{kk!\zeta(k+1)}. \quad (14.6.2)$$

From the Gram series, one sees that the Riemann R function is a well-defined analytic function (except for a branch cut along the negative real half-axis); it can be evaluated for arbitrary real or complex arguments.

The Riemann R function gives a very accurate approximation of the prime counting function. For example, it is wrong by at most 2 for  $x < 1000$ , and for  $x = 10^9$  differs from the exact value

of  $\pi(x)$  by 79, or less than two parts in a million. It is about 10 times more accurate than the logarithmic integral estimate (see `li()`), which however is even faster to evaluate. It is orders of magnitude more accurate than the extremely fast  $x/\log x$  estimate.

For small arguments, the Riemann R function almost exactly gives the prime counting function if rounded to the nearest integer:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> primepi(50), riemannr(50)
(15, 14.9757023241462)
>>> max(abs(primepi(n)-int(round(riemannr(n)))) for n in range(100))
1
>>> max(abs(primepi(n)-int(round(riemannr(n)))) for n in range(300))
2
```

---

The Riemann R function can be evaluated for arguments far too large for exact determination of  $\pi(x)$  to be computationally feasible with any presently known algorithm:

---

```
>>> riemannr(10**30)
1.46923988977204e+28
>>> riemannr(10**100)
4.3619719871407e+97
>>> riemannr(10**1000)
4.3448325764012e+996
```

---

Evaluation is supported for arbitrary arguments and at arbitrary precision:

---

```
>>> mp.dps = 30
>>> riemannr(7.5)
3.72934743264966261918857135136
>>> riemannr(-4+2j)
(-0.551002208155486427591793957644 + 2.16966398138119450043195899
```

---

## 14.7 Miscellaneous functions

### 14.7.1 Cyclotomic polynomials

---

Function **cyclotomic(*n* As mpNum, *x* As mpNum)** As mpNum

---

The function `cyclotomic` returns the cyclotomic polynomial  $\Phi_n(x)$

**Parameters:**

- n*: A real or complex number.
- x*: A real or complex number.

The cyclotomic polynomial  $\Phi_n(x)$  is defined by

$$\Phi_n(x) = \prod_{\zeta} (x - \zeta) \quad (14.7.1)$$

where  $\zeta$  ranges over all primitive  $n$ -th roots of unity (see `unitroots()`). An equivalent representation, used for computation, is

$$\Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)} \quad (14.7.2)$$

where  $\mu(m)$  denotes the Moebius function. The cyclotomic polynomials are integer polynomials, the first of which can be written explicitly as

$$\Phi_0(x) = 1; \quad \Phi_1(x) = x - 1; \quad \Phi_2(x) = x + 1; \quad \Phi_3(x) = x^3 + x^2 + 1 \quad (14.7.3)$$

$$\Phi_4(x) = x^2 + 1; \quad \Phi_5(x) = x^4 + x^3 + x^2 + x + 1; \quad \Phi_6(x) = x^2 - x + 1. \quad (14.7.4)$$

The coefficients of low-order cyclotomic polynomials can be recovered using Taylor expansion:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> for n in range(9):
... p = chop(taylor(lambda x: cyclotomic(n,x), 0, 10))
... print("%s %s" % (n, nstr(p[:10+1-p[::-1].index(1)])))
...
0 [1.0]
1 [-1.0, 1.0]
2 [1.0, 1.0]
3 [1.0, 1.0, 1.0]
4 [1.0, 0.0, 1.0]
5 [1.0, 1.0, 1.0, 1.0, 1.0]
6 [1.0, -1.0, 1.0]
7 [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
8 [1.0, 0.0, 0.0, 0.0, 1.0]
```

---

### 14.7.2 von Mangoldt function

---

Function **mangoldt**(*n* As mpNum) As mpNum

---

The function **mangoldt** returns the von Mangoldt function

**Parameter:**

*n*: An integer

The von Mangoldt function is defined as  $\Lambda(n) = \log p$  if  $n = p^k$  is a power of a prime, and  $\Lambda(n) = 0$  otherwise.

Examples

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> [mangoldt(n) for n in range(-2,3)]
[0.0, 0.0, 0.0, 0.0, 0.6931471805599453094172321]
>>> mangoldt(6)
0.0
>>> mangoldt(7)
1.945910149055313305105353
>>> mangoldt(8)
0.6931471805599453094172321
>>> fsum(mangoldt(n) for n in range(101))
94.04531122935739224600493
>>> fsum(mangoldt(n) for n in range(10001))
10013.39669326311478372032
```

---

# Chapter 15

## q-functions

### 15.1 q-Pochhammer symbol

---

Function **qp**(*a* As mpNum, *q* As mpNum, *n* As mpNum) As mpNum

---

The function **qp** returns the q-Pochhammer symbol (or q-rising factorial)

**Parameters:**

*a*: A real or complex number.

*q*: A real or complex number.

*n*: An integer.

The q-Pochhammer symbol (or q-rising factorial) is defined as

$$(a; q)_n = \prod_{k=0}^{n-1} (1 - aq^k) \quad (15.1.1)$$

where  $n = \infty$  is permitted if  $|q| < 1$ . Called with two arguments, **qp**(*a*,*q*) computes  $(a; q)_\infty$ ; with a single argument, **qp**(*q*) computes  $(q; q)_\infty$ . The special case

$$\phi(q) = (q; q)_\infty = \prod_{k=1}^{\infty} (1 - q^k) = \sum_{k=-\infty}^{\infty} (-1)^k q^{(3k^2-k)/2} \quad (15.1.2)$$

is also known as the Euler function, or (up to a factor  $q^{-1/24}$ ) the Dedekind eta function.

Examples

If *n* is a positive integer, the function amounts to a finite product:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> qp(2,3,5)
-725305.0
>>> fprod(1-2*3**k for k in range(5))
-725305.0
>>> qp(2,3,0)
1.0
```

---

Complex arguments are allowed:

---

```
>>> qp(2-1j, 0.75j)
(0.4628842231660149089976379 + 4.481821753552703090628793j)
```

---

## 15.2 q-gamma and factorial

### 15.2.1 q-gamma

---

Function **qgamma(z As mpNum, q As mpNum)** As mpNum

---

The function `qgamma` returns the q-gamma function

**Parameters:**

*z*: A real or complex number.

*q*: A real or complex number.

The q-gamma function is defined as

$$\Gamma_q(z) = \frac{(q; q)_\infty}{(q^z; q)_\infty} (1 - q)^{1-z}. \quad (15.2.1)$$

Examples

Evaluation for real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> qgamma(4,0.75)
4.046875
>>> qgamma(6,6)
121226245.0
>>> qgamma(3+4j, 0.5j)
(0.1663082382255199834630088 + 0.01952474576025952984418217j)
```

---

### 15.2.2 q-factorial

---

Function **qfac(z As mpNum, q As mpNum)** As mpNum

---

The function `qfac` returns the q-factorial

**Parameters:**

*z*: A real or complex number.

*q*: A real or complex number.

The q-factorial is defined as

$$[n]_q! = (1 + q)(1 + q + q^2) \cdots (1 + q + \cdots + q^{n-1}) \quad (15.2.2)$$

or more generally

$$[z]_q! = \frac{(q; q)_z}{(1 - q)^z} \quad (15.2.3)$$

Examples

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> qfac(0,0)
1.0
```

---

```
>>> qfac(4,3)
2080.0
>>> qfac(5,6)
121226245.0
>>> qfac(1+1j, 2+1j)
(0.4370556551322672478613695 + 0.2609739839216039203708921j)
```

---

## 15.3 Hypergeometric q-series

---

Function **qhyper(as As mpNum, bs As mpNum, q As mpNum, z As mpNum)** As mpNum

---

The function `qhyper` returns the hypergeometric q-series

**Parameters:**

*as*: A real or complex number.

*bs*: A real or complex number.

*q*: A real or complex number.

*z*: A real or complex number.

The basic hypergeometric series or hypergeometric q-series is defined as

$$\text{qhyper}(A, B, q, z) = \sum_{n=0}^{\infty} \frac{(a_1; q)_n, \dots, (a_r; q)_n}{(b_1; q)_n, \dots, (b_s; q)_n} \left( (-1)^n q^{\binom{n}{2}} \right)^{1+s-r} \frac{z^n}{(q; q)_n} \quad (15.3.1)$$

where  $(a; q)_n$  denotes the q-Pochhammer symbol (see `qp()`).

Examples

Evaluation works for real and complex arguments:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> qhyper([0.5], [2.25], 0.25, 4)
-0.1975849091263356009534385
>>> qhyper([0.5], [2.25], 0.25-0.25j, 4)
(2.806330244925716649839237 + 3.568997623337943121769938j)
>>> qhyper([1+j], [2,3+0.5j], 0.25, 3+4j)
(9.112885171773400017270226 - 1.272756997166375050700388j)
```

---

# Chapter 16

## Matrix functions

### 16.1 Matrix exponential

---

Function **expm(A As mpNum, Keywords As String)** As mpNum

---

The function `expm` returns the matrix exponential of a square matrix  $A$

**Parameters:**

$A$ : A real or complex matrix.

**Keywords:** `method='taylor'`.

The matrix exponential of a square matrix  $A$  is defined by the power series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots \quad (16.1.1)$$

With `method='taylor'`, the matrix exponential is computed using the Taylor series. With `method='pade'`, Pade approximants are used instead.

**Examples**

Basic examples:

---

```
>>> from mpFormulaPy import *
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> expm(zeros(3))
[1.0 0.0 0.0]
[0.0 1.0 0.0]
[0.0 0.0 1.0]
>>> expm(eye(3))
[2.71828182845905          0.0          0.0]
[          0.0 2.71828182845905          0.0]
[          0.0          0.0 2.71828182845905]
>>> expm([[1,1,0],[1,0,1],[0,1,0]])
[ 3.86814500615414 2.26812870852145 0.841130841230196]
[ 2.26812870852145 2.44114713886289 1.42699786729125]
[0.841130841230196 1.42699786729125 1.6000162976327]
>>> expm([[1,1,0],[1,0,1],[0,1,0]], method='pade')
[ 3.86814500615414 2.26812870852145 0.841130841230196]
[ 2.26812870852145 2.44114713886289 1.42699786729125]
```

---

```
[0.841130841230196 1.42699786729125 1.6000162976327]
>>> expm([[1+j, 0], [1+j,1]])
[(1.46869393991589 + 2.28735528717884j) 0.0]
[ (1.03776739863568 + 3.536943175722j) (2.71828182845905 + 0.0j)]
```

---

Matrices with large entries are allowed:

---

```
>>> expm(matrix([[1,2],[2,3]])**25)
[5.65024064048415e+2050488462815550 9.14228140091932e+2050488462815550]
[9.14228140091932e+2050488462815550 1.47925220414035e+2050488462815551]
```

---

The identity  $\exp(A + B) = \exp(A) \exp(B)$  does not hold for noncommuting matrices:

---

```
>>> A = hilbert(3)
>>> B = A + eye(3)
>>> chop(mnorm(A*B - B*A))
0.0
>>> chop(mnorm(expm(A+B) - expm(A)*expm(B)))
0.0
>>> B = A + ones(3)
>>> mnorm(A*B - B*A)
1.8
>>> mnorm(expm(A+B) - expm(A)*expm(B))
42.0927851137247
```

---

## 16.2 Matrix cosine

---

Function **cosm(A As mpNum) As mpNum**

---

The function **cosm** returns the matrix cosine of a square matrix  $A$

**Parameter:**

$A$ : A real or complex matrix.

The cosine of a square matrix  $A$  is defined in analogy with the matrix exponential.

Examples:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> cosm(X)
[0.54030230586814 0.0 0.0]
[ 0.0 0.54030230586814 0.0]
[ 0.0 0.0 0.54030230586814]
>>> X = hilbert(3)
>>> cosm(X)
[ 0.424403834569555 -0.316643413047167 -0.221474945949293]
[-0.316643413047167 0.820646708837824 -0.127183694770039]
[-0.221474945949293 -0.127183694770039 0.909236687217541]
>>> X = matrix([[1+j, -2], [0, -j]])
>>> cosm(X)
[(0.833730025131149 - 0.988897705762865j) (1.07485840848393 - 0.17192140544213j)]
[ 0.0 (1.54308063481524 + 0.0j)]
```

---

## 16.3 Matrix sine

---

Function **sinm(A As mpNum) As mpNum**

---

The function **sinm** returns the matrix sine of a square matrix  $A$

**Parameter:**

$A$ : A real or complex matrix.

The sine of a square matrix  $A$  is defined in analogy with the matrix exponential.

Examples:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> sinm(X)
[0.841470984807897 0.0 0.0]
[ 0.0 0.841470984807897 0.0]
[ 0.0 0.0 0.841470984807897]
>>> X = hilbert(3)
>>> sinm(X)
[0.711608512150994 0.339783913247439 0.220742837314741]
[0.339783913247439 0.244113865695532 0.187231271174372]
[0.220742837314741 0.187231271174372 0.155816730769635]
>>> X = matrix([[1+j,-2],[0,-j]])
>>> sinm(X)
[(1.29845758141598 + 0.634963914784736j) (-1.96751511930922 + 0.314700021761367j)]
[ 0.0 (0.0 - 1.1752011936438j)]
```

---

## 16.4 Matrix square root

---

Function **sqrtm**(*A* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function **sqrtm** returns a square root of a square matrix *A*

**Parameters:**

*A*: A real or complex matrix.

**Keywords:** mayrotate=2.

A square root of the square matrix *A* is a matrix *B* =  $A^{1/2}$  such that  $B^2 = A$ . The square root of a matrix, if it exists, is not unique

**Examples:**

Square roots of some simple matrices:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> sqrtm([[1,0], [0,1]])
[1.0  0.0]
[0.0  1.0]
>>> sqrtm([[0,0], [0,0]])
[0.0  0.0]
[0.0  0.0]
>>> sqrtm([[2,0],[0,1]])
[1.4142135623731 0.0]
[      0.0  1.0]
>>> sqrtm([[1,1],[1,0]])
[(0.920442065259926 - 0.21728689675164j) (0.568864481005783 + 0.351577584254143j)]
[(0.568864481005783 + 0.351577584254143j) (0.351577584254143 - 0.568864481005783j)]
>>> sqrtm([[1,0],[0,1]])
[1.0  0.0]
[0.0  1.0]
>>> sqrtm([[-1,0],[0,1]])
[(0.0 - 1.0j)      0.0]
[      0.0 (1.0 + 0.0j)]
>>> sqrtm([[j,0],[0,j]])
[(0.707106781186547 + 0.707106781186547j)      0.0]
[      0.0 (0.707106781186547 + 0.707106781186547j)]
```

---

A square root of a rotation matrix, giving the corresponding half-angle rotation matrix:

---

```
>>> t1 = 0.75
>>> t2 = t1 * 0.5
>>> A1 = matrix([[cos(t1), -sin(t1)], [sin(t1), cos(t1)]])
>>> A2 = matrix([[cos(t2), -sin(t2)], [sin(t2), cos(t2)]])
>>> sqrtm(A1)
[0.930507621912314 -0.366272529086048]
[0.366272529086048 0.930507621912314]
>>> A2
[0.930507621912314 -0.366272529086048]
[0.366272529086048 0.930507621912314]
```

---

The identity  $(A^2)^{1/2}$  does not necessarily hold:

---

```
>>> A = matrix([[4,1,4],[7,8,9],[10,2,11]])
>>> sqrtm(A**2)
[ 4.0 1.0 4.0]
[ 7.0 8.0 9.0]
[10.0 2.0 11.0]
>>> sqrtm(A)**2
[ 4.0 1.0 4.0]
[ 7.0 8.0 9.0]
[10.0 2.0 11.0]
>>> A = matrix([[-4,1,4],[7,-8,9],[10,2,11]])
>>> sqrtm(A**2)
[ 7.43715112194995 -0.324127569985474 1.8481718827526]
[-0.251549715716942 9.32699765900402 2.48221180985147]
[ 4.11609388833616 0.775751877098258 13.017955697342]
>>> chop(sqrtm(A)**2)
[-4.0 1.0 4.0]
[ 7.0 -8.0 9.0]
[10.0 2.0 11.0]
```

---

For some matrices, a square root does not exist:

---

```
>>> sqrtm([[0,1], [0,0]])
Traceback (most recent call last):
...
ZeroDivisionError: matrix is numerically singular
```

---

Two examples from the documentation for Matlab's sqrtm:

---

```
>>> mp.dps = 15; mp.pretty = True
>>> sqrtm([[7,10],[15,22]])
[1.56669890360128 1.74077655955698]
[2.61116483933547 4.17786374293675]
>>>
>>> X = matrix(\
... [[5,-4,1,0,0],\
... [-4,6,-4,1,0],\
... [1,-4,6,-4,1],\
... [0,1,-4,6,-4],\
... [0,0,1,-4,5]])
>>> Y = matrix(\
... [[2,-1,-0,-0,-0],\
... [-1,2,-1,0,-0],\
... [0,-1,2,-1,0],\
... [-0,0,-1,2,-1],\
... [-0,-0,-0,-1,2]])
>>> mnorm(sqrtm(X) - Y)
4.53155328326114e-19
```

---

## 16.5 Matrix logarithm

---

### Function **logm(A As mpNum) As mpNum**

---

The function **logm** returns the matrix logarithm of a square matrix  $A$

**Parameter:**

$A$ : A real or complex matrix.

A logarithm of the square matrix  $A$  is a matrix  $B$  such that  $\exp(B) = A$ . The logarithm of a matrix, if it exists, is not unique.

**Examples:**

Logarithms of some simple matrices:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> X = eye(3)
>>> logm(X)
[[0.0 0.0 0.0]
 [0.0 0.0 0.0]
 [0.0 0.0 0.0]
>>> logm(2*X)
[[0.693147180559945 0.0 0.0]
 [0.0 0.693147180559945 0.0]
 [0.0 0.0 0.693147180559945]
>>> logm(expm(X))
[[1.0 0.0 0.0]
 [0.0 1.0 0.0]
 [0.0 0.0 1.0]]
```

---

A logarithm of a complex matrix:

---

```
>>> X = matrix([[2+j, 1, 3], [1-j, 1-2*j, 1], [-4, -5, j]])
>>> B = logm(X)
>>> nprint(B)
[[0.808757 + 0.107759j, 2.20752 + 0.202762j, 1.07376 - 0.773874j]
 [0.905709 - 0.107795j, 0.0287395 - 0.824993j, 0.111619 + 0.514272j]
 [(-0.930151 + 0.399512j), (-2.06266 - 0.674397j), (0.791552 + 0.519839j)]
>>> chop(expm(B))
[[2.0 + 1.0j, 1.0, 3.0]
 [(1.0 - 1.0j) (1.0 - 2.0j), 1.0]
 [-4.0, -5.0 (0.0 + 1.0j)]]
```

---

A matrix  $X$  close to the identity matrix, for which  $\log(\exp(X)) = \exp(\log(X)) = X$  holds:

---

```
>>> X = eye(3) + hilbert(3)/4
>>> X
[[1.25, 0.125, 0.0833333333333333]
 [0.125, 1.083333333333333, 0.0625]
 [0.0833333333333333, 0.0625, 1.05]
>>> logm(expm(X))
[[1.25, 0.125, 0.0833333333333333]
 [0.125, 1.083333333333333, 0.0625]]
```

---

---

```
[0.0833333333333333 0.0625 1.05]
>>> expm(logm(X))
[ 1.25 0.125 0.0833333333333333]
[ 0.125 1.083333333333333 0.0625]
[0.0833333333333333 0.0625 1.05]
```

---

A logarithm of a rotation matrix, giving back the angle of the rotation:

---

```
>>> t = 3.7
>>> A = matrix([[cos(t),sin(t)],[-sin(t),cos(t)]])
>>> chop(logm(A))
[ 0.0 -2.58318530717959]
[2.58318530717959 0.0]
>>> (2*pi-t)
2.58318530717959
```

---

For some matrices, a logarithm does not exist:

---

```
>>> logm([[1,0], [0,0]])
Traceback (most recent call last):
...
ZeroDivisionError: matrix is numerically singular
```

---

Logarithm of a matrix with large entries:

---

```
>>> logm(hilbert(3) * 10**20).apply(re)
[ 45.5597513593433 1.27721006042799 0.317662687717978]
[ 1.27721006042799 42.5222778973542 2.24003708791604]
[0.317662687717978 2.24003708791604 42.395212822267]
```

---

## 16.6 Matrix power

---

Function **powm**(*A* As *mpNum*, *r* As *mpNum*) As *mpNum*

---

The function **powm** returns  $A^r = \exp(A \log r)$  for a matrix *A* and complex number *r*

**Parameters:**

*A*: A real or complex matrix.

*r*: A real or complex number.

Computes  $A^r = \exp(A \log r)$  for a matrix *A* and complex number *r*.

**Examples**

Powers and inverse powers of a matrix:

---

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> A = matrix([[4,1,4],[7,8,9],[10,2,11]])
>>> powm(A, 2)
[ 63.0 20.0 69.0]
[174.0 89.0 199.0]
[164.0 48.0 179.0]
>>> chop(powm(powm(A, 4), 1/4.))
[ 4.0 1.0 4.0]
[ 7.0 8.0 9.0]
[10.0 2.0 11.0]
>>> powm(extraprec(20)(powm)(A, -4), -1/4.)
[ 4.0 1.0 4.0]
[ 7.0 8.0 9.0]
[10.0 2.0 11.0]
>>> chop(powm(powm(A, 1+0.5j), 1/(1+0.5j)))
[ 4.0 1.0 4.0]
[ 7.0 8.0 9.0]
[10.0 2.0 11.0]
>>> powm(extraprec(5)(powm)(A, -1.5), -1/(1.5))
[ 4.0 1.0 4.0]
[ 7.0 8.0 9.0]
[10.0 2.0 11.0]
```

---

A Fibonacci-generating matrix:

---

```
>>> powm([[1,1],[1,0]], 10)
[89.0 55.0]
[55.0 34.0]
>>> fib(10)
55.0
>>> powm([[1,1],[1,0]], 6.5)
[(16.5166626964253 - 0.0121089837381789j) (10.2078589271083 + 0.0195927472575932j)]
[(10.2078589271083 + 0.0195927472575932j) (6.30880376931698 - 0.0317017309957721j)]
>>> (phi**6.5 - (1-phi)**6.5)/sqrt(5)
(10.2078589271083 - 0.0195927472575932j)
>>> powm([[1,1],[1,0]], 6.2)
[ (14.3076953002666 - 0.008222855781077j) (8.81733464837593 + 0.0133048601383712j)]
```

```
[(8.81733464837593 + 0.0133048601383712j) (5.49036065189071 - 0.0215277159194482j)]  
>>> (phi**6.2 - (1-phi)**6.2)/sqrt(5)  
(8.81733464837593 - 0.0133048601383712j)
```

---

# Chapter 17

## Eigensystems and related Decompositions

### 17.1 Singular value decomposition

---

Function **svd**(*A* As *mpNum*, *Keywords* As *String*) As *mpNum*

---

The function **svd** returns the singular value decomposition of matrix *A*

**Parameters:**

*A*: A real or complex number.

*Keywords*: *compute\_uv* = True.

The routines **svd\_r** and **svd\_c** compute the singular value decomposition of a real or complex matrix *A*. **svd** is an unified interface calling either **svd\_r** or **svd\_c** depending on whether *A* is real or complex.

Given *A*, two orthogonal (*A* real) or unitary (*A* complex) matrices *U* and *V* are calculated such that

$$A = USV; \quad U'U = 1; \quad VV' = 1, \quad (17.1.1)$$

where *S* is a suitable shaped matrix whose off-diagonal elements are zero. Here ' denotes the hermitian transpose (i.e. transposition and complex conjugation). The diagonal elements of *S* are the singular values of *A*, i.e. the square roots of the eigenvalues of  $A'A$  or  $AA'$ .

Examples:

---

```
>>> from mpFormulaPy import mp
>>> A = mp.matrix([[2, -2, -1], [3, 4, -2], [-2, -2, 0]])
>>> S = mp.svd_r(A, compute_uv = False)
>>> print S
[6.0]
[3.0]
[1.0]
>>> U, S, V = mp.svd_r(A)
>>> print mp.chop(A - U * mp.diag(S) * V)
[0.0 0.0 0.0]
[0.0 0.0 0.0]
[0.0 0.0 0.0]
```

---

## 17.2 The Schur decomposition

---

Function **schur**(*A* As *mpNum*) As *mpNum*

---

The function **schur** returns the Schur decomposition of a square matrix *A*

**Parameter:**

*A*: A real or complex matrix.

This function computes the Schur decomposition of a square matrix *A*. Given *A*, a unitary matrix *Q* is determined such that

$$Q' A Q = R; \quad Q' Q = Q Q' = 1, \quad (17.2.1)$$

where *R* is an upper right triangular matrix. Here ' denotes the hermitian transpose (i.e. transposition and conjugation).

Examples:

---

```
>>> from mpFormulaPy import mp
>>> A = mp.matrix([[3, -1, 2], [2, 5, -5], [-2, -3, 7]])
>>> Q, R = mp.schur(A)
>>> mp.nprint(R, 3)
[2.0 0.417 -2.53]
[0.0 4.0 -4.74]
[0.0 0.0 9.0]
>>> print(mp.chop(A - Q * R * Q.transpose_conj()))
[0.0 0.0 0.0]
[0.0 0.0 0.0]
[0.0 0.0 0.0]
```

---

## 17.3 The eigenvalue problem

---

Function **eig**(*A* As *mpNum*, *Keywords* As *String*) As *mpNum*

---

The function **eig** returns the solution of the (ordinary) eigenvalue problem for a real or complex square matrix *A*

**Parameters:**

*A*: A real or complex number.

*Keywords*: *left* = *False*, *right* = *False*.

The routine **eig** solves the (ordinary) eigenvalue problem for a real or complex square matrix *A*. Given *A*, a vector *E* and matrices *ER* and *EL* are calculated such that

---

```
A ER[:,i] = E[i] ER[:,i]
EL[i,:] A = EL[i,:] E[i]
```

---

*E* contains the eigenvalues of *A*. The columns of *ER* contain the right eigenvectors of *A* whereas the rows of *EL* contain the left eigenvectors.

**Examples**

---

```
>>> from mpFormulaPy import mp
>>> A = mp.matrix([[3, -1, 2], [2, 5, -5], [-2, -3, 7]])
>>> E, ER = mp.eig(A)
>>> print(mp.chop(A * ER[:,0] - E[0] * ER[:,0]))
[0.0]
[0.0]
[0.0]
>>> E, EL, ER = mp.eig(A, left = True, right = True)
>>> E, EL, ER = mp.eig_sort(E, EL, ER)
>>> mp.nprint(E)
[2.0, 4.0, 9.0]
>>> print(mp.chop(A * ER[:,0] - E[0] * ER[:,0]))
[0.0]
[0.0]
[0.0]
>>> print(mp.chop(EL[0,:] * A - EL[0,:] * E[0]))
[0.0 0.0 0.0]
```

---

## 17.4 The symmetric eigenvalue problem

---

Function **eigh**(*A* As *mpNum*, **Keywords** As *String*) As *mpNum*

---

The function **eigh** returns the solution of the (ordinary) eigenvalue problem for a real symmetric or complex hermitian square matrix *A*

**Parameters:**

*A*: A real or complex number.

*Keywords*: *eigvals\_only* = False.

The routines **eigsy** and **eighe** solve the (ordinary) eigenvalue problem for a real symmetric or complex hermitian square matrix *A*. **eigh** is an unified interface for these two functions calling either **eigsy** or **eighe** depending on whether *A* is real or complex.

Given *A*, an orthogonal (*A* real) or unitary matrix *Q* (*A* complex) is calculated which diagonalizes *A*:

$$Q' A Q = \text{diag}(E); \quad Q' Q = Q' Q = 1. \quad (17.4.1)$$

Here  $\text{diag}(E)$  is a diagonal matrix whose diagonal is *E*.  $'$  denotes the hermitian transpose (i.e. ordinary transposition and complex conjugation).

The columns of *Q* are the eigenvectors of *A* and *E* contains the eigenvalues:

---

*A* *Q*[:,*i*] = *E*[*i*] *Q*[:,*i*]

---

Examples:

---

```
>>> from mpFormulaPy import mp
>>> A = mp.matrix([[3, 2], [2, 0]])
>>> E = mp.eigsy(A, eigvals_only = True)
>>> print E
[-1.0]
[ 4.0]
>>> A = mp.matrix([[1, 2], [2, 3]])
>>> E, Q = mp.eigsy(A)           # alternative: E, Q = mp.eigh(A)
>>> print mp.chop(A * Q[:,0] - E[0] * Q[:,0])
[0.0]
[0.0]
>>> A = mp.matrix([[1, 2 + 5j], [2 - 5j, 3]])
>>> E, Q = mp.eighe(A)          # alternative: E, Q = mp.eigh(A)
>>> print mp.chop(A * Q[:,0] - E[0] * Q[:,0])
[0.0]
[0.0]
```

---

# Part IV

## GMP and related libraries

# Chapter 18

## GMP, MPD, and related libraries: an overview

### 18.1 Integer Types and Fractions

### 18.2 FloatingPoint Types

#### 18.2.1 Fixed Single Precision

The IEEE 754 standard specifies a binary32 as having:

Sign bit: 1 bit Exponent width: 8 bits Significand precision: 24 (23 explicitly stored) This gives from 6 to 9 significant decimal digits precision (if a decimal string with at most 6 significant decimal is converted to IEEE 754 single precision and then converted back to the same number of significant decimal, then the final string should match the original; and if an IEEE 754 single precision is converted to a decimal string with at least 9 significant decimal and then converted back to single, then the final number must match the original [3]).

Sign bit determines the sign of the number, which is the sign of the significand as well. Exponent is either an 8 bit signed integer from  $-128$  to  $127$  (2's Complement) or an 8 bit unsigned integer from  $0$  to  $255$  which is the accepted biased form in IEEE 754 binary32 definition. For this case an exponent value of  $127$  represents the actual zero.

The true significand includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zeros. Thus only 23 fraction bits of the significand appear in the memory format but the total precision is 24 bits (equivalent to  $\log_{10}(224) \approx 7.225$  decimal digits). See [Kahan \(1997\)](#).

#### 18.2.2 Fixed Double Precision

Double-precision binary floating-point is a commonly used format on PCs, due to its wider range over single-precision floating point, in spite of its performance and bandwidth cost. As with single-precision floating-point format, it lacks precision on integer numbers when compared with an integer format of the same size. It is commonly known simply as double. The IEEE 754 standard specifies a binary64 as having:

Sign bit: 1 bit Exponent width: 11 bits Significand precision: 53 bits (52 explicitly stored) This gives from  $15 \times 17$  significant decimal digits precision. If a decimal string with at most 15 significant digits is converted to IEEE 754 double precision representation and then converted back to a string with the same number of significant digits, then the final string should match

the original; and if an IEEE 754 double precision is converted to a decimal string with at least 17 significant digits and then converted back to double, then the final number must match the original (see [Kahan \(1997\)](#)).

## 18.3 Arithmetic Operators

### 18.3.1 Addition

---

Operator  $+$

---

Function **.Plus**(*a* As mpNum, *b* As mpNum) As mpNum  
 Function **.PlusInt**(*a* As mpNum, *b* As Integer) As mpNum

---

The binary operator  $+$  is used to return the sum of the 2 operands *a* and *b*, and assign the result to *c*:  $c = a + b$ .

For languages not supporting operator overloading, the function **.Plus** can be used to achieve the same:  $c = a.\text{Plus}(b)$

The function **.PlusInt** can be used if the second operand is an integer:  $c = a.\text{PlusInt}(b)$

### 18.3.2 Subtraction

---

Operator  $-$

---

Function **.Minus**(*a* As mpNum, *b* As mpNum) As mpNum  
 Function **.MinusInt**(*a* As mpNum, *b* As Integer) As mpNum

---

The binary operator  $-$  is used to return the difference of the 2 operands *a* and *b*, and assign the result to *c*:  $c = a - b$ .

For languages not supporting operator overloading, the function **.Minus** can be used to achieve the same:  $c = a.\text{Minus}(b)$

The function **.MinusInt** can be used if the second operand is an integer:  $c = a.\text{MinusInt}(b)$

### 18.3.3 Multiplication (Scalars, Vectors and Matrices)

---

Operator  $*$

---

Function **.Times**(*a* As mpNum, *b* As mpNum) As mpNum  
 Function **.TimesInt**(*a* As mpNum, *b* As Integer) As mpNum  
 Function **.TimesMat**(*a* As mpNum, *b* As Integer) As mpNum  
 Function **.DotProd**(*a* As mpNum, *b* As Integer) As mpNum  
 Function **.LSH**(*a* As mpNum, *b* As Integer) As mpNum

---

The binary operator  $*$  is used to return the product of the 2 operands *a* and *b*, and assign the result to *c*:  $c = a * b$ .

For languages not supporting operator overloading, the function **.Times** can be used to achieve the same:  $c = a.\text{Times}(b)$

The function **.TimesInt** can be used if the second operand is an integer:  $c = a.\text{TimesInt}(b)$

### 18.3.4 Scalar Division

---

Operator /

---

Function **.Div**(**a** As mpNum, **b** As mpNum) As mpNum  
 Function **.DivInt**(**a** As mpNum, **b** As Integer) As mpNum  
 Function **.RSH**(**a** As mpNum, **b** As Integer) As mpNum

---

The binary operator / is used to return the quotient of the 2 operands  $a$  and  $b$ , and assign the result to  $c$ :  $c = a / b$ .

For languages not supporting operator overloading, the function **.Div** can be used to achieve the same:  $c = a$ .**Div**( $b$ )

The function **.DivInt** can be used if the second operand is an integer:  $c = a$ .**DivInt**( $b$ )

### 18.3.5 Modulo

---

Operator **mod**

---

Function **.Mod**(**a** As mpNum, **b** As mpNum) As mpNum  
 Function **.ModInt**(**a** As mpNum, **b** As Integer) As mpNum

---

The binary operator **mod** is used to return the modulo of the 2 operands  $a$  and  $b$ , and assign the result to  $c$ :  $c = a \bmod b$ .

For languages not supporting operator overloading, the function **.Mod** can be used to achieve the same:  $c = a$ .**Mod**( $b$ )

The function **.ModInt** can be used if the second operand is an integer:  $c = a$ .**ModInt**( $b$ )

### 18.3.6 Power

---

Operator ^

---

Function **.Pow**(**a** As mpNum, **b** As mpNum) As mpNum  
 Function **.PowInt**(**a** As mpNum, **b** As Integer) As mpNum

---

The binary operator ^ is used to return  $a$  raised to the power of  $b$ , and assign the result to  $c$ :  $c = a ^ b$ .

For languages not supporting operator overloading, the function **.Pow** can be used to achieve the same:  $c = a$ .**Pow**( $b$ )

The function **.PowInt** can be used if the second operand is an integer:  $c = a$ .**PowInt**( $b$ )

## 18.4 Comparison Operators and Sorting

### 18.4.1 Equal

---

Operator = (VB.NET)  
 Operator == (C#)

---

Function **.EQ**(**a** As mpNum, **b** As mpNum) As Boolean

---

The binary logical operator = returns TRUE if  $a = b$  and FALSE otherwise, e.g.:  
 if ( $a = b$ ) then

For languages not supporting operator overloading, the function `.EQ` can be used to achieve the same, e.g.:

`if a.EQ(b) then`

#### 18.4.2 Greater or equal

---

Operator `>=`

Function `.GE(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `>=` returns TRUE if  $a \geq b$  and FALSE otherwise, e.g.:

`if (a >= b) then`

For languages not supporting operator overloading, the function `.GE` can be used to achieve the same, e.g.:

`if a.GE(b) then`

#### 18.4.3 Greater than

---

Operator `>`

Function `.GT(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `>` returns TRUE if  $a > b$  and FALSE otherwise, e.g.:

`if (a > b) then`

For languages not supporting operator overloading, the function `.GT` can be used to achieve the same, e.g.:

`if a.GT(b) then`

#### 18.4.4 Less or equal

---

Operator `<=`

Function `.LE(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `<=` returns TRUE if  $a \leq b$  and FALSE otherwise, e.g.:

`if (a <= b) then`

For languages not supporting operator overloading, the function `.LE` can be used to achieve the same, e.g.:

`if a.LE(b) then`

#### 18.4.5 Less than

---

Operator `<`

Function `.LT(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `>` returns TRUE if  $a < b$  and FALSE otherwise, e.g.:

`if (a < b) then`

For languages not supporting operator overloading, the function `.LT` can be used to achieve the same, e.g.:

`if a.LT(b) then`

### 18.4.6 Not equal

---

Operator `<>` (VB.NET)

Operator `!=` (C#)

---

Function `.NE(a As mpNum, b As mpNum) As Boolean`

---

The binary logical operator `<>` returns TRUE if  $a \neq b$  and FALSE otherwise, e.g.:

`if (a <> b) then`

For languages not supporting operator overloading, the function `.NE` can be used to achieve the same, e.g.:

`if a.NE(b) then`

### 18.4.7 IsApproximate

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

### 18.4.8 IsSmall

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

## 18.5 Vectors, Matrices and Tables

### 18.5.1 Dimension (Vectors and Matrices)

---

Property `.Rows(a As mpNum, b As mpNum) As mpNum`

Property `.Cols(a As mpNum, b As mpNum) As mpNum`

Property `.Size(a As mpNum, b As mpNum) As mpNum`

---

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 18.5.2 Precision

---

Property **.Prec10**(*a* As mpNum, *b* As mpNum) As mpNum

Property **.Prec2**(*a* As mpNum, *b* As mpNum) As mpNum

---

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 18.5.3 Item

---

Property **.Item**(*a* As mpNum, *b* As mpNum) As mpNum

---

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 18.5.4 Row

---

Property **.Row**(*a* As mpNum, *b* As mpNum) As mpNum

Property **.TopRows**(*a* As mpNum, *b* As mpNum) As mpNum

Property **.MiddleRows**(*a* As mpNum, *b* As mpNum) As mpNum

Property **.BottomRows**(*a* As mpNum, *b* As mpNum) As mpNum

---

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 18.5.5 Column

---

Property **.Col**(*a* As mpNum, *b* As mpNum) As mpNum

Property **.LeftCols**(*a* As mpNum, *b* As mpNum) As mpNum

Property **.MiddleCols**(*a* As mpNum, *b* As mpNum) As mpNum

---

Property **.RightCols**(*a* As mpNum, *b* As mpNum) As mpNum

---

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 18.5.6 Matrix

---

Property **.FillLinearByStep**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.SetRandomSymmetric**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.Block**(*a* As mpNum, *b* As mpNum) As mpNum

Property **.TopLeftCorner**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.TopRightCorner**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.BottomLeftCorner**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.BottomRightCorner**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.Diagonal**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.TriangularView**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.Adjoint**(*a* As mpNum, *b* As mpNum) As mpNum  
 Property **.AsDiagonal**(*a* As mpNum, *b* As mpNum) As mpNum

---

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 18.5.7 Sorting

---

Function **.Sorted**(*a* As mpNum, *b* As mpNum) As mpNum

---

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

### 18.5.8 Table

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

### 18.5.9 List of Tables

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

# Chapter 19

## FMPZ

### 19.1 Multiprecision Rational Numbers (GMP: MPQ)

The GMP reference is [Granlund & the GMP development team \(2013\)](#)

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers and floating point numbers. There is no practical limit on the precision except the ones implied by the available memory in the machine GMP runs on.

The GNU Multiple Precision Arithmetic Library (GMP) is a free library for arbitrary-precision arithmetic, operating on signed integers, rational numbers, and floating point numbers.[2] There are no practical limits to the precision except the ones implied by the available memory in the machine GMP runs on (operand dimension limit is 232-1 bits on 32-bit machines and 237 bits on 64-bit machines).[3] GMP has a rich set of functions, and the functions have a regular interface. The basic interface is for C but wrappers exist for other languages including Ada, C++, C#, OCaml, Perl, PHP, and Python. In the past, the Kaffe Java virtual machine used GMP to support Java built-in arbitrary precision arithmetic. This feature has been removed from recent releases, causing protests from people who claim that they used Kaffe solely for the speed benefits afforded by GMP.[4] As a result, GMP support has been added to GNU Classpath.[5]

The main target applications of GMP are cryptography applications and research, Internet security applications, and computer algebra systems.

GMP aims to be faster than any other bignum library for all operand sizes. Some important factors in doing this are:

Using full words as the basic arithmetic type. Using different algorithms for different operand sizes; algorithms that are faster for very big numbers are usually slower for small numbers. Highly optimized assembly language code for the most important inner loops, specialized for different processors. The first GMP release was made in 1991. It is constantly developed and maintained.[1] GMP is part of the GNU project (although its website being off gnu.org may cause confusion), and is distributed under the GNU Lesser General Public License (LGPL).

GMP is used for integer arithmetic in many computer algebra systems such as Mathematica[6] and Maple.[7] It is also used in the Computational Geometry Algorithms Library (CGAL) because geometry algorithms tend to 'explode' when using ordinary floating point CPU math.[8]

GMP is needed to build the GNU Compiler Collection (GCC).[9]

#### 19.1.0.1 mpz input

```
int mpz_set_str (mpz t rop, const char *str, int base) Set the value of rop from str, a null-terminated C string in base base. White space is allowed in the string, and is simply ignored. The base may vary from 2 to 62, or if base is 0, then the leading characters are used: 0x and 0X
```

for hexadecimal, 0b and 0B for binary, 0 for octal, or decimal otherwise. For bases up to 36, case is ignored; upper-case and lower-case letters have the same value. For bases 37 to 62, upper-case letter represent the usual 10..35 while lower-case letter represent 36..61. This function returns 0 if the entire string is a valid number in base base. Otherwise it returns  $\text{aL}\check{S}1$ .

### 19.1.0.2 mpz output

`char * mpz_get_str (char *str, int base, const mpz t op)` Convert op to a string of digits in base base. The base argument may vary from 2 to 62 or from  $\text{aL}\check{S}2$  to  $\text{aL}\check{S}36$ . For base in the range 2..36, digits and lower-case letters are used; for  $\text{aL}\check{S}2..\text{aL}\check{S}36$ , digits and upper-case letters are used; for 37..62, digits, upper-case letters, and lower-case letters (in that signifi cance order) are used.

If str is NULL, the result string is allocated using the current allocation function (see Chapter 13 [Custom Allocation], page 86). The block will be `strlen(str)+1` bytes, that being exactly enough for the string and null-terminator. If str is not NULL, it should point to a block of storage large enough for the result, that being `mpz_sizeinbase (op, base) + 2`. The two extra bytes are for a possible minus sign, and the null-terminator. A pointer to the result string is returned, being either the allocated block, or the given str.

Book reference: [Shoup \(2009\)](#)

Background on discrete applied algebra: [Hardy D.W. \(2009\)](#)

## 19.2 Arithmetic Operators

### 19.2.1 Unary Minus

---

Function **intNeg(*n* As mpNum) As mpNum**

---

The function `intNeg` returns  $-n$

**Parameter:**

*n*: An Integer.

### 19.2.2 Addition

---

Function **intAdd(*n1* As mpNum, *n2* As mpNum) As mpNum**

---

The function `intAdd` returns  $n_1 + n_2$ .

**Parameters:**

*n1*: An Integer.

*n2*: An Integer.

The function `intAdd(n1, n2)` returns the sum of *n<sub>1</sub>* and *n<sub>2</sub>*:

$$\text{intAdd}(n_1, n_2) = n_1 + n_2. \quad (19.2.1)$$

### 19.2.3 Subtraction

---

Function **intSub(*n1* As mpNum, *n2* As mpNum) As mpNum**

---

The function `intSub` returns  $n_1 - n_2$ .

**Parameters:**

- n1*: An Integer.  
*n2*: An Integer.

The function `intSub`(*n1*, *n2*) returns the difference of *n1* and *n2*:

$$\text{intSub}(n_1, n_2) = n_1 - n_2. \quad (19.2.2)$$

### 19.2.4 Multiplication

---

#### Function `intMul`(*n1* As *mpNum*, *n2* As *mpNum*) As *mpNum*

---

The function `intMul` returns  $n_1 \times n_2$ .

**Parameters:**

- n1*: An Integer.  
*n2*: An Integer.

The function `intMul`(*n1*, *n2*) returns the product of *n1* and *n2*:

$$\text{intMul}(n_1, n_2) = n_1 \times n_2. \quad (19.2.3)$$

### 19.2.5 Fused-Multiply-Add fma

---

#### Function `intFma`(*n1* As *mpNum*, *n2* As *mpNum*, *n3* As *mpNum*) As *mpNum*

---

The function `intFma` returns  $(n_1 \times n_2) + n_3$ .

**Parameters:**

- n1*: An Integer.  
*n2*: An Integer.  
*n3*: An Integer.

The function `intFma`(*n1*, *n2*, *n3*) returns the product of *n1* and *n2*, plus *n3*:

$$\text{intFma}(n_1, n_2, n_3) = (n_1 \times n_2) + n_3. \quad (19.2.4)$$

### 19.2.6 Fused-Multiply-Subtract fms

---

#### Function `intFms`(*n1* As *mpNum*, *n2* As *mpNum*, *n3* As *mpNum*) As *mpNum*

---

The function `intFms` returns  $(n_1 \times n_2) - n_3$ .

**Parameters:**

- n1*: An Integer.  
*n2*: An Integer.  
*n3*: An Integer.

The function `intFms`(*n1*, *n2*, *n3*) returns the product of *n1* and *n2*, minus *n3*:

$$\text{intFms}(n_1, n_2, n_3) = (n_1 \times n_2) - n_3. \quad (19.2.5)$$

### 19.2.7 Multiplication by multiples of 2 (LSH)

---

Function **intLSH**(*n* As mpNum, *k* As mpNum) As mpNum

---

The function intLSH returns the product of *n* and  $2^k$

**Parameters:**

*n*: An Integer.  
*k*: An Integer.

The function intLSH(*n, k*) returns the product of *n* and  $2^k$ :

$$\text{intLSH}(n, k) = n \times 2^k. \quad (19.2.6)$$

This operation can also be defined as a left shift by *k* bits.

### 19.2.8 Division by multiples of 2 (RSH)

---

Function **intRSH**(*n* As mpNum, *k* As mpNum) As mpNum

---

The function intRSH returns the quotient of *n* and  $2^k$

**Parameters:**

*n*: An Integer.  
*k*: An Integer.

The function intRSH(*n, k*) returns the quotient of *n* and  $2^k$ :

$$\text{intRSH}(n, k) = n \div 2^k. \quad (19.2.7)$$

This operation can also be defined as a right shift by *k* bits.

### 19.2.9 Exact Division

---

Function **intDivExact**(*n* As mpNum, *d* As mpNum) As mpNum

---

The function intDivExact returns *n/d*

**Parameters:**

*n*: An Integer.  
*d*: An Integer.

Returns *n/d*. This function produces correct results only when it is known in advance that *d* divides *n*. This routine is much faster than the other division functions, and is the best choice when exact division is known to occur, for example reducing a rational to lowest terms.

### 19.2.10 Modulo Division

---

Function **intMod**(*n* As mpNum, *d* As mpNum) As mpNum

---

The function intMod returns *n* mod *d*.

**Parameters:**

*n*: An Integer.

*d*: An Integer.

The sign of the divisor is ignored; the result is always non-negative.

## 19.3 Divisions, forming quotients and/or remainder

Division is undefined if the divisor is zero. Passing a zero divisor to the division or modulo functions (including the modular powering functions), will cause an intentional division by zero. This lets a program handle arithmetic exceptions in these functions the same way as for normal integer arithmetic.

The following routines calculate *n* divided by *d*, forming a quotient *q* and/or remainder *r*. For the 2exp functions,  $d = 2^b$ . The rounding is in three styles, each suiting different applications.

- cdiv rounds *q* up towards  $+\infty$ , and *r* will have the opposite sign to *d*. The c stands for "ceil".
- fdiv rounds *q* down towards  $-\infty$ , and *r* will have the same sign as *d*. The f stands for "floor".
- tdiv rounds *q* towards zero, and *r* will have the same sign as *n*. The t stands for "truncate".

In all cases *q* and *r* will satisfy  $n = qd + r$ , and *r* will satisfy  $0 \leq |r| < |d|$ . The *q* functions calculate only the quotient, the *r* functions only the remainder, and the *qr* functions calculate both. Note that for *qr* the same variable cannot be passed for both *q* and *r*, or results will be unpredictable.

### 19.3.1 Quotient only, rounded up

---

#### Function **intCDivQ(*n* As mpNum, *d* As mpNum) As mpNum**

---

The function intCDivQ returns the quotient of *n* and *d*, rounded up towards  $+\infty$ .

**Parameters:**

*n*: An Integer.

*d*: An Integer.

$$\text{intCDivQ}(n, d) = \lceil n \div d \rceil. \quad (19.3.1)$$

---

#### Function **intCDivQ2exp(*n* As mpNum, *b* As mpNum) As mpNum**

---

The function intCDivQ2exp returns the quotient of *n* and  $2^b$ , rounded up towards  $+\infty$ .

**Parameters:**

*n*: An Integer.

*b*: An Integer.

$$\text{intCDivQ2exp}(n, b) = \lceil n \div 2^b \rceil. \quad (19.3.2)$$

### 19.3.2 Remainder only (Quotient rounded up)

---

#### Function **intCDivR**(*n* As mpNum, *d* As mpNum) As mpNum

---

The function `intCDivR` returns the remainder, once the quotient of *n* and *d*, rounded up towards  $+\infty$ , has been obtained.

**Parameters:**

*n*: An Integer.

*d*: An Integer.

$$\text{intCDivR}(n, d) = n - d \times \lceil n \div d \rceil. \quad (19.3.3)$$

---

#### Function **intCDivR2exp**(*n* As mpNum, *b* As mpNum) As mpNum

---

The function `intCDivR2exp` returns the remainder, once the quotient of *n* and  $2^b$ , rounded up towards  $+\infty$ , has been obtained.

**Parameters:**

*n*: An Integer.

*b*: An Integer.

$$\text{intCDivR2exp}(n, d) = n - 2^b \times \lceil n \div 2^b \rceil. \quad (19.3.4)$$

### 19.3.3 Quotient and Remainder, Quotient rounded up

---

#### Function **intCDivQR**(*n* As mpNum, *d* As mpNum) As mpNumList[2]

---

The function `intCDivQR` returns the quotient of *n* and *d*, rounded up towards  $+\infty$ , and the remainder.

**Parameters:**

*n*: An Integer.

*d*: An Integer.

`intCDivQR[1]` returns `intCDivQ(n, k)` as defined in equation 19.3.1, and `intCDivQR[2]` returns `intCDivR(n, k)` as defined in in equation 19.3.3.

### 19.3.4 Quotient only, rounded down

---

#### Function **intFDivQ**(*n* As mpNum, *d* As mpNum) As mpNum

---

The function `intFDivQ` returns the quotient of *n* and *d*, rounded down towards  $-\infty$ .

**Parameters:**

*n*: An Integer.

*d*: An Integer.

$$\text{intFDivQ}(n, d) = \lfloor n \div d \rfloor. \quad (19.3.5)$$

---

#### Function **intFDivQ2exp**(*n* As mpNum, *b* As mpNum) As mpNum

---

The function `intFDivQ2exp` returns the quotient of  $n$  and  $2^b$ , rounded down towards  $-\infty$ .

**Parameters:**

- $n$ : An Integer.
- $b$ : An Integer.

$$\text{intFDivQ2exp}(n, d) = \left\lfloor n \div 2^b \right\rfloor. \quad (19.3.6)$$

### 19.3.5 Remainder only (Quotient rounded down)

---

Function **intFDivR**( $n$  As `mpNum`,  $d$  As `mpNum`) As `mpNum`

---

The function `intFDivR` returns the remainder, once the quotient of  $n$  and  $d$ , rounded down towards  $-\infty$ , has been obtained.

**Parameters:**

- $n$ : An Integer.
- $d$ : An Integer.

$$\text{intFDivR}(n, d) = n - d \times \left\lfloor n \div d \right\rfloor. \quad (19.3.7)$$

---

Function **intFDivR2exp**( $n$  As `mpNum`,  $b$  As `mpNum`) As `mpNum`

---

The function `intFDivR2exp` returns the remainder, once the quotient of  $n$  and  $2^b$ , rounded down towards  $-\infty$ , has been obtained.

**Parameters:**

- $n$ : An Integer.
- $b$ : An Integer.

$$\text{intFDivR2exp}(n, d) = n - 2^b \times \left\lfloor n \div 2^b \right\rfloor. \quad (19.3.8)$$

### 19.3.6 Quotient and Remainder, Quotient rounded down

---

Function **intFDivQR**( $n$  As `mpNum`,  $d$  As `mpNum`) As `mpNumList[2]`

---

The function `intFDivQR` returns the quotient of  $n$  and  $d$ , rounded down towards  $-\infty$ , and the remainder.

**Parameters:**

- $n$ : An Integer.
- $d$ : An Integer.

`intFDivQR[1]` returns `intFDivQ`( $n, k$ ) as defined in equation 19.3.5, and `intFDivQR[2]` returns `intFDivR`( $n, k$ ) as defined in in equation 19.3.7.

### 19.3.7 Quotient only, Quotient truncated

---

Function **intTDivQ**( $n$  As `mpNum`,  $d$  As `mpNum`) As `mpNum`

---

The function `intTDivQ` returns the quotient of  $n$  and  $d$ , rounded towards zero.

**Parameters:**

- n*: An Integer.  
*d*: An Integer.

$$\text{intTDivQ}(n, d) = \lfloor n \div d \rfloor. \quad (19.3.9)$$

---

**Function `intTDivQ2exp`(*n* As mpNum, *b* As mpNum) As mpNum**

---

The function `intTDivQ2exp` returns the quotient of *n* and  $2^b$ , rounded towards zero.

**Parameters:**

- n*: An Integer.  
*b*: An Integer.

$$\text{intTDivQ2exp}(n, d) = \left\lfloor n \div 2^b \right\rfloor. \quad (19.3.10)$$

### 19.3.8 Remainder only (Quotient truncated)

---

**Function `intTDivR`(*n* As mpNum, *d* As mpNum) As mpNum**

---

The function `intTDivR` returns the remainder, once the quotient of *n* and *d*, rounded towards zero, has been obtained.

**Parameters:**

- n*: An Integer.  
*d*: An Integer.

$$\text{intTDivR}(n, d) = n - d \times \lfloor n \div d \rfloor. \quad (19.3.11)$$

---

**Function `intTDivR2exp`(*n* As mpNum, *b* As mpNum) As mpNum**

---

The function `intTDivR2exp` returns the remainder, once the quotient of *n* and  $2^b$ , rounded towards zero, has been obtained.

**Parameters:**

- n*: An Integer.  
*b*: An Integer.

$$\text{intTDivR2exp}(n, d) = n - 2^b \times \left\lfloor n \div 2^b \right\rfloor. \quad (19.3.12)$$

### 19.3.9 Quotient and Remainder, Quotient truncated

---

**Function `intTDivQr`(*n* As mpNum, *d* As mpNum) As mpNumList[2]**

---

The function `intTDivQr` returns the quotient of *n* and *d*, rounded towards zero, and the remainder.

**Parameters:**

- n*: An Integer.  
*d*: An Integer.

`intTDivQR[1]` returns  $\text{intTDivQ}(n, k)$  as defined in equation 19.3.9, and `intTDivQR[2]` returns  $\text{intTDivR}(n, k)$  as defined in equation 19.3.11.

## 19.4 Logical Operators

### 19.4.1 Bitwise AND

---

Function **intAND**(*n1* As mpNum, *n2* As mpNum) As mpNum

---

The function `intAND` returns  $n_1$  bitwise-and  $n_2$ .

**Parameters:**

*n1*: An Integer.

*n2*: An Integer.

### 19.4.2 Bitwise Inclusive OR

---

Function **intIOR**(*n1* As mpNum, *n2* As mpNum) As mpNum

---

The function `intIOR` returns  $n_1$  bitwise-inclusive-or  $n_2$ .

**Parameters:**

*n1*: An Integer.

*n2*: An Integer.

### 19.4.3 Bitwise Exclusive OR

---

Function **intXOR**(*n1* As mpNum, *n2* As mpNum) As mpNum

---

The function `intXOR` returns  $n_1$  bitwise-exclusive-or  $n_2$ .

**Parameters:**

*n1*: An Integer.

*n2*: An Integer.

## 19.5 Bit-Oriented Functions

### 19.5.1 Complement

---

Function **intComplement**(*n* As mpNum) As mpNum

---

The function `intComplement` returns the one's complement of  $n$ .

**Parameter:**

*n*: An Integer.

### 19.5.2 Hamming Distance

---

Function **intHamDist**(*n1* As mpNum, *n2* As mpNum) As mpNum

---

The function `intHamDist` returns the hamming distance between the two operands

**Parameters:**

*n1*: An Integer.

*n2*: An Integer.

If  $n_1$  and  $n_2$  are both  $\geq 0$  or both  $< 0$ , return the hamming distance between the two operands, which is the number of bit positions where  $n_1$  and  $n_2$  have different bit values. If one operand is  $\geq 0$  and the other  $< 0$  then the number of bits different is infinite, and the return value is the largest possible `mp_bitcnt_t`.

### 19.5.3 Testing , setting, and clearing a Bit

---

#### Function `intTestBit(n As mpNum, k As mpNum) As mpNum`

---

The function `intTestBit` returns 1 or 0 according to whether bit  $k$  in  $n$  is set or not.

**Parameters:**

*n*: An Integer.

*k*: An Integer.

#### Function `intComBit(n As mpNum, k As mpNum) As mpNum`

---

The function `intComBit` returns  $n$  with the complement bit  $k$  set in  $n$ .

**Parameters:**

*n*: An Integer.

*k*: An Integer.

#### Function `intClearBit(n As mpNum, k As mpNum) As mpNum`

---

The function `intClearBit` returns  $n$  with the bit  $k$  cleared in  $n$ .

**Parameters:**

*n*: An Integer.

*k*: An Integer.

#### Function `intSetBit(n As mpNum, k As mpNum) As mpNum`

---

The function `intSetBit` returns  $n$  with the bit  $k$  set in  $n$ .

**Parameters:**

*n*: An Integer.

*k*: An Integer.

### 19.5.4 Scanning for 0 or 1

---

#### Function `intScan0(n As mpNum, k As mpNum) As mpNum`

---

The function `intScan0` returns the index of the found bit 0, starting from bit  $k$ .

**Parameters:**

*n*: An Integer.

*k*: An Integer.

---

**Function `intScan1(n As mpNum, k As mpNum) As mpNum`**

---

The function `intScan1` returns the index of the found bit 1, starting from bit *k*.

**Parameters:**

*n*: An Integer.

*k*: An Integer.

Scan *n*, starting from bit *k*, towards more significant bits, until the first 0 or 1 bit (respectively) is found. Return the index of the found bit.

If the bit at starting bit is already whatâŽs sought, then *k* is returned. If thereâŽs no bit found, then the largest possible `mp_bitcnt_t` is returned. This will happen in `mpz_scan0` past the end of a negative number, or `mpz_scan1` past the end of a nonnegative number.

### 19.5.5 Population Count

---

**Function `intPopCount(n As mpNum) As mpNum`**

---

The function `intPopCount` returns the population count of *n*.

**Parameter:**

*n*: An Integer.

If  $n \geq 0$ , return the population count of *n*, which is the number of 1 bits in the binary representation. If  $n < 0$ , the number of 1s is infinite, and the return value is the largest possible `mp_bitcnt_t`.

## 19.6 Sign, Powers and Roots

### 19.6.1 Sign

---

**Function `intSgn(n As mpNum) As mpNum`**

---

The function `intSgn` returns the sign of *n*.

**Parameter:**

*n*: An Integer.

### 19.6.2 Absolute value

---

**Function `intAbs(n As mpNum) As mpNum`**

---

The function `intAbs` returns the absolute value of *n*.

**Parameter:**

*n*: An Integer.

### 19.6.3 Power Function: $n^k$ ; $n, k \in \mathbb{Z}$

---

Function **intPow**(*n* As mpNum, *k* As mpNum) As mpNum

---

The function intPow returns the value of  $n^k$ . The case  $0^0$  yields 1.

**Parameters:**

*n*: An Integer.  
*k*: An Integer.

### 19.6.4 Power Function modulo m: $n^k \bmod m$ ; $m, n, k \in \mathbb{Z}$

---

Function **intPowMod**(*n* As mpNum, *k* As mpNum, *m* As mpNum) As mpNum

---

The function intPowMod returns the value of  $n^k \bmod m$ .

**Parameters:**

*n*: An Integer.  
*k*: An Integer.  
*m*: An Integer.

Returns the value of  $n^k \bmod m$ ;  $m, n, k \in \mathbb{Z}$ .

Negative *k* is supported if an inverse  $n^{-1} \bmod m$  exists (see mpz\_invert in Section 5.9 [Number Theoretic Functions], page 36). If an inverse does not exist then a divide by zero is raised.

### 19.6.5 Truncated integer part of the square root: $\lfloor \sqrt{n} \rfloor$

---

Function **intSqrt**(*n* As mpNum) As mpNum

---

The function intSqrt returns the truncated integer part of the square root of *n*.

**Parameter:**

*n*: An Integer.

Returns  $\lfloor \sqrt{m} \rfloor$ , the truncated integer part of the square root of *m*.

### 19.6.6 Truncated integer part of the square root: $\lfloor \sqrt{m} \rfloor$ , with remainder

---

Function **intSqrtRem**(*n* As mpNum) As mpNumList[2]

---

The function intSqrtRem returns the truncated integer part of the square root of *n*, and the remainder.

**Parameter:**

*n*: An Integer.

intSqrtRem[1] returns *s* = intSqrt(*n*) as defined in section 19.6.5, and intSqrtRem[2] returns the remainder (*m* - *s*<sup>2</sup>), which will be zero if *m* is a perfect square.

### 19.6.7 Truncated integer part of the nth root: $\lfloor \sqrt[n]{m} \rfloor$

---

Function **intRoot(*n* As mpNum, *m* As mpNum)** As mpNum

---

The function intRoot returns the truncated integer part of the  $n^{th}$  root of *m*

**Parameters:**

*n*: An Integer.

*m*: An Integer.

Returns  $\lfloor \sqrt[n]{m} \rfloor$ , the truncated integer part of the  $n^{th}$  root of *m*.

### 19.6.8 Truncated integer part of the nth root: $\lfloor \sqrt[n]{m} \rfloor$ , with remainder

---

Function **intRootRem(*n* As mpNum, *m* As mpNum)** As mpNumList[2]

---

The function intRootRem returns the truncated integer part of the  $n^{th}$  root of *m*, with remainder

**Parameters:**

*n*: An Integer.

*m*: An Integer.

intRootRem[1] returns *s* = intSqrt(*m*, *n*) as defined in section 19.6.7, and intRootRem[2] returns the remainder  $(m - s^2)$ .

## 19.7 Numbertheoretic Functions

### 19.7.1 Factorial

---

Function **intFactorial(*n* As mpNum)** As mpNum

---

The function intFactorial returns  $n!$ , the factorial of *n*

**Parameter:**

*n*: An Integer.

### 19.7.2 Binomial Coefficient, Combinations

---

Function **intBinCoeff(*n* As mpNum, *k* As mpNum)** As mpNum

---

The function intBinCoeff returns the binomial coefficient

**Parameters:**

*n*: An Integer.

*k*: An Integer.

Returns the binomial coefficient,  $\binom{n}{k}$ . Negative values of *n* are supported, using the identity

$$\binom{-n}{k} = (-1)^k \binom{n+k-1}{k}. \quad (19.7.1)$$

### 19.7.3 Next Prime

---

#### Function **intNextprime(*n* As Integer) As Integer**

---

The function `intNextprime` returns the next prime greater than  $n$ .

**Parameter:**

$n$ : An Integer.

Returns the next prime greater than  $n$ . This function uses a probabilistic algorithm to identify primes. The chance of a composite passing will be extremely small.

### 19.7.4 Greatest Common Divisor (GCD)

---

#### Function **intGcd(*n1* As mpNum, *n2* As mpNum) As mpNum**

---

The function `intGcd` returns the greatest common divisor of  $n_1$  and  $n_2$ .

**Parameters:**

$n1$ : An Integer.

$n2$ : An Integer.

The result is always positive even if one or both input operands are negative. Except if both inputs are zero; then this function defines  $\text{intGcd}(0, 0) = 0$ .

### 19.7.5 Greatest Common Divisor, Extended

---

#### Function **intGcdExt(*n1* As mpNum, *n2* As mpNum) As mpNumList[3]**

---

The function `intGcdExt` returns the extended greatest common divisor of  $n_1$  and  $n_2$ .

**Parameters:**

$n1$ : An Integer.

$n2$ : An Integer.

Set  $\text{intGcdExt}[1] = g$  to the greatest common divisor of  $a$  and  $b$ , and in addition set  $\text{intGcdExt}[2] = s$  and  $\text{intGcdExt}[3] = t$  to coefficients satisfying  $as + bt = g$ . The value in  $g$  is always positive, even if one or both of  $a$  and  $b$  are negative (or zero if both inputs are zero). The values in  $s$  and  $t$  are chosen such that normally,  $|s| < |b|/(2g)$  and  $|t| < |a|/(2g)$ , and these relations define  $s$  and  $t$  uniquely. There are a few exceptional cases:

If  $|a| = |b|$ , then  $s = 0, t = \text{sgn}(b)$ .

Otherwise,  $s = \text{sgn}(a)$  if  $b = 0$  or  $|b| = 2g$ , and  $t = \text{sgn}(b)$  if  $a = 0$  or  $|a| = 2g$ .

In all cases,  $s = 0$  if and only if  $g = |b|$ , i.e., if  $b$  divides  $a$  or  $a = b = 0$ .

### 19.7.6 Least Common Multiple (LCM)

---

#### Function **intLcm(*n1* As mpNum, *n2* As mpNum) As mpNum**

---

The function `intLcm` returns the least common multiple of  $n_1$  and  $n_2$ .

**Parameters:**

*n1*: An Integer.

*n2*: An Integer.

Returns the least common multiple of  $n_1$  and  $n_2$ . The returned value is always positive, irrespective of the signs of  $n_1$  and  $n_2$ . The returned value will be zero if either  $n_1$  or  $n_2$  is zero.

### 19.7.7 Inverse Modulus

---

Function **intInvertMod(*n1* As mpNum, *n2* As mpNum) As mpNum**

---

The function `intInvertMod` returns the inverse of  $n_1$  modulo  $n_2$

**Parameters:**

*n1*: An Integer.

*n2*: An Integer.

Returns the inverse of  $n_1$  modulo  $n_2$ . If the inverse exists, the indicator value is non-zero and the returned value will satisfy  $0 < \text{rop} < |n_2|$ . If an inverse does not exist the indicator value is zero and `rop` is undefined. The behaviour of this function is undefined when  $n_2$  is zero.

### 19.7.8 Remove Factor

---

Function **intRemoveFactor(*n* As mpNum, *f* As mpNum) As mpNum**

---

The function `intRemoveFactor` returns  $n$  with all occurrences of the factor  $f$  removed from  $n$ .

**Parameters:**

*n*: An Integer.

*f*: An Integer.

Remove all occurrences of the factor  $f$  from  $n$  and return the result in `intRemoveFactor[1]`. `intRemoveFactor[2]` contains how many such occurrences were removed.

### 19.7.9 Legendre Symbol

---

Function **intLegendreSymbol(*a* As mpNum, *p* As mpNum) As mpNum**

---

The function `intLegendreSymbol` returns the Legendre symbol  $\left(\frac{a}{p}\right)$ .

**Parameters:**

*a*: An Integer.

*p*: An Integer.

Calculate the Legendre symbol  $\left(\frac{a}{p}\right)$ . This is defined only for  $p$  an odd positive prime, and for such  $p$  it is identical to the Jacobi symbol.

### 19.7.10 Jacobi Symbol

---

Function **intJacobiSymbol(*a* As mpNum, *b* As mpNum) As mpNum**

---

The function `intJacobiSymbol` returns the Jacobi symbol  $\left(\frac{a}{b}\right)$

**Parameters:**

- a*: An Integer.  
*b*: An Integer.

Calculate the Jacobi symbol  $\left(\frac{a}{b}\right)$ . This is defined only for *b* odd.

### 19.7.11 Kronecker Symbol

---

Function **intKroneckerSymbol(*a* As mpNum, *b* As mpNum) As mpNum**

---

The function intKroneckerSymbol returns the Kronecker symbol  $\left(\frac{a}{b}\right)$

**Parameters:**

- a*: An Integer.  
*b*: An Integer.

Calculate the Jacobi symbol  $\left(\frac{a}{b}\right)$  with the Kronecker extension  $\left(\frac{a}{2}\right) = \left(\frac{2}{a}\right)$  when *a* odd, or when *a* odd,  $\left(\frac{a}{2}\right) = 0$  when *a* even. When *b* is odd the Jacobi symbol and Kronecker symbol are identical.

### 19.7.12 Fibonacci Numbers

---

Function **intFibonacci(*n* As mpNum) As mpNum**

---

The function intFibonacci returns the *n*<sup>th</sup> Fibonacci number.

**Parameter:**

- n*: An Integer.

### 19.7.13 Lucas Numbers

---

Function **intLucas(*n* As mpNum) As mpNum**

---

The function intLucas returns the *n*<sup>th</sup> Lucas number.

**Parameter:**

- n*: An Integer.

## 19.8 Additional Numbertheoretic Functions

### 19.8.1 Pseudoprimes

An overview is provided by [Grantham \(2001\)](#).

---

Function **intIsBpswPrp(*n* As mpNum) As mpNum**

---

The function intIsBpswPrp returns True if *n* is a Baillie-Pomerance-Selfridge-Wagstaff probable prime.

**Parameter:**

- n*: An Integer.

is\_bpsw\_prp(n) will return True if n is a Baillie-Pomerance-Selfridge-Wagstaff probable prime. A BPSW probable prime passes the is\_strong\_prp() test with base 2 and the is\_selfridge\_prp() test.

---

**Function `intIsEulerPrp(n As mpNum, a As mpNum) As mpNum`**

---

The function intIsEulerPrp returns True if n is an Euler (also known as Solovay-Strassen) probable

**Parameters:**

*n*: An Integer.

*a*: An Integer.

is\_euler\_prp(n,a) will return True if n is an Euler (also known as Solovay-Strassen) probable prime to the base a.

Assuming:  $\gcd(n, a) == 1$  n is odd

Then an Euler probable prime requires:

$$a^{(n-1)/2} == 1 \pmod{n}$$

---

**Function `intIsExtraStrongLucasPrp(n As mpNum, p As mpNum) As mpNum`**

---

The function intIsExtraStrongLucasPrp returns True if n is an extra strong Lucas probable prime

**Parameters:**

*n*: An Integer.

*p*: An Integer.

is\_extra\_strong\_lucas\_prp(n,p) will return True if n is an extra strong Lucas probable prime with parameters (p,1). Assuming: n is odd  $D = p^2 - 4$ ,  $D != 0$   $\gcd(n, 2^k D) == 1$   $n = s \cdot (2^k r) + \text{Jacobi}(D, n)$ , s odd

Then an extra strong Lucas probable prime requires:

$\text{lucasu}(p, 1, s) == 0 \pmod{n}$  or  $\text{lucasv}(p, 1, s) == +/-2 \pmod{n}$  or  $\text{lucasv}(p, 1, s \cdot (2^k t)) == 0 \pmod{n}$  for some t,  $0 \leq t \leq r$

---

**Function `intIsFermatPrp(n As mpNum, a As mpNum) As mpNum`**

---

The function intIsFermatPrp returns True if n is a Fermat probable prime to the base a

**Parameters:**

*n*: An Integer.

*a*: An Integer.

is\_fermat\_prp(n,a) will return True if n is a Fermat probable prime to the base a.

Assuming:  $\gcd(n, a) == 1$  Then a Fermat probable prime requires:  $a^{(n-1)} == 1 \pmod{n}$

---

**Function `intIsFibonacciPrp(n As mpNum, p As mpNum, q As mpNum) As mpNum`**

---

The function intIsFibonacciPrp returns True if n is an Fibonacci probable prime with parameters (p,q).

**Parameters:**

*n*: An Integer.

*p*: An Integer.

*q*: An Integer.

`is_fibonacci_prp(n,p,q)` will return True if *n* is an Fibonacci probable prime with parameters (*p,q*).

Assuming: *n* is odd  $p \neq 0$ ,  $q = +/-1$   $p^2 - 4q \neq 0$

Then a Fibonacci probable prime requires: `lucasv(p,q,n) == p (mod n)`.

#### Function **intIsLucasPrp(*n* As mpNum, *p* As mpNum, *q* As mpNum) As mpNum**

The function `intIsLucasPrp` returns True if *n* is a Lucas probable prime with parameters (*p,q*).

**Parameters:**

*n*: An Integer.

*p*: An Integer.

*q*: An Integer.

`is_lucas_prp(n,p,q)` will return True if *n* is a Lucas probable prime with parameters (*p,q*).

Assuming: *n* is odd  $D = p^2 - 4q$ ,  $D \neq 0$   $\gcd(n, 2^5 q^5 D) == 1$

Then a Lucas probable prime requires:

`lucasu(p,q,n - Jacobi(D,n)) == 0 (mod n)`

#### Function **intIsSelfridgePrp(*a* As mpNum) As mpNum**

The function `intIsSelfridgePrp` returns True if *n* is a Lucas probable prime with Selfridge parameters (*p,q*).

**Parameter:**

*a*: An Integer.

`is_selfridge_prp(n)` will return True if *n* is a Lucas probable prime with Selfridge parameters (*p,q*).

The Selfridge parameters are chosen by finding the first element *D* in the sequence 5, -7, 9, -11, 13, ... such that `Jacobi(D,n) == -1`. Let *p*=1 and *q* =  $(1-D)/4$  and then perform a Lucas probable prime test.

#### Function **intIsStrongBpswPrp(*a* As mpNum) As mpNum**

The function `intIsStrongBpswPrp` returns True if *n* is a strong Baillie-Pomerance-Selfridge-Wagstaff probable prime

**Parameter:**

*a*: An Integer.

`is_strong_bpsw_prp(n)` will return True if *n* is a strong Baillie-Pomerance-Selfridge-Wagstaff probable prime. A strong BPSW probable prime passes the `is_strong_prp()` test with base 2 and the `is_strongselfridge_prp()` test.

#### Function **intIsStrongLucasPrp(*n* As mpNum, *p* As mpNum, *q* As mpNum) As mpNum**

The function `intIsStrongLucasPrp` returns True if *n* is a strong Lucas probable prime with parameters (*p,q*).

**Parameters:**

*n*: An Integer.

*p*: An Integer.

*q*: An Integer.

`is_strong_lucas_prp(n,p,q)` will return True if *n* is a strong Lucas probable prime with parameters (*p*,*q*).

Assuming: *n* is odd  $D = p^*p - 4^*q$ ,  $D \neq 0$

$\gcd(n, 2^*q^*D) == 1$   $n = s^*(2^{**r}) + \text{Jacobi}(D,n)$ , *s* odd Then a strong Lucas probable prime requires:

$\text{lucasu}(p,q,s) == 0 \pmod{n}$  or  $\text{lucasv}(p,q,s^*(2^{**t})) == 0 \pmod{n}$  for some *t*,  $0 \leq t \leq r$

#### Function **intIsStrongPrp(*n* As mpNum, *a* As mpNum) As mpNum**

The function `intIsStrongPrp` returns True if *n* is an strong (also known as Miller-Rabin) probable prime

#### Parameters:

*n*: An Integer.

*a*: An Integer.

`is_strong_prp(n,a)` will return True if *n* is an strong (also known as Miller-Rabin) probable prime to the base *a*.

Assuming:  $\gcd(n,a) == 1$  *n* is odd  $n = s^*(2^{**r}) + 1$ , with *s* odd

Then a strong probable prime requires one of the following is true:  $a^{**s} == 1 \pmod{n}$  or  $a^{**(s^*(2^{**t}))} == -1 \pmod{n}$  for some *t*,  $0 \leq t \leq r$ .

#### Function **intIsStrongSelfridgePrp(*a* As mpNum) As mpNum**

The function `intIsStrongSelfridgePrp` returns True if *n* is a strong Lucas probable prime with Selfridge parameters

#### Parameter:

*a*: An Integer.

`is_strong_selfridge_prp(n)` will return True if *n* is a strong Lucas probable prime with Selfridge parameters (*p*,*q*). The Selfridge parameters are chosen by finding the first element *D* in the sequence 5, -7, 9, -11, 13, ... such that  $\text{Jacobi}(D,n) == -1$ . Let *p*=1 and *q* =  $(1-D)/4$  and then perform a strong Lucas probable prime test.

### 19.8.2 Lucas Sequences

An overview is provided by [Joye & Quisquater \(1996\)](#).

#### Function **intLucasU(*p* As mpNum, *q* As mpNum, *k* As mpNum) As mpNum**

The function `intLucasU` returns the *k*-th element of the Lucas U sequence defined by *p*,*q*

#### Parameters:

*p*: An Integer.

*q*: An Integer.

*k*: An Integer.

`lucasu(p,q,k)` will return the *k*-th element of the Lucas U sequence defined by *p,q*.  $p^*p - 4^*q$  must not equal 0; *k* must be greater than or equal to 0.

---

**Function `intLucasModU(p As mpNum, q As mpNum, k As mpNum, n As mpNum) As mpNum`**

---

The function `intLucasModU` returns the *k*-th element of the Lucas U sequence defined by *p,q* ( $\bmod n$ )

**Parameters:**

*p*: An Integer.

*q*: An Integer.

*k*: An Integer.

*n*: An Integer.

`lucasu_mod(p,q,k,n)` will return the *k*-th element of the Lucas U sequence defined by *p,q* ( $\bmod n$ ).  $p^*p - 4^*q$  must not equal 0; *k* must be greater than or equal to 0; *n* must be greater than 0.

---

**Function `intLucasV(p As mpNum, q As mpNum, k As mpNum) As mpNum`**

---

The function `intLucasV` returns the *k*-th element of the Lucas V sequence defined by *p,q*

**Parameters:**

*p*: An Integer.

*q*: An Integer.

*k*: An Integer.

`lucasv(p,q,k)` will return the *k*-th element of the Lucas V sequence defined by parameters (*p,q*).  $p^*p - 4^*q$  must not equal 0; *k* must be greater than or equal to 0.

---

**Function `intLucasModV(p As mpNum, q As mpNum, k As mpNum, n As mpNum) As mpNum`**

---

The function `intLucasModV` returns the *k*-th element of the Lucas V sequence defined by *p,q* ( $\bmod n$ )

**Parameters:**

*p*: An Integer.

*q*: An Integer.

*k*: An Integer.

*n*: An Integer.

`lucasv_mod(p,q,k,n)` will return the *k*-th element of the Lucas V sequence defined by parameters (*p,q*) ( $\bmod n$ ).  $p^*p - 4^*q$  must not equal 0; *k* must be greater than or equal to 0; *n* must be greater than 0.

## 19.9 Random Numbers

### 19.9.1 intUrandomb

---

Function **intUrandomb(*n* As mpNum) As mpNum**

---

The function **intUrandomb** returns a uniformly distributed random integer in the range 0 to  $2^n - 1$ , inclusive.

**Parameter:**

*n*: An Integer.

### 19.9.2 intUrandomm

---

Function **intUrandomm(*n* As mpNum) As mpNum**

---

The function **intUrandomm** returns a uniformly distributed random integer in the range 0 to  $n - 1$ , inclusive.

**Parameter:**

*n*: An Integer.

### 19.9.3 intRrandomb

---

Function **intRrandomb(*n* As mpNum) As mpNum**

---

The function **intRrandomb** returns a random integer with long strings of zeros and ones in the binary representation.

**Parameter:**

*n*: An Integer.

Useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs. The random number will be in the range 0 to  $2^n - 1$ , inclusive.

## 19.10 Information Functions for Integers

### 19.10.1 Congruence: IsCongruent(*n*, *c*, *d*)

---

Function **IsCongruent(*n* As mpNum, *d* As mpNum, *m* As mpNum) As mpNum**

---

The function **IsCongruent** returns TRUE if *n* is congruent to *c* modulo *d*, and FALSE otherwise.

**Parameters:**

*n*: An Integer.

*d*: An Integer.

*m*: An Integer.

Returns TRUE if *n* is congruent to *c* modulo *d*, and FALSE otherwise.

*n* is congruent to *c* mod *d* if there exists an integer *q* satisfying  $n = c + qd$ .

Unlike the other division functions,  $d = 0$  is accepted and following the rule it can be seen that  $n$  and  $c$  are considered congruent mod 0 only when exactly equal.

### 19.10.2 Congruence 2n: IsCongruent2exp( $n, c, b$ )

---

Function **IsCongruent2exp**( $n$  As mpNum,  $c$  As mpNum,  $b$  As mpNum) As mpNum

---

The function **IsCongruent2exp** returns TRUE if  $n$  is congruent to  $c$  modulo  $d$ , and FALSE otherwise.

**Parameters:**

$n$ : An Integer.

$c$ : An Integer.

$b$ : An Integer.

$n$  is congruent to  $c$  mod  $d$  if there exists an integer  $q$  satisfying  $n = c + qd$ .

Unlike the other division functions,  $d = 0$  is accepted and following the rule it can be seen that  $n$  and  $c$  are considered congruent mod 0 only when exactly equal.

### 19.10.3 Primality Testing: IsProbablyPrime( $n, reps$ )

---

Function **IsProbablyPrime**( $n$  As mpNum,  $reps$  As mpNum) As mpNum

---

The function **IsProbablyPrime** returns 2 if  $n$  is definitely prime, returns 1 if  $n$  is probably prime (without being certain), and returns 0 if  $n$  is definitely composite.

**Parameters:**

$n$ : An Integer.

$reps$ : An Integer.

This function does some trial divisions, then some Miller-Rabin probabilistic primality tests.

The argument  $reps$  controls how many such tests are done; a higher value will reduce the chances of a composite being returned as “probably prime”. 25 is a reasonable number; a composite number will then be identified as a prime with a probability of less than  $2^{-50}$ . Miller-Rabin and similar tests can be more properly called compositeness tests. Numbers which fail are known to be composite but those which pass might be prime or might be composite. Only a few composites pass, hence those which pass are considered probably prime.

### 19.10.4 Divisibility: IsDivisible( $n, d$ )

---

Function **IsDivisible**( $n$  As mpNum,  $d$  As mpNum) As mpNum

---

The function **IsDivisible** returns TRUE if  $n$  is exactly divisible by  $d$ .

**Parameters:**

$n$ : An Integer.

$d$ : An Integer.

$n$  is divisible by  $d$  if there exists an integer  $q$  satisfying  $n = qd$ .

Unlike the other division functions,  $d = 0$  is accepted and following the rule it can be seen that only 0 is considered divisible by 0.

### 19.10.5 Divisibility by (2 pow b): IsDivisible2exp( $n$ , $b$ )

---

Function **IsDivisible2exp( $n$  As mpNum,  $b$  As mpNum) As mpNum**

---

The function **IsDivisible2exp** returns TRUE if  $n$  is exactly divisible by  $2^b$ .

**Parameters:**

- $n$ : An Integer.  
 $b$ : An Integer.

$n$  is divisible by  $d$  if there exists an integer  $q$  satisfying  $n = qd$ .

Unlike the other division functions,  $d = 0$  is accepted and following the rule it can be seen that only 0 is considered divisible by 0.

### 19.10.6 Perfect Power: IsPerfectPower( $n$ )

---

Function **IsPerfectPower( $n$  As mpNum) As mpNum**

---

The function **IsPerfectPower** returns TRUE if  $n$  is a perfect power.

**Parameter:**

- $n$ : An Integer.

Returns TRUE if  $n$  is a perfect power, i.e., if there exist integers  $a$  and  $b$ , with  $b > 1$ , such that  $n = a^b$ . Under this definition both 0 and 1 are considered to be perfect powers. Negative values of  $n$  are accepted, but of course can only be odd perfect powers.

### 19.10.7 Perfect Square: IsPerfectSquare( $n$ )

---

Function **IsPerfectSquare( $n$  As mpNum) As mpNum**

---

The function **IsPerfectSquare** returns non-zero if  $n$  is a perfect square.

**Parameter:**

- $n$ : An Integer.

Returns non-zero if  $n$  is a perfect square, i.e., if the square root of  $n$  is an integer. Under this definition both 0 and 1 are considered to be perfect squares.

# Chapter 20

## FMPQ

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

### 20.0.7.1 mpq input

[Function] `int mpq_set_str (mpq t rop, const char *str, int base)` Set rop from a null-terminated string str in the given base.

The string can be an integer like `41` or a fraction like `41/152`. The fraction must be in canonical form (see Chapter 6 [Rational Number Functions], page 45), or if not then `mpq_canonicalize` must be called.

The numerator and optional denominator are parsed the same as in `mpz_set_str` (see Section 5.2 [Assigning Integers], page 30). White space is allowed in the string, and is simply ignored. The base can vary from 2 to 62, or if base is 0 then the leading characters are used:

`0x` or `0X` for hex, `0b` or `0B` for binary, `0` for octal, or decimal otherwise. Note that this is done separately for the numerator and denominator, so for instance `0xEF/100` is `239/100`, whereas `0xEF/0x100` is `239/256`.

The return value is 0 if the entire string is a valid number, or `1` if not.

### 20.0.7.2 mpq output

[Function] `char * mpq_get_str (char *str, int base, const mpq t op)` Convert op to a string of digits in base base. The base may vary from 2 to 36. The string will be of the form `num/den`, or if the denominator is 1 then just `num`. If str is `NULL`, the result string is allocated using the current allocation function (see Chapter 13 [Custom Allocation], page 86). The block will be `strlen(str)+1` bytes, that being exactly enough for the string and null-terminator. If str is not `NULL`, it should point to a block of storage large enough for the result, that being `mpz_sizeinbase (mpq_numref(op), base) + mpz_sizeinbase (mpq_denref(op), base) + 3`. The three extra bytes are for a possible minus sign, possible slash, and the null-terminator. A pointer to the result string is returned, being either the allocated block, or the given str.

# Chapter 21

## ARB

### 21.1 Multiprecision Ball Arithmetic (ARB)

MPFI (Multiple Precision Floating-Point Interval Library) is a library for arbitrary precision interval arithmetic with intervals represented using MPFR reliable floating-point numbers. It is based on the GNU MP library and on the MPFR library. The purpose of an arbitrary precision interval arithmetic is on the one hand to get guaranteed results, thanks to interval computation, and on the other hand to obtain accurate results, thanks to multiple precision arithmetic. The MPFI library is built upon MPFR to benefit from the correct roundings provided by MPFR, its portability, and its compliance with the IEEE 754 standard for floating-point arithmetic.

References for MPFI: [Revol & Rouillier \(2005\)](#), [Moore R. E. \(2009\)](#), [Hayes \(2003\)](#), and [Rump \(1999\)](#)

References for C-XSC 2.0 [Hofschuster & Krämer \(2004\)](#)

Manual for MPFR/MPFI version: [Blomquist et al. \(2012\)](#)

C++ Toolbox for Verified Scientific Computing I: [Hammer et al. \(1995\)](#)

C++ Toolbox for Verified Scientific Computing II: [Krämer et al. \(1994\)](#) and [Krämer et al. \(2006\)](#)

PASCAL-XSC Language Reference: [Klatte et al. \(1991\)](#)

speziellen Funktionen der mathematischen Physik: [Hofschuster \(2000\)](#)

Integration: [Wedner \(2000\)](#)

A priori error estimates : [Blomquist \(2005\)](#)

Other papers: [Blomquist et al. \(2008b\)](#) and [Blomquist et al. \(2008a\)](#) with detailed description of extended complex interval arithmetic.

and [Krämer et al. \(2012\)](#)

#### 21.1.0.3 mpfi input

```
int mpfi_set_str (mpfi t rop, char *s, int base)
```

Sets rop to the value of the string s, in base base (between 2 and 36), outward rounded to the precision of rop: op then belongs to rop. The exponent is read in decimal. The string is of the form "number" or "[ number1 , number2 ]". Each endpoint has the form "M@N" or, if the base is 10 or less, alternatively "MeN" or "MEN". "M" is the mantissa and "N" is the exponent. The mantissa is always in the specified base. The exponent is in decimal. The argument base may be in the ranges 2 to 36.

This function returns 1 if the input is incorrect, and 0 otherwise.

### 21.1.0.4 mpfi output

size\_t mpfi\_out\_str (FILE \*stream, int base, size\_t n\_digits, mpfi\_t op)

Outputs op on stdio stream stream, as a string of digits in base base. The output is an opening square bracket "[", followed by the lower endpoint, a separating comma, the upper endpoint and a closing square bracket "]".

For each endpoint, the output is performed by mpfr\_out\_str. The following piece of information is taken from MPFR documentation. The base may vary from 2 to 36. For each endpoint, it prints at most n digits significant digits, or if n digits is 0, the maximum number of digits accurately representable by op. In addition to the significant digits, a decimal point at the right of the first digit and a trailing exponent, in the form "eNNN", are printed. If base is greater than 10, "@" will be used instead of "e" as exponent delimiter.

Returns the number of bytes written, or if an error occurred, return 0.

As mpfi\_out\_str outputs an enclosure of the input interval, and as mpfi\_inp\_str provides an enclosure of the interval it reads, these functions are not reciprocal. More precisely, when they are called one after the other, the resulting interval contains the initial one, and this inclusion may be strict.

## 21.2 Information Functions for Intervals

### 21.2.1 IsEmpty

---

Function **IsEmpty**(*x* As *mpNum*) As *mpNum*

---

The function IsEmpty returns TRUE if *x* is empty (its endpoints are in reverse order), and FALSE otherwise.

**Parameter:**

*x*: A real number.

Nothing is done in arithmetic or special functions to handle empty intervals: it is the responsibility of the user to avoid computing with empty intervals.

### 21.2.2 IsInside

---

Function **IsInside**(*x* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function IsInside returns TRUE if *x* is contained in *y*, and FALSE otherwise.

**Parameters:**

*x*: A real number.

*y*: A real number.

Returns FALSE if at least one argument is NaN or an invalid interval.

### 21.2.3 IsStrictlyInside

---

Function **IsStrictlyInside**(*x* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function IsStrictlyInside returns TRUE if the second interval *y* is contained in the interior of *x*, and FALSE otherwise.

**Parameters:**

*x*: A real number.  
*y*: A real number.

### 21.2.4 IsStrictlyNeg

---

Function **IsStrictlyNeg**(*x* As *mpNum*) As *mpNum*

---

The function **IsStrictlyNeg** returns TRUE if *x* contains only negative numbers, and FALSE otherwise.

**Parameter:**

*x*: A real number.

### 21.2.5 IsStrictlyPos

---

Function **IsStrictlyPos**(*x* As *mpNum*) As *mpNum*

---

The function **IsStrictlyPos** returns TRUE if *x* contains only positive numbers, and FALSE otherwise.

**Parameter:**

*x*: A real number.

# Chapter 22

ACB

# **Part V**

## **Eigen and related libraries**

# Chapter 23

## BLAS Support (based on Eigen)

The Eigen reference is [Guennebaud \*et al.\* \(2010\)](#)

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. Specifications for Level 1, Level 2 and Level 3 BLAS can be found in [Dongarra \*et al.\* \(1988, 1990\)](#); [Lawson \*et al.\* \(1979\)](#).

Based on BLAS in LAPACK, which is described in [Anderson \*et al.\* \(1999\)](#); [Barker \*et al.\* \(2001\)](#).

The library provides high-level interface for blas operations on vectors and matrices. This should satisfy the needs of most users. Note that currently 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 will be available in later versions of the library.

There are three levels of blas operations,

Level 1: Vector operations, e.g.  $y = ax + y$

Level 2: Matrix-vector operations, e.g.  $y = \hat{\|}Ax + \hat{\|}sy$

Level 3: Matrix-matrix operations, e.g.  $C = \hat{\|}AB + C$

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,

DOT scalar product,  $x^T y$

AXPY vector sum,  $\hat{\|}sx + y$

MV matrix-vector product,  $Ax$

SV matrix-vector solve,  $\text{inv}(A)x$

MM matrix-matrix product,  $AB$

SM matrix-matrix solve,  $\text{inv}(A)B$

The types of matrices are,

GE general

GB general band

SY symmetric

SB symmetric band

SP symmetric packed

HE hermitian

HB hermitian band

HP hermitian packed

TR triangular

TB triangular band

TP triangular packed

Each operation is defined for four precisions,

S single real

D double real

C single complex

Z 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.

Book reference: [Golub & Van Loan \(1996\)](#)

Book reference: [Bernstein \(2009\)](#)

Book reference: [Seber \(2008\)](#)

## 23.1 BLAS Level 1 Support and related Functions

### 23.1.1 Vector-Vector Product

---

#### Function **RDot**(*x* As *mpNum[]*, *y* As *mpNum[]*) As *mpNum*

---

The function RDot returns the real scalar product  $\mathbf{x}^T \mathbf{y}$  for the real vectors  $\mathbf{x}$  and  $\mathbf{y}$ .

**Parameters:**

*x*: A vector of real numbers.

*y*: A vector of real numbers.

---

#### Function **cplxDotu**(*x* As *mpNum[]*, *y* As *mpNum[]*) As *mpNum*

---

The function cplxDotu returns the complex scalar product  $\mathbf{x}^T \mathbf{y}$  for the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ .

**Parameters:**

*x*: A vector of complex numbers.

*y*: A vector of complex numbers.

---

#### Function **cplxDotc**(*x* As *mpNum[]*, *y* As *mpNum[]*) As *mpNum*

---

The function cplxDotc returns the complex conjugate scalar product  $\mathbf{x}^H \mathbf{y}$  for the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ .

**Parameters:**

*x*: A vector of complex numbers.

*y*: A vector of complex numbers.

### 23.1.2 Euclidian Norm

---

#### Function **RNrm2**(*x* As *mpNum[]*, *y* As *mpNum[]*) As *mpNum*

---

The function RNrm2 returns the Euclidean norm  $\|\mathbf{x}\|_2$  of the real vector  $\mathbf{x}$ .

**Parameters:**

*x*: A vector of real numbers.  
*y*: A vector of real numbers.

---

**Function `cplxNrm2(x As mpNum[], y As mpNum[])` As mpNum**

---

The function `cplxNrm2` returns the Euclidean norm  $\|\mathbf{x}\|_2$  of the complex vector  $\mathbf{x}$ .

**Parameters:**

*x*: A vector of complex numbers.  
*y*: A vector of complex numbers.

### 23.1.3 Absolute Sum

---

**Function `RAsum(x As mpNum[], y As mpNum[])` As mpNum**

---

The function `RAsum` returns the the absolute sum of the elements of the real vector  $\mathbf{x}$ .

**Parameters:**

*x*: A vector of real numbers.  
*y*: A vector of real numbers.

---

**Function `cplxAsum(x As mpNum[], y As mpNum[])` As mpNum**

---

The function `cplxAsum` returns the sum of the magnitudes of the real and imaginary parts of the complex vector  $\mathbf{x}$ .

**Parameters:**

*x*: A vector of complex numbers.  
*y*: A vector of complex numbers.

### 23.1.4 Addition

---

**Function `RAxpy(alpha As mpNum, x As mpNum[], y As mpNum[])` As mpNum**

---

The function `RAxpy` returns the sum  $\alpha\mathbf{x} + \mathbf{y}$  for the real scalar  $\alpha$  and the real vectors  $\mathbf{x}$  and  $\mathbf{y}$ .

**Parameters:**

$\alpha$ : A real scalar.  
*x*: A vector of real numbers.  
*y*: A vector of real numbers.

---

**Function `cplxAxpyp(alpha As mpNum, x As mpNum[], y As mpNum[])` As mpNum**

---

The function `cplxAxpyp` returns the sum  $\alpha\mathbf{x} + \mathbf{y}$  for the complex scalar  $\alpha$  and the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ .

**Parameters:**

$\alpha$ : A complex scalar.  
*x*: A vector of complex numbers.  
*y*: A vector of complex numbers.

## 23.2 BLAS Level 2 Support

### 23.2.1 Matrix-Vector Product and Sum (General Matrix)

---

Function **RGemv**(*TransA* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum $[,]$ ,  $\mathbf{x}$  As mpNum $[]$ ,  $\beta$  As mpNum,  $\mathbf{y}$  As mpNum $[]$ ) As mpNum

---

The function RGemv returns the matrix-vector product and sum for a general matrix.

**Parameters:**

*TransA*: An indicator specifying the multiplication.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\mathbf{x}$ : A vector of real numbers.

$\beta$ : A real scalar.

$\mathbf{y}$ : A vector of real numbers.

For the real scalars  $\alpha$  and  $\beta$ , the real vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the real matrix  $\mathbf{A}$ , the function RGemv computes the matrix-vector product and sum

$$\text{RGemv} = \begin{cases} \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}, & \text{for } \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{A}^T \mathbf{x} + \beta \mathbf{y}, & \text{for } \text{TransA} = \text{mpBlasTrans}. \end{cases} \quad (23.2.1)$$

---

Function **cplxGemv**(*TransA* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum $[,]$ ,  $\mathbf{x}$  As mpNum $[]$ ,  $\beta$  As mpNum,  $\mathbf{y}$  As mpNum $[]$ ) As mpNum

---

The function cplxGemv returns the matrix-vector product and sum for a general matrix.

**Parameters:**

*TransA*: An indicator specifying the multiplication.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\mathbf{x}$ : A vector of complex numbers.

$\beta$ : A complex scalar.

$\mathbf{y}$ : A vector of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the complex matrix  $\mathbf{A}$ , the function cplxGemv computes the matrix-vector product and sum

$$\text{cplxGemv} = \begin{cases} \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}, & \text{for } \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{A}^T \mathbf{x} + \beta \mathbf{y}, & \text{for } \text{TransA} = \text{mpBlasTrans}, \\ \alpha \mathbf{A}^H \mathbf{x} + \beta \mathbf{y}, & \text{for } \text{TransA} = \text{mpBlasConjTrans}. \end{cases} \quad (23.2.2)$$

### 23.2.2 Matrix-Vector Product (Triangular Matrix)

---

Function **RTrmv**(*Uplo* As Integer, *TransA* As Integer, *Diag* As Integer, *A* As mpNum[,], *x* As mpNum[]) As mpNum

---

The function **RTrmv** returns the matrix-vector product for a triangular matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

*Diag*: An indicator specifying the use of the diagonal.

*A*: A matrix of real numbers.

*x*: A vector of real numbers.

For the real vector *x* and the real triangular matrix *A*, the function **RTrmv** computes the matrix-vector product

$$RTrmv = \begin{cases} Ax, & \text{for } TransA = mpBlasNoTrans, \\ A^T x, & \text{for } TransA = mpBlasTrans. \end{cases} \quad (23.2.3)$$

When *Uplo* is 0 then the upper triangle of *A* is used, and when *Uplo* is 1 then the lower triangle of *A* is used. If *Diag* is 0 then the diagonal of the matrix is used, but if *Diag* is 1 then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

---

Function **cplxTrmv**(*Uplo* As Integer, *TransA* As Integer, *Diag* As Integer, *A* As mpNum[,], *x* As mpNum[]) As mpNum

---

The function **cplxTrmv** returns the matrix-vector product for a triangular matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

*Diag*: An indicator specifying the use of the diagonal.

*A*: A matrix of complex numbers.

*x*: A vector of complex numbers.

For the complex vector *x* and the complex triangular matrix *A*, the function **cplxTrmv** computes the matrix-vector product

$$cplxTrmv = \begin{cases} Ax, & \text{for } TransA = mpBlasNoTrans, \\ A^T x, & \text{for } TransA = mpBlasTrans, \\ A^H x, & \text{for } TransA = mpBlasConjTrans. \end{cases} \quad (23.2.4)$$

When *Uplo* is 0 then the upper triangle of *A* is used, and when *Uplo* is 1 then the lower triangle of *A* is used. If *Diag* is 0 then the diagonal of the matrix is used, but if *Diag* is 1 then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

### 23.2.3 Inverse Matrix-Vector Product (Triangular Matrix)

---

Function **RTrsv**(*Uplo* As Integer, *TransA* As Integer, *Diag* As Integer, *A* As mpNum[,], *x* As mpNum[]) As mpNum

---

The function **RTrsv** returns the inverse matrix-vector product for a triangular matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

*Diag*: An indicator specifying the use of the diagonal.

*A*: A matrix of real numbers.

*x*: A vector of real numbers.

For the real vector *x* and the real triangular matrix *A*, the function **RTrsv** computes the matrix-vector product

$$RTrsv = \begin{cases} \mathbf{A}^{-1} \mathbf{x}, & \text{for } \text{TransA} = \text{mpBlasNoTrans}, \\ (\mathbf{A}^T)^{-1} \mathbf{x}, & \text{for } \text{TransA} = \text{mpBlasTrans}. \end{cases} \quad (23.2.5)$$

When *Uplo* is 0 then the upper triangle of *A* is used, and when *Uplo* is 1 then the lower triangle of *A* is used. If *Diag* is 0 then the diagonal of the matrix is used, but if *Diag* is 1 then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

---

Function **cplxTrsv**(*Uplo* As Integer, *TransA* As Integer, *Diag* As Integer, *A* As mpNum[,], *x* As mpNum[]) As mpNum

---

The function **cplxTrsv** returns the inverse matrix-vector product for a triangular matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

*Diag*: An indicator specifying the use of the diagonal.

*A*: A matrix of complex numbers.

*x*: A vector of complex numbers. For the complex vector *x* and the complex triangular matrix

*A*, the function **cplxTrsv** computes the matrix-vector product

$$cplxTrsv = \begin{cases} \mathbf{A}^{-1} \mathbf{x} & \text{for } \text{TransA} = \text{mpBlasNoTrans}, \\ (\mathbf{A}^T)^{-1} \mathbf{x}, & \text{for } \text{TransA} = \text{mpBlasTrans}, \\ (\mathbf{A}^H)^{-1} \mathbf{x}, & \text{for } \text{TransA} = \text{mpBlasConjTrans}. \end{cases} \quad (23.2.6)$$

When *Uplo* is 0 then the upper triangle of *A* is used, and when *Uplo* is 1 then the lower triangle of *A* is used. If *Diag* is 0 then the diagonal of the matrix is used, but if *Diag* is 1 then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

### 23.2.4 Matrix-Vector Product and Sum (Symmetric/Hermitian Matrix)

---

Function **RSymv**(*Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[],  $\mathbf{x}$  As mpNum[],  $\beta$  As mpNum,  $\mathbf{y}$  As mpNum[]) As mpNum

---

The function **RSymv** returns the matrix-vector product and sum for a symmetric matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\mathbf{x}$ : A vector of real numbers.

$\beta$ : A real scalar.

$\mathbf{y}$ : A vector of real numbers.

For the real scalars  $\alpha$  and  $\beta$ , the real vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the real symmetric matrix  $\mathbf{A}$ , the function **RSymv** computes the matrix-vector product and sum

$$\text{RSymv} = \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}. \quad (23.2.7)$$

When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used.

---

Function **cplxHemv**(*Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[],  $\mathbf{x}$  As mpNum[],  $\beta$  As mpNum,  $\mathbf{y}$  As mpNum[]) As mpNum

---

The function **cplxHemv** returns the matrix-vector product and sum for a hermitian matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\mathbf{x}$ : A vector of complex numbers.

$\beta$ : A complex scalar.

$\mathbf{y}$ : A vector of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the complex hermitian matrix  $\mathbf{A}$ , the function **cplxHemv** computes the matrix-vector product and sum

$$\text{cplxHemv} = \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}. \quad (23.2.8)$$

When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used.

In **cplxHemv**, the imaginary elements of the diagonal are automatically assumed to be zero and are not referenced.

### 23.2.5 Rank-1 update (General Matrix)

---

Function **RGer**( $\alpha$  As *mpNum*,  $\mathbf{x}$  As *mpNum*[],  $\mathbf{y}$  As *mpNum*[],  $\mathbf{A}$  As *mpNum*[,]) As *mpNum*

---

The function **RGer** returns the rank-1 update for a general matrix

**Parameters:**

- $\alpha$ : A real scalar.
- $\mathbf{x}$ : A vector of real numbers.
- $\mathbf{y}$ : A vector of real numbers.
- $\mathbf{A}$ : A matrix of real numbers.

For the real scalar  $\alpha$ , the real vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the real general matrix  $\mathbf{A}$ , the function **RGer** computes the rank-1 update of the matrix  $\mathbf{A}$ , defined as

$$\text{RGer} = \alpha \mathbf{x} \mathbf{y}^T + \mathbf{A}. \quad (23.2.9)$$

---

Function **cplxGeru**( $\alpha$  As *mpNum*,  $\mathbf{x}$  As *mpNum*[],  $\mathbf{y}$  As *mpNum*[],  $\mathbf{A}$  As *mpNum*[,]) As *mpNum*

---

The function **cplxGeru** returns the rank-1 update for a general matrix

**Parameters:**

- $\alpha$ : A complex scalar.
- $\mathbf{x}$ : A vector of complex numbers.
- $\mathbf{y}$ : A vector of complex numbers.
- $\mathbf{A}$ : A matrix of complex numbers.

For the complex scalar  $\alpha$ , the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the complex general matrix  $\mathbf{A}$ , the function **cplxGeru** computes the rank-1 update of the matrix  $\mathbf{A}$ , defined as

$$\text{cplxGeru} = \alpha \mathbf{x} \mathbf{y}^T + \mathbf{A}. \quad (23.2.10)$$

---

Function **cplxGerc**( $\alpha$  As *mpNum*,  $\mathbf{x}$  As *mpNum*[],  $\mathbf{y}$  As *mpNum*[],  $\mathbf{A}$  As *mpNum*[,]) As *mpNum*

---

The function **cplxGerc** returns the rank-1 update for a general matrix

**Parameters:**

- $\alpha$ : A complex scalar.
- $\mathbf{x}$ : A vector of complex numbers.
- $\mathbf{y}$ : A vector of complex numbers.
- $\mathbf{A}$ : A matrix of complex numbers.

For the complex scalar  $\alpha$ , the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the complex general matrix  $\mathbf{A}$ , the function **cplxGerc** computes the rank-1 update of the matrix  $\mathbf{A}$ , defined as

$$\text{cplxGerc} = \alpha \mathbf{x} \mathbf{y}^H + \mathbf{A}. \quad (23.2.11)$$

### 23.2.6 Rank-1 update (Symmetric/Hermitian Matrix)

---

Function **RSyr**(*Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{x}$  As mpNum[],  $\mathbf{A}$  As mpNum[, $\mathbf{j}$ ]) As mpNum

---

The function **RSyr** returns the Rank-1 update for a symmetric matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A real scalar.

$\mathbf{x}$ : A vector of real numbers.

$\mathbf{A}$ : A matrix of real numbers.

For the real scalar  $\alpha$ , the real vector  $\mathbf{x}$ , and the real symmetric matrix  $\mathbf{A}$ , the function **RSyr** computes the symmetric rank-1 update of the matrix  $\mathbf{A}$ , defined as

$$\text{RSyr} = \alpha \mathbf{x} \mathbf{x}^T + \mathbf{A}. \quad (23.2.12)$$

Since the matrix  $\mathbf{A}$  is symmetric, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used.

---

Function **cplxHer**(*Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{x}$  As mpNum[],  $\mathbf{A}$  As mpNum[, $\mathbf{j}$ ]) As mpNum

---

The function **cplxHer** returns the Rank-1 update for a hermitian matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A complex scalar.

$\mathbf{x}$ : A vector of complex numbers.

$\mathbf{A}$ : A matrix of complex numbers.

For the complex scalar  $\alpha$ , the complex vector  $\mathbf{x}$ , and the complex hermitian matrix  $\mathbf{A}$ , the function **cplxHer** computes the hermitian rank-1 update of the matrix  $\mathbf{A}$ , defined as

$$\text{cplxHer} = \alpha \mathbf{x} \mathbf{x}^H + \mathbf{A}. \quad (23.2.13)$$

Since the matrix  $\mathbf{A}$  is hermitian, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used. The imaginary elements of the diagonal are automatically set to zero.

### 23.2.7 Rank-2 update (Symmetric/Hermitian Matrix)

---

Function **RSyr2**(*Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{x}$  As mpNum[],  $\mathbf{y}$  As mpNum[],  $\mathbf{A}$  As mpNum[,]) As mpNum

---

The function RSyr2 returns the Rank-1 update for a symmetric matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A real scalar.

$\mathbf{x}$ : A vector of real numbers.

$\mathbf{y}$ : A vector of real numbers.

$\mathbf{A}$ : A matrix of real numbers.

For the real scalar  $\alpha$ , the real vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the real symmetric matrix  $\mathbf{A}$ , the function RSyr2 computes the symmetric rank-1 update of the matrix  $\mathbf{A}$ , defined as

$$\text{RSyr2} = \alpha \mathbf{x} \mathbf{y}^T + \alpha \mathbf{y} \mathbf{x}^T + \mathbf{A}. \quad (23.2.14)$$

Since the matrix  $\mathbf{A}$  is symmetric, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used.

---

Function **cplxHer2**(*Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{x}$  As mpNum[],  $\mathbf{y}$  As mpNum[],  $\mathbf{A}$  As mpNum[,]) As mpNum

---

The function cplxHer2 returns the Rank-1 update for a hermitian matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A complex scalar.

$\mathbf{x}$ : A vector of complex numbers.

$\mathbf{y}$ : A vector of complex numbers.

$\mathbf{A}$ : A matrix of complex numbers.

For the complex scalar  $\alpha$ , the complex vectors  $\mathbf{x}$  and  $\mathbf{y}$ , and the complex hermitian matrix  $\mathbf{A}$ , the function cplxHer2 computes the hermitian rank-1 update of the matrix  $\mathbf{A}$ , defined as

$$\text{cplxHer2} = \alpha \mathbf{x} \mathbf{y}^H + \alpha^* \mathbf{y} \mathbf{x}^H + \mathbf{A}. \quad (23.2.15)$$

Since the matrix  $\mathbf{A}$  is hermitian, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used. The imaginary elements of the diagonal are automatically set to zero.

## 23.3 BLAS Level 3 Support

### 23.3.1 Matrix-Matrix-Product and Sum (General Matrix $\mathbf{A}$ )

---

Function **RGemm**(*TransA* As Integer, *TransB* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[,]) As mpNum

---

The function `RGemm` returns the matrix-matrix product and sum for a general matrix.

**Parameters:**

*TransA*: An indicator specifying the multiplication.

*TransB*: An indicator specifying the multiplication.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\mathbf{B}$ : A matrix of real numbers.

$\beta$ : A real scalar.

$\mathbf{C}$ : A matrix of real numbers.

For the real scalars  $\alpha$  and  $\beta$ , and the real matrices  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\mathbf{C}$ , the function `RGemm` computes the matrix-matrix product and sum

$$\text{RGemm} = \begin{cases} \alpha \mathbf{AB} + \beta \mathbf{C}, & \text{for TransA} = \text{mpBlasNoTrans}, \text{TransB} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{AB}^T + \beta \mathbf{C}, & \text{for TransA} = \text{mpBlasNoTrans}, \text{TransB} = \text{mpBlasTrans}, \\ \alpha \mathbf{A}^T \mathbf{B} + \beta \mathbf{C}, & \text{for TransA} = \text{mpBlasTrans}, \text{TransB} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{A}^T \mathbf{B}^T + \beta \mathbf{C}, & \text{for TransA} = \text{mpBlasTrans}, \text{TransB} = \text{mpBlasTrans}, \end{cases} \quad (23.3.1)$$

---

Function **cplxGemm**( *TransA* As Integer, *TransB* As Integer,  $\alpha$  As mpNum, **A** As mpNum[,], **B** As mpNum[,],  $\beta$  As mpNum, **C** As mpNum[,]) As mpNum

---

The function **cplxGemm** returns the matrix-matrix product and sum for a general matrix.

**Parameters:**

*TransA*: An indicator specifying the multiplication.

*TransB*: An indicator specifying the multiplication.

$\alpha$ : A real scalar.

**A**: A matrix of real numbers.

**B**: A matrix of real numbers.

$\beta$ : A real scalar.

**C**: A matrix of real numbers.

For the complex scalars  $\alpha$  and  $\beta$ , and the complex matrices **A**, **B**, **C**, the function **cplxGemm** computes the matrix-matrix product and sum

$$\text{cplxGemm} = \begin{cases} \alpha \mathbf{AB} + \beta \mathbf{C}, & \text{for TransA = mpBlasNoTrans, TransB = mpBlasNoTrans,} \\ \alpha \mathbf{AB}^T + \beta \mathbf{C}, & \text{for TransA = mpBlasNoTrans, TransB = mpBlasTrans,} \\ \alpha \mathbf{AB}^H + \beta \mathbf{C}, & \text{for TransA = mpBlasNoTrans, TransB = mpBlasConjTrans,} \\ \alpha \mathbf{A}^T \mathbf{B} + \beta \mathbf{C}, & \text{for TransA = mpBlasTrans, TransB = mpBlasNoTrans,} \\ \alpha \mathbf{A}^T \mathbf{B}^T + \beta \mathbf{C}, & \text{for TransA = mpBlasTrans, TransB = mpBlasTrans,} \\ \alpha \mathbf{A}^T \mathbf{B}^H + \beta \mathbf{C}, & \text{for TransA = mpBlasTrans, TransB = mpBlasConjTrans,} \\ \alpha \mathbf{A}^H \mathbf{B} + \beta \mathbf{C}, & \text{for TransA = mpBlasConjTrans, TransB = mpBlasNoTrans,} \\ \alpha \mathbf{A}^H \mathbf{B}^T + \beta \mathbf{C}, & \text{for TransA = mpBlasConjTrans, TransB = mpBlasTrans,} \\ \alpha \mathbf{A}^H \mathbf{B}^H + \beta \mathbf{C}, & \text{for TransA = mpBlasConjTrans, TransB = mpBlasConjTrans,} \end{cases} \quad (23.3.2)$$

### 23.3.2 Matrix-Matrix-Product and Sum (Symmetric/Hermitian Matrix $\mathbf{A}$ )

---

Function **RSymm**(*Side* As Integer, *Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[,]) As mpNum

---

The function **RSymm** returns the matrix-matrix product and sum for a symmetric matrix.

**Parameters:**

*Side*: An indicator specifying the order of the multiplication.

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\mathbf{B}$ : A matrix of real numbers.

$\beta$ : A real scalar.

$\mathbf{C}$ : A matrix of real numbers.

For the real scalars  $\alpha$  and  $\beta$ , the real symmetric matrix  $\mathbf{A}$ , and the real general matrices  $\mathbf{B}$  and  $\mathbf{C}$ , the function **RSymm** computes the matrix-matrix product and sum

$$\text{RSymm} = \begin{cases} \alpha \mathbf{AB} + \beta \mathbf{C}, & \text{for } \text{Side} = \text{mpBlasLeft} \\ \alpha \mathbf{BA} + \beta \mathbf{C}, & \text{for } \text{Side} = \text{mpBlasRight} \end{cases} \quad (23.3.3)$$

When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used.

---

Function **cplxSymm**(*Side* As Integer, *Uplo* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[,]) As mpNum

---

The function **cplxSymm** returns the matrix-matrix product and sum for a symmetric matrix.

**Parameters:**

*Side*: An indicator specifying the order of the multiplication.

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\mathbf{B}$ : A matrix of complex numbers.

$\beta$ : A complex scalar.

$\mathbf{C}$ : A matrix of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex symmetric matrix  $\mathbf{A}$ , and the complex general matrices  $\mathbf{B}$  and  $\mathbf{C}$ , the function **cplxSymm** computes the matrix-matrix product and sum

$$\text{cplxSymm} = \begin{cases} \alpha \mathbf{AB} + \beta \mathbf{C}, & \text{for } \text{Side} = \text{mpBlasLeft} \\ \alpha \mathbf{BA} + \beta \mathbf{C}, & \text{for } \text{Side} = \text{mpBlasRight} \end{cases} \quad (23.3.4)$$

When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{A}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{A}$  are used.

---

Function **cplxHemm**(*Side* As Integer, *Uplo* As Integer,  $\alpha$  As mpNum, **A** As mpNum[,], **B** As mpNum[,],  $\beta$  As mpNum, **C** As mpNum[,]) As mpNum

---

The function **cplxHemm** returns the matrix-matrix product and sum for a hermitian matrix.

**Parameters:**

*Side*: An indicator specifying the order of the multiplication.

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

$\alpha$ : A complex scalar.

**A**: A matrix of complex numbers.

**B**: A matrix of complex numbers.

$\beta$ : A complex scalar.

**C**: A matrix of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex hermitian matrix **A**, and the complex general matrices **B** and **C**, the function **cplxHemm** computes the matrix-matrix product and sum

$$\text{cplxHemm} = \begin{cases} \alpha \mathbf{AB} + \beta \mathbf{C}, & \text{for } \text{Side} = \text{mpBlasLeft} \\ \alpha \mathbf{BA} + \beta \mathbf{C}, & \text{for } \text{Side} = \text{mpBlasRight} \end{cases} \quad (23.3.5)$$

When *Uplo* is 0 then the upper triangle and diagonal of **A** are used, and when *Uplo* is 1 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.

### 23.3.3 Matrix-Matrix-Product (Triangular Matrix $\mathbf{A}$ )

---

Function **RTrmm**(*Side* As Integer, *Uplo* As Integer, *TransA* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,]) As mpNum

---

The function **RTrmm** returns the matrix-matrix product for a triangular matrix.

**Parameters:**

*Side*: An indicator specifying the order of the multiplication.

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\mathbf{B}$ : A matrix of real numbers.

For the real scalar  $\alpha$ , the real triangular matrix  $\mathbf{A}$ , and the real general matrix  $\mathbf{B}$ , the function **RTrmm** computes the matrix-matrix product

$$\text{RTrmm} = \begin{cases} \alpha \mathbf{AB}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{BA}, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{A}^T \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasTrans}, \\ \alpha \mathbf{B} \mathbf{A}^T, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasTrans}, \end{cases} \quad (23.3.6)$$

When *Uplo* is 0 then the upper triangle of  $\mathbf{A}$  is used, and when *Uplo* is 1 then the lower triangle of  $\mathbf{A}$  is used. If *Diag* is 0 then the diagonal of  $\mathbf{A}$  is used, but if *Diag* is 1 then the diagonal elements of the matrix  $\mathbf{A}$  are taken as unity and are not referenced.

---

Function **cplxTrmm**(*Side* As Integer, *Uplo* As Integer, *TransA* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,]) As mpNum

---

The function **cplxTrmm** returns the matrix-matrix product for a triangular matrix.

**Parameters:**

*Side*: An indicator specifying the order of the multiplication.

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\mathbf{B}$ : A matrix of complex numbers.

For the complex scalar  $\alpha$ , the complex triangular matrix  $\mathbf{A}$ , and the complex general matrix  $\mathbf{B}$ , the function **cplxTrmm** computes the matrix-matrix product

$$\text{cplxTrmm} = \begin{cases} \alpha \mathbf{AB}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{BA}, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{A}^T \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasTrans}, \\ \alpha \mathbf{B} \mathbf{A}^T, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasTrans}, \\ \alpha \mathbf{A}^H \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasConjTrans}, \\ \alpha \mathbf{B} \mathbf{A}^H, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasConjTrans}, \end{cases} \quad (23.3.7)$$

When `Uplo` is 0 then the upper triangle of  $\mathbf{A}$  is used, and when `Uplo` is 1 then the lower triangle of  $\mathbf{A}$  is used. If `Diag` is 0 then the diagonal of  $\mathbf{A}$  is used, but if `Diag` is 1 then the diagonal elements of the matrix  $\mathbf{A}$  are taken as unity and are not referenced.

### 23.3.4 Inverse Matrix-Matrix-Product (Triangular Matrix $\mathbf{A}$ )

---

Function **RTrsm**(*Side* As Integer, *Uplo* As Integer, *TransA* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,]) As mpNum

---

The function **RTrsm** returns the inverse matrix-matrix product for a triangular matrix.

**Parameters:**

*Side*: An indicator specifying the order of the multiplication.

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\mathbf{B}$ : A matrix of real numbers.

For the real scalar  $\alpha$ , the real triangular matrix  $\mathbf{A}$ , and the real general matrix  $\mathbf{B}$ , the function **RTrsm** computes the matrix-matrix product

$$\text{RTrsm} = \begin{cases} \alpha \mathbf{A}^{-1} \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{B} \mathbf{A}^{-1}, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha (\mathbf{A}^T)^{-1} \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasTrans}, \\ \alpha \mathbf{B} (\mathbf{A}^T)^{-1}, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasTrans}, \end{cases} \quad (23.3.8)$$

When *Uplo* is 0 then the upper triangle of  $\mathbf{A}$  is used, and when *Uplo* is 1 then the lower triangle of  $\mathbf{A}$  is used. If *Diag* is 0 then the diagonal of  $\mathbf{A}$  is used, but if *Diag* is 1 then the diagonal elements of the matrix  $\mathbf{A}$  are taken as unity and are not referenced.

---

Function **cplxTrsm**(*Side* As Integer, *Uplo* As Integer, *TransA* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,]) As mpNum

---

The function **cplxTrsm** returns the inverse matrix-matrix product for a triangular matrix.

**Parameters:**

*Side*: An indicator specifying the order of the multiplication.

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*TransA*: An indicator specifying the multiplication.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\mathbf{B}$ : A matrix of complex numbers.

For the complex scalar  $\alpha$ , the complex triangular matrix  $\mathbf{A}$ , and the complex general matrix  $\mathbf{B}$ , the function **cplxTrsm** computes the matrix-matrix product

$$\text{cplxTrsm} = \begin{cases} \alpha \mathbf{A}^{-1} \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha \mathbf{B} \mathbf{A}^{-1}, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasNoTrans}, \\ \alpha (\mathbf{A}^T)^{-1} \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasTrans}, \\ \alpha \mathbf{B} (\mathbf{A}^T)^{-1}, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasTrans}, \\ \alpha (\mathbf{A}^H)^{-1} \mathbf{B}, & \text{for } \text{Side} = \text{mpBlasLeft}, \text{TransA} = \text{mpBlasConjTrans}, \\ \alpha \mathbf{B} (\mathbf{A}^H)^{-1}, & \text{for } \text{Side} = \text{mpBlasRight}, \text{TransA} = \text{mpBlasConjTrans}, \end{cases} \quad (23.3.9)$$

When `Uplo` is 0 then the upper triangle of  $\mathbf{A}$  is used, and when `Uplo` is 1 then the lower triangle of  $\mathbf{A}$  is used. If `Diag` is 0 then the diagonal of  $\mathbf{A}$  is used, but if `Diag` is 1 then the diagonal elements of the matrix  $\mathbf{A}$  are taken as unity and are not referenced.

### 23.3.5 Rank-k update (Symmetric/Hermitian Matrix $\mathbf{C}$ )

---

Function **Rsyrk**(*Uplo* As Integer, *Trans* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[,]) As mpNum

---

The function **Rsyrk** returns a rank-k update for a symmetric matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*Trans*: An indicator specifying the multiplication.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\beta$ : A real scalar.

$\mathbf{C}$ : A matrix of real numbers.

For the real scalars  $\alpha$  and  $\beta$ , the real symmetric matrix  $\mathbf{C}$ , and the real general matrix  $\mathbf{A}$ , the function **Rsyrk** computes a rank-k update of the symmetric matrix  $\mathbf{C}$ , defined as

$$\text{Rsyrk} = \begin{cases} \alpha \mathbf{A} \mathbf{A}^T + \beta \mathbf{C}, & \text{for Trans = mpNoTrans} \\ \alpha \mathbf{A}^T \mathbf{A} + \beta \mathbf{C}, & \text{for Trans = mpTrans} \end{cases} \quad (23.3.10)$$

Since the matrix  $\mathbf{C}$  is symmetric, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{C}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{C}$  are used.

---

Function **cplxSyrk**(*Uplo* As Integer, *Trans* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[,]) As mpNum

---

The function **cplxSyrk** returns a rank-k update for a symmetric matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*Trans*: An indicator specifying the multiplication.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\beta$ : A complex scalar.

$\mathbf{C}$ : A matrix of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex symmetric matrix  $\mathbf{C}$ , and the complex general matrix  $\mathbf{A}$ , the function **cplxSyrk** computes a rank-k update of the complex matrix  $\mathbf{C}$ , defined as

$$\text{cplxSyrk} = \begin{cases} \alpha \mathbf{A} \mathbf{A}^T + \beta \mathbf{C}, & \text{for Trans = mpNoTrans} \\ \alpha \mathbf{A}^T \mathbf{A} + \beta \mathbf{C}, & \text{for Trans = mpTrans} \end{cases} \quad (23.3.11)$$

Since the matrix  $\mathbf{C}$  is symmetric, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{C}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{C}$  are used.

---

Function **cplxHerk**(*Uplo* As Integer, *Trans* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[,]) As mpNum

---

The function **cplxHerk** returns a rank-k update for a hermitian matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*Trans*: An indicator specifying the multiplication.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\beta$ : A complex scalar.

$\mathbf{C}$ : A matrix of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex hermitian matrix  $\mathbf{C}$ , and the complex general matrix  $\mathbf{A}$ , the function **cplxHerk** computes a rank-k update of the hermitian matrix  $\mathbf{C}$ , defined as

$$\text{cplxHerk} = \begin{cases} \alpha \mathbf{A} \mathbf{A}^H + \beta \mathbf{C}, & \text{for } \text{Trans} = \text{mpNoTrans} \\ \alpha \mathbf{A}^H \mathbf{A} + \beta \mathbf{C}, & \text{for } \text{Trans} = \text{mpTrans} \end{cases} \quad (23.3.12)$$

Since the matrix  $\mathbf{C}$  is hermitian, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{C}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{C}$  are used.

### 23.3.6 Rank-2k update (Symmetric/Hermitian Matrix $\mathbf{C}$ )

---

Function **Rsyr2k**(*Uplo* As Integer, *Trans* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[],  $\mathbf{B}$  As mpNum[],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[],) As mpNum

---

The function **Rsyr2k** returns a rank-k update for a symmetric matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*Trans*: An indicator specifying the multiplication.

$\alpha$ : A real scalar.

$\mathbf{A}$ : A matrix of real numbers.

$\mathbf{B}$ : A matrix of real numbers.

$\beta$ : A real scalar.

$\mathbf{C}$ : A matrix of real numbers.

For the real scalars  $\alpha$  and  $\beta$ , the real symmetric matrix  $\mathbf{C}$ , and the real general matrices  $\mathbf{A}$  and  $\mathbf{B}$ , the function **Rsyr2k** computes a rank-k update of the symmetric matrix  $\mathbf{C}$ , defined as

$$\text{Rsyr2k} = \begin{cases} \alpha \mathbf{AB}^T + \alpha \mathbf{BA}^T + \beta \mathbf{C}, & \text{for Trans = mpNoTrans} \\ \alpha \mathbf{A}^T \mathbf{B} + \alpha \mathbf{B}^T \mathbf{A} + \beta \mathbf{C}, & \text{for Trans = mpTrans} \end{cases} \quad (23.3.13)$$

Since the matrix  $\mathbf{C}$  is symmetric/hermitian, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{C}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{C}$  are used.

---

Function **cplxSyr2k**(*Uplo* As Integer, *Trans* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[],  $\mathbf{B}$  As mpNum[],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[],) As mpNum

---

The function **cplxSyr2k** returns a rank-k update for a symmetric matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*Trans*: An indicator specifying the multiplication.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\mathbf{B}$ : A matrix of complex numbers.

$\beta$ : A complex scalar.

$\mathbf{C}$ : A matrix of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex symmetric matrix  $\mathbf{C}$ , and the complex general matrices  $\mathbf{A}$  and  $\mathbf{B}$ , the function **cplxSyrk** computes a rank-k update of the complex matrix  $\mathbf{C}$ , defined as

$$\text{cplxSyrk} = \begin{cases} \alpha \mathbf{AB}^T + \alpha \mathbf{BA}^T + \beta \mathbf{C}, & \text{for Trans = mpNoTrans} \\ \alpha \mathbf{A}^T \mathbf{B} + \alpha \mathbf{B}^T \mathbf{A} + \beta \mathbf{C}, & \text{for Trans = mpTrans} \end{cases} \quad (23.3.14)$$

Since the matrix  $\mathbf{C}$  is symmetric/hermitian, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{C}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{C}$  are used.

---

Function **cplxHer2k**(*Uplo* As Integer, *Trans* As Integer,  $\alpha$  As mpNum,  $\mathbf{A}$  As mpNum[,],  $\mathbf{B}$  As mpNum[,],  $\beta$  As mpNum,  $\mathbf{C}$  As mpNum[,]) As mpNum

---

The function **cplxHer2k** returns a rank-k update for a hermitian matrix.

**Parameters:**

*Uplo*: An indicator specifying whether the upper or lower triangle will be used.

*Trans*: An indicator specifying the multiplication.

$\alpha$ : A complex scalar.

$\mathbf{A}$ : A matrix of complex numbers.

$\mathbf{B}$ : A matrix of complex numbers.

$\beta$ : A complex scalar.

$\mathbf{C}$ : A matrix of complex numbers.

For the complex scalars  $\alpha$  and  $\beta$ , the complex hermitian matrix  $\mathbf{C}$ , and the complex general matrices  $\mathbf{A}$  and  $\mathbf{B}$ , the function **cplxHer2k** computes a rank-k update of the hermitian matrix  $\mathbf{C}$ , defined as

$$\text{cplxHer2k} = \begin{cases} \alpha \mathbf{A} \mathbf{B}^H + \alpha \mathbf{B} \mathbf{A}^H + \beta \mathbf{C}, & \text{for } \text{Trans} = \text{mpNoTrans} \\ \alpha \mathbf{A}^H \mathbf{B} + \alpha \mathbf{B}^H \mathbf{A} + \beta \mathbf{C}, & \text{for } \text{Trans} = \text{mpTrans} \end{cases} \quad (23.3.15)$$

Since the matrix  $\mathbf{C}$  is symmetric/hermitian, only its upper half or lower half need to be stored. When *Uplo* is 0 then the upper triangle and diagonal of  $\mathbf{C}$  are used, and when *Uplo* is 1 then the lower triangle and diagonal of  $\mathbf{C}$  are used.

# Chapter 24

## Linear Solvers (based on Eigen)

Book reference: [Golub & Van Loan \(1996\)](#)

### 24.1 Cholesky Decomposition without Pivoting

#### 24.1.1 Decomposition

---

Function **DecompCholeskyLLT**(*A* As *mpNum*[,], *B* As *mpNum*[,], *UpLo* As Integer, *Output* As String) As *mpNumList*

---

The function **DecompCholeskyLLT** returns the Cholesky decomposition  $A = LL^* = U^*U$  of a matrix.

**Parameters:**

*A*: the real matrix of which we are computing the  $LL^T$  Cholesky decomposition.

*B*: A vector or matrix of real numbers.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

*Output*: A string specifying the output options.

---

Function **cplxDecompCholeskyLLT**(*A* As *mpNum*[,], *B* As *mpNum*[,], *UpLo* As Integer, *Output* As String) As *mpNumList*

---

The function **cplxDecompCholeskyLLT** returns the Cholesky decomposition  $A = LL^* = U^*U$  of a matrix.

**Parameters:**

*A*: the complex matrix of which we are computing the  $LL^T$  Cholesky decomposition.

*B*: A vector or complex of real numbers.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

*Output*: A string specifying the output options.

These functions perform a  $LL^T$  Cholesky decomposition of a symmetric, positive definite matrix *A* such that  $A = LL^* = U^*U$ , where *L* is lower triangular. While the Cholesky decomposition is particularly useful to solve selfadjoint problems like  $D^*Dx = b$ , for that purpose, we recommend the Cholesky decomposition without square root which is more stable and even faster.

Nevertheless, this standard Cholesky decomposition remains useful in many other situations like generalised eigen problems with hermitian matrices.

Remember that Cholesky decompositions are not rank-revealing. This LLT decomposition is only stable on positive definite matrices, use LDLT instead for the semidefinite case. Also, do not use a Cholesky decomposition to determine whether a system of equations has a solution.

LLT<sub>i</sub> MatrixType, <sub>i</sub> UpLo & compute ( const MatrixType & a)

Computes / recomputes the Cholesky decomposition  $A = LL^* = U^*U$  of matrix

Returns: a reference to \*this

**ComputationInfo** info () const: Reports whether previous computation was successful.

Returns: Success if computation was successful, NumericalIssue if the matrix.appears to be negative.

Traits::MatrixL **matrixL** () const

Returns: a view of the lower triangular matrix L

const MatrixType& **matrixLLT** () const inline

Returns: the LLT decomposition matrix

TODO: document the storage layout

Traits::MatrixU **matrixU** () const

Returns: a view of the upper triangular matrix U

LLT<sub>i</sub> MatrixType, <sub>i</sub> UpLo<sub>i</sub> **rankUpdate** ( const VectorType & v, const RealScalar & sigma )

Performs a rank one update (or dowdate) of the current decomposition. If  $A = LL^*$  before the rank one update, then after it we have  $LL^* = A + \sigma \times vv^*$  where v must be a vector of same dimension. References Eigen::NumericalIssue, and Eigen::Success.

MatrixType **reconstructedMatrix** () const

Returns: the matrix represented by the decomposition, i.e., it returns the product:  $LL^*$ . This function is provided for debug purpose.

const internal::solve\_retval<sub>i</sub>LLT, Rhs<sub>i</sub> **solve** ( const MatrixBase<sub>i</sub> Rhs<sub>i</sub> & b) const

Returns: the solution  $x$  of  $Ax = b$  using the current decomposition of  $A$ . Since this LLT class assumes anyway that the matrix  $A$  is invertible, the solution theoretically exists and is unique regardless of  $b$ .

## 24.1.2 Linear Solver

---

Function **SolveCholeskyLLT(A As mpNum[,], B As mpNum[,], UpLo As Integer) As mpNum[]**

---

The function **SolveCholeskyLLT** returns the solution  $x$  of  $Ax = b$ , based on a Cholesky decomposition.

### Parameters:

**A**: A symmetric positive definite real matrix.

**B**: A real vector or matrix.

**UpLo**: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

---

Function **cplxSolveCholeskyLLT**(*A* As *mpNum*[,], *B* As *mpNum*[,], *UpLo* As Integer) As *mpNum*[]

---

The function `cplxSolveCholeskyLLT` returns the solution  $x$  of  $Ax = b$ , based on a Cholesky decomposition.

**Parameters:**

*A*: A symmetric positive definite complex matrix.

*B*: A complex vector or matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

### 24.1.3 Matrix Inversion

---

Function **InvertCholeskyLLT**(*A* As *mpNum*[,], *UpLo* As Integer) As *mpNum*[]

---

The function `InvertCholeskyLLT` returns  $A^{-1}$ , the inverse of *A*, based on a Cholesky decomposition.

**Parameters:**

*A*: A symmetric positive definite real matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

---

Function **cplxInvertCholeskyLLT**(*A* As *mpNum*[,], *UpLo* As Integer) As *mpNum*[]

---

The function `cplxInvertCholeskyLLT` returns  $A^{-1}$ , the inverse of *A*, based on a Cholesky decomposition.

**Parameters:**

*A*: A symmetric positive definite complex matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

### 24.1.4 Determinant

---

Function **DetCholeskyLLT**(*A* As *mpNum*[,], *UpLo* As Integer) As *mpNum*

---

The function `DetCholeskyLLT` returns  $|A|$ , the determinant of *A*, based on a Cholesky decomposition.

**Parameters:**

*A*: A symmetric positive definite real matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

---

Function **cplxDetCholeskyLLT**(*A* As *mpNum*[,], *UpLo* As Integer) As *mpNum*

---

The function `cplxDetCholeskyLLT` returns  $|A|$ , the determinant of *A*, based on a Cholesky decomposition.

**Parameters:**

*A*: A symmetric positive definite complex matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

### 24.1.5 Example

Example:

---

```
MatrixXd A(3,3);
A << 4,-1,2, -1,6,0, 2,0,5;
cout << "The matrix A is" << endl << A << endl;
LLT<MatrixXd> lltOfA(A);
// compute the Cholesky decomposition of A
MatrixXd L = lltOfA.matrixL();
// retrieve factor L in the decomposition
// The previous two lines can also be written as "L = A.llt().matrixL()"
cout << "The Cholesky factor L is" << endl << L << endl;
cout << "To check this, let us compute L * L.transpose()" << endl;
cout << L * L.transpose() << endl;
cout << "This should equal the matrix A" << endl;
```

---

Output:

The matrix A is

```
4 -1 2
-1 6 0
2 0 5
```

The Cholesky factor L is

```
2 0 0
-0.5 2.4 0
1 0.209 1.99
```

To check this, let us compute L \* L.transpose()

```
4 -1 2
-1 6 0
2 0 5
```

This should equal the matrix A

## 24.2 Cholesky Decomposition with Pivoting

### 24.2.1 Decomposition

---

Function **DecompCholeskyLDLT**(**A** As *mpNum*[,], **B** As *mpNum*[,], **UpLo** As Integer, **Output** As String) As *mpNumList*

---

The function **DecompCholeskyLDLT** returns the Cholesky decomposition with pivoting of  $A = LL^* = U^*U$ .

**Parameters:**

*A*: the real matrix of which we are computing the  $LL^T$  Cholesky decomposition.

*B*: A vector or matrix of real numbers.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

*Output*: A string specifying the output options.

---

Function **cplxDecompCholeskyLDLT**(**A** As *mpNum*[,], **B** As *mpNum*[,], **UpLo** As Integer, **Output** As String) As *mpNumList*

---

The function **cplxDecompCholeskyLDLT** returns the Cholesky decomposition with pivoting of  $A = LL^* = U^*U$ .

**Parameters:**

*A*: the complex matrix of which we are computing the  $LL^T$  Cholesky decomposition.

*B*: A vector or complex of real numbers.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

*Output*: A string specifying the output options.

Perform a robust Cholesky decomposition of a positive semidefinite or negative semidefinite matrix such that  $A = P^TLDL^*P$ , where  $P$  is a permutation matrix,  $L$  is lower triangular with a unit diagonal and  $D$  is a diagonal matrix.

The decomposition uses pivoting to ensure stability, so that  $L$  will have zeros in the bottom right  $\text{rank}(A) - n$  submatrix. Avoiding the square root on  $D$  also stabilizes the computation.

Remember that Cholesky decompositions are not rank-revealing. Also, do not use a Cholesky decomposition to determine whether a system of equations has a solution.

**LDLT**: MatrixType, **\_UpLo** & **compute** ( const MatrixType & a)

Compute / recompute the LDLT decomposition  $A = LDL^* = U^*DU$  of matrix

ComputationInfo **info** ( ) const

Reports whether previous computation was successful.

Returns Success if computation was succesful, NumericalIssue if the matrix appears to be negative.

bool **isNegative** ( void ) const

Returns true if the matrix is negative (semidefinite)

bool **isPositive** ( ) const

Returns true if the matrix is positive (semidefinite)

Traits::MatrixL **matrixL** ( ) const

Returns a view of the lower triangular matrix L

const MatrixType& **matrixLDLT** ( ) const

Returns the internal LDLT decomposition matrix TODO: document the storage layout

Traits::MatrixU **matrixU** ( ) const

Returns a view of the upper triangular matrix U

LDLT<sub>1</sub>MatrixType, UpLo<sub>1</sub>& **rankUpdate** ( const MatrixBase<sub>1</sub> Derived<sub>1</sub> & w, const typename NumTraits<sub>1</sub> typename MatrixType::Scalar<sub>1</sub>::Real & sigma)

Update the LDLT decomposition: given  $A = LDL^T$ , efficiently compute the decomposition of  $A + \sigma w w^T$ .

Parameters: w a vector to be incorporated into the decomposition. sigma a scalar, +1 for updates and -1 for "downdates," which correspond to removing previously-added column vectors. Optional; default value is +1.

MatrixType **reconstructedMatrix** ( ) const

Returns the matrix represented by the decomposition, i.e., it returns the product:  $P^T LDL^* P$ .

This function is provided for debug purpose.

void **setZero** ( )

Clear any existing decomposition

const internal::solve\_retval<sub>1</sub> LDLT<sub>1</sub>, Rhs<sub>1</sub> **solve** ( const MatrixBase<sub>1</sub> Rhs<sub>1</sub> & b) const

Returns a solution  $x$  of  $Ax = b$  using the current decomposition of  $A$ .

This function also supports in-place solves using the syntax  $x = \text{decompositionObject.solve}(x)$ .

This method just tries to find as good a solution as possible. If you want to check whether a solution exists or if it is accurate, just call this function to get a result and then compute the error of this result, or use MatrixBase::isApprox() directly, for instance like this:

bool a\_solution\_exists = (A \* result).isApprox(b, precision);

This method avoids dividing by zero, so that the non-existence of a solution doesn't by itself mean that you'll get inf or nan values.

More precisely, this method solves  $Ax = b$  using the decomposition  $A = P^T LDL^* P$  by solving the systems  $P^T y_1 = b$ ,  $LY_2 = y_1$ ,  $Dy_3 = y_2$ ,  $L^* y_4 = y_3$  and  $Px = y_4$  in succession. If the matrix  $A$  is singular, then  $D$  will also be singular (all the other matrices are invertible). In that case, the least-square solution of  $Dy_3 = y_2$  is computed. This does not mean that this function computes the least-square solution of  $Ax = b$  if  $A$  is singular.

const TranspositionType& **transpositionsP** ( ) const

Returns the permutation matrix P as a transposition sequence.

Diagonal<sub>1</sub>const MatrixType<sub>1</sub> **vectorD** ( ) const

Returns the coefficients of the diagonal matrix D

## 24.2.2 Linear Solver

---

Function **SolveCholeskyLDLT**(**A** As mpNum[], **B** As mpNum[], **UpLo** As Integer) As mpNum[]

---

The function **SolveCholeskyLDLT** returns the solution  $x$  of  $Ax = b$ , based on a Cholesky decomposition with pivoting.

### Parameters:

**A**: A symmetric positive definite real matrix.

*B*: A real vector or matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

---

Function **cplxSolveCholeskyLDLT**(*A* As *mpNum*[,], *B* As *mpNum*[,], *UpLo* As Integer) As *mpNum*[]

---

The function `cplxSolveCholeskyLDLT` returns the solution  $x$  of  $Ax = b$  , based on a Cholesky decomposition with pivoting.

**Parameters:**

*A*: A symmetric positive definite complex matrix.

*B*: A complex vector or matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

### 24.2.3 Matrix Inversion

---

Function **InvertCholeskyLDLT**(*A* As *mpNum*[,], *UpLo* As Integer) As *mpNum*[]

---

The function `InvertCholeskyLDLT` returns  $A^{-1}$ , the inverse of *A*, based on a Cholesky decomposition with pivoting.

**Parameters:**

*A*: A symmetric positive definite real matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

---

Function **cplxInvertCholeskyLDLT**(*A* As *mpNum*[,], *UpLo* As Integer) As *mpNum*[]

---

The function `cplxInvertCholeskyLDLT` returns  $A^{-1}$ , the inverse of *A*, based on a Cholesky decomposition with pivoting.

**Parameters:**

*A*: A symmetric positive definite complex matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

### 24.2.4 Determinant

---

Function **DetCholeskyLDLT**(*A* As *mpNum*[,], *UpLo* As Integer) As *mpNum*

---

The function `DetCholeskyLDLT` returns  $|A|$ , the determinant of *A*, based on a Cholesky decomposition.

**Parameters:**

*A*: A symmetric positive definite real matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

---

**Function `cplxDetCholeskyLDLT(A As mpNum[,], UpLo As Integer) As mpNum`**


---

The function `cplxDetCholeskyLDLT` returns  $|A|$ , the determinant of  $A$ , based on a Cholesky decomposition.

**Parameters:**

*A*: A symmetric positive definite complex matrix.

*UpLo*: the triangular part that will be used for the decompositon: Lower (default) or Upper. The other triangular part won't be read.

### 24.2.5 Example

Example:

---

```
MatrixXd A(3,3);
A << 4,-1,2, -1,6,0, 2,0,5;
cout << "The matrix A is" << endl << A << endl;
LLT<MatrixXd> lltOfA(A);
// compute the Cholesky decomposition of A
MatrixXd L = lltOfA.matrixL();
// retrieve factor L in the decomposition
// The previous two lines can also be written as "L = A.llt().matrixL()"
cout << "The Cholesky factor L is" << endl << L << endl;
cout << "To check this, let us compute L * L.transpose()" << endl;
cout << L * L.transpose() << endl;
cout << "This should equal the matrix A" << endl;
```

---

Output:

```
The matrix A is
4 -1  2
-1  6  0
2   0  5
The Cholesky factor L is
2   0   0
-0.5 2.4   0
1 0.209 1.99
To check this, let us compute L * L.transpose()
4 -1  2
-1  6  0
2   0  5
This should equal the matrix A
```

## 24.3 LU Decomposition with partial Pivoting

### 24.3.1 Decomposition

---

Function **DecompPartialPivLU**(*A* As *mpNum*[, *B* As *mpNum*[, *Output* As *String*]) As *mpNumList*

---

The function **DecompPartialPivLU** returns the LU decomposition with partial pivoting of  $A = PLU$ .

**Parameters:**

*A*: the square real matrix of which we are computing the *LU* decomposition.

*B*: A vector or matrix of real numbers.

*Output*: A string specifying the output options.

---

Function **cplxDecompPartialPivLU**(*A* As *mpNum*[, *B* As *mpNum*[, *Output* As *String*]) As *mpNumList*

---

The function **cplxDecompPartialPivLU** returns the LU decomposition with partial pivoting of  $A = PLU$ .

**Parameters:**

*A*: the square complex matrix of which we are computing the *LU* decomposition.

*B*: A vector or complex of real numbers.

*Output*: A string specifying the output options.

This class represents a LU decomposition of a square invertible matrix, with partial pivoting: the matrix  $A$  is decomposed as  $A = PLU$  where  $L$  is unit-lower-triangular,  $U$  is upper-triangular, and  $P$  is a permutation matrix.

Typically, partial pivoting LU decomposition is only considered numerically stable for square invertible matrices. Thus LAPACK's **dgesv** and **dgesvx** require the matrix to be square and invertible. The present class does the same. It will assert that the matrix is square, but it won't (actually it can't) check that the matrix is invertible: it is your task to check that you only use this decomposition on invertible matrices.

The guaranteed safe alternative, working for all matrices, is the full pivoting LU decomposition, provided by class **FullPivLU**.

This is not a rank-revealing LU decomposition. Many features are intentionally absent from this class, such as rank computation. If you need these features, use class **FullPivLU**.

This LU decomposition is suitable to invert invertible matrices. It is what **MatrixBase::inverse()** uses in the general case. On the other hand, it is not suitable to determine whether a given matrix is invertible.

The data of the LU decomposition can be directly accessed through the methods **matrixLU()**, **permutationP()**.

Returns the determinant of the matrix of which *\*this* is the LU decomposition. It has only linear complexity (that is,  $O(n)$  where  $n$  is the dimension of the square matrix) as the LU decomposition has already been computed.

Warning: a determinant can be very big or small, so for matrices of large enough dimension, there is a risk of overflow/underflow. See Also **MatrixBase::determinant()**

const internal::solve\_retval **inverse** (

) const

Returns the inverse of the matrix of which *\*this* is the LU decomposition.

Warning: The matrix being decomposed here is assumed to be invertible. If you need to check for invertibility, use class FullPivLU instead.

const MatrixType& **matrixLU** ( ) const

Returns the LU decomposition matrix: the upper-triangular part is U, the unit-lower-triangular part is L (at least for square matrices; in the non-square case, special care is needed, see the documentation of class FullPivLU).

const PermutationType& **permutationP** ( ) const

Returns the permutation matrix P.

MatrixType **reconstructedMatrix** ( ) const

Returns the matrix represented by the decomposition, i.e., it returns the product:  $P^{-1}LU$ . This function is provided for debug purpose.

const internal::solve\_retval<PartialPivLU, Rhs> **solve** ( const MatrixBase<Rhs> & b) const

This method returns the solution  $x$  to the equation  $Ax = b$ , where  $A$  is the matrix of which *\*this* is the LU decomposition.

Parameters:  $b$  the right-hand-side of the equation to solve. Can be a vector or a matrix, the only requirement in order for the equation to make sense is that  $b.rows() == A.rows()$ , where  $A$  is the matrix of which *\*this* is the LU decomposition.

Returns the solution.

### 24.3.2 Linear Solver

---

Function **SolvePartialPivLU**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function **SolvePartialPivLU** returns the solution  $x$  of  $Ax = b$ , based on a LU decomposition with partial pivoting.

**Parameters:**

*A*: A square real matrix.

*B*: A real vector or matrix.

---

Function **cplxSolvePartialPivLU**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function **cplxSolvePartialPivLU** returns the solution  $x$  of  $Ax = b$ , based on a LU decomposition with partial pivoting.

**Parameters:**

*A*: A square complex matrix.

*B*: A complex vector or matrix.

### 24.3.3 Matrix Inversion

---

Function **InvertPartialPivLU**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function **InvertPartialPivLU** returns  $A^{-1}$ , the inverse of  $A$ , based on a LU decomposition with partial pivoting.

**Parameter:**

*A*: A square real matrix.

---

**Function `cplxInvertPartialPivLU(A As mpNum[,]) As mpNum[]`**

---

The function `cplxInvertPartialPivLU` returns  $A^{-1}$ , the inverse of  $A$ , based on a LU decomposition with partial pivoting.

**Parameter:**

*A*: A square complex matrix.

### 24.3.4 Determinant

---

**Function `DetPartialPivLU(A As mpNum[,]) As mpNum`**

---

The function `DetPartialPivLU` returns  $|A|$ , the determinant of  $A$ , based on a LU decomposition with partial pivoting.

**Parameter:**

*A*: A square real matrix.

---

**Function `cplxDetPartialPivLU(A As mpNum[,]) As mpNum`**

---

The function `cplxDetPartialPivLU` returns  $|A|$ , the determinant of  $A$ , based on a LU decomposition with partial pivoting.

**Parameter:**

*A*: A square complex matrix.

### 24.3.5 Example

Example:

---

```
MatrixXd A = MatrixXd::Random(3,3);
MatrixXd B = MatrixXd::Random(3,2);
cout << "Here is the invertible matrix A:" << endl << A << endl;
cout << "Here is the matrix B:" << endl << B << endl;
MatrixXd X = A.lu().solve(B);
cout << "Here is the (unique) solution X to the equation AX=B:" << endl << X << endl;
cout << "Relative error: " << (A*X-B).norm() / B.norm() << endl;
```

---

Output:

Here is the invertible matrix A:

```
0.68  0.597 -0.33
-0.211 0.823  0.536
0.566 -0.605 -0.444
```

Here is the matrix B:

```
0.108  -0.27
-0.0452 0.0268
```

0.258 0.904

Here is the (unique) solution X to the equation AX=B:

0.609 2.68

-0.231 -1.57

0.51 3.51

Relative error: 3.28e-16

## 24.4 LU Decomposition with full Pivoting

### 24.4.1 Decomposition

---

Function **DecompFullPivLU**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function `DecompFullPivLU` returns the LU decomposition with full pivoting of  $A = PLUQ$ .

**Parameters:**

*A*: the square real matrix of which we are computing the *LU* decomposition.

*B*: A vector or matrix of real numbers.

*Output*: A string specifying the output options.

---

Function **cplxDecompFullPivLU**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function `cplxDecompFullPivLU` returns the LU decomposition with full pivoting of  $A = PLUQ$ .

**Parameters:**

*A*: the square complex matrix of which we are computing the *LU* decomposition.

*B*: A vector or complex of real numbers.

*Output*: A string specifying the output options.

This class represents a LU decomposition of any matrix, with complete pivoting: the matrix *A* is decomposed as  $A = PLUQ$  where *L* is unit-lower-triangular, *U* is upper-triangular, and *P* and *Q* are permutation matrices. This is a rank-revealing LU decomposition. The eigenvalues (diagonal coefficients) of *U* are sorted in such a way that any zeros are at the end.

This decomposition provides the generic approach to solving systems of linear equations, computing the rank, invertibility, inverse, kernel, and determinant. This LU decomposition is very stable and well tested with large matrices. However there are use cases where the SVD decomposition is inherently more stable and/or flexible. For example, when computing the kernel of a matrix, working with the SVD allows to select the smallest singular values of the matrix, something that the LU decomposition doesn't see.

The data of the LU decomposition can be directly accessed through the methods `matrixLU()`, `permutationP()`, `permutationQ()`.

Computes the LU decomposition of the given matrix.

Parameters: *matrix* the matrix of which to compute the LU decomposition. It is required to be nonzero.

Returns: a reference to `*this` Referenced by `FullPivLU`; `MatrixType` `l::FullPivLU()`.

internal::traits`j` `MatrixType` `l::Scalar` **determinant** ( ) const

Returns the determinant of the matrix of which `*this` is the LU decomposition. It has only linear complexity (that is,  $O(n)$  where  $n$  is the dimension of the square matrix) as the LU decomposition has already been computed.

For fixed-size matrices of size up to 4, `MatrixBase::determinant()` offers optimized paths.

Warning: a determinant can be very big or small, so for matrices of large enough dimension, there is a risk of overflow/underflow.

Index **dimensionOfKernel** ( ) const

Returns the dimension of the kernel of the matrix of which `*this` is the LU decomposition. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivLU``::MatrixType` `::rank()`.

`const internal::image_retvalFullPivLU::image ( const MatrixType & originalMatrix ) const`  
 Returns the image of the matrix, also called its column-space. The columns of the returned matrix will form a basis of the kernel.

Parameters: `originalMatrix` the original matrix, of which `*this` is the LU decomposition. The reason why it is needed to pass it here, is that this allows a large optimization, as otherwise this method would need to reconstruct it from the LU decomposition.

Note: If the image has dimension zero, then the returned matrix is a column-vector filled with zeros. This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`.

Example:

---

```
Matrix3d m;m << 1,1,0, 1,3,2, 0,1,1;
cout << "Here is the matrix m:" << endl << m << endl;
cout << "Notice that the middle column is the sum of the two others, so the "
<< "columns are linearly dependent." << endl;
cout << "Here is a matrix whose columns have the same span but are linearly
      independent:"
<< endl << m.fullPivLu().image(m) << endl;
```

---

Output:

Here is the matrix m:

```
1 1 0
1 3 2
0 1 1
```

Notice that the middle column is the sum of the two others, so the columns are linearly

Here is a matrix whose columns have the same span but are linearly independent:

```
1 1
3 1
1 0
```

`const internal::solve_retvalFullPivLU,typename MatrixType::IdentityReturnType::inverse ( ) const`

Returns the inverse of the matrix of which `*this` is the LU decomposition. Note: If this matrix is not invertible, the returned matrix has undefined coefficients. Use `isInvertible()` to first determine whether this matrix is invertible.

`bool isInjective ( ) const`

Returns true if the matrix of which `*this` is the LU decomposition represents an injective linear map, i.e. has trivial kernel; false otherwise. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivLU``::MatrixType` `::rank()`.

`bool isInvertible ( ) const`

Returns true if the matrix of which `*this` is the LU decomposition is invertible. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold

value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivLU`; `MatrixType` `isInjective()`.

**bool isSurjective () const**

Returns true if the matrix of which `*this` is the LU decomposition represents a surjective linear map; false otherwise. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivLU`; `MatrixType` `rank()`.

**const internal::kernel\_retval`FullPivLU`::kernel () const**

Returns the kernel of the matrix, also called its null-space. The columns of the returned matrix will form a basis of the kernel. Note: If the kernel has dimension zero, then the returned matrix is a column-vector filled with zeros. This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`.

Example:

---

```
MatrixXf m = MatrixXf::Random(3,5);
cout << "Here is the matrix m:" << endl << m << endl;
MatrixXf ker = m.fullPivLu().kernel();
cout << "Here is a matrix whose columns form a basis of the kernel of m:"
<< endl << ker << endl; cout << "By definition of the kernel, m*ker is zero:"
<< endl << m*ker << endl;
```

---

Output:

Here is the matrix m:

```
0.68  0.597  -0.33  0.108  -0.27
-0.211  0.823  0.536 -0.0452  0.0268
0.566  -0.605  -0.444  0.258   0.904
```

Here is a matrix whose columns form a basis of the kernel of m:

```
-0.219  0.763
0.00335 -0.447
0      1
1      0
-0.145 -0.285
```

By definition of the kernel, m\*ker is zero:

```
-1.12e-08  1.49e-08
-1.4e-09 -4.05e-08
1.49e-08 -2.98e-08
```

**const MatrixType& matrixLU () const**

Returns the LU decomposition matrix: the upper-triangular part is U, the unit-lower-triangular part is L (at least for square matrices; in the non-square case, special care is needed, see the documentation of class `FullPivLU`).

**RealScalar maxPivot () const**

Returns the absolute value of the biggest pivot, i.e. the biggest diagonal coefficient of U.

**Index nonzeroPivots () const**

Returns the number of nonzero pivots in the LU decomposition. Here nonzero is meant in the exact sense, not in a fuzzy sense. So that notion isn't really intrinsically interesting, but it is still useful when implementing algorithms.

const PermutationPType& **permutationP** ( ) const inline

Returns the permutation matrix P

const PermutationQType& **permutationQ** ( ) const

Returns the permutation matrix Q

Index **rank** ( ) const

Returns the rank of the matrix of which *\*this* is the LU decomposition. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivLU`; `MatrixType` *i*::`threshold()`.

`MatrixType reconstructedMatrix` ( ) const

Returns the matrix represented by the decomposition, i.e., it returns the product:  $P^{-1}LUQ^{-1}$ . This function is provided for debug purpose.

`FullPivLU& setThreshold` ( const `RealScalar` & threshold)

Allows to prescribe a threshold to be used by certain methods, such as `rank()`, who need to determine when pivots are to be considered nonzero. This is not used for the LU decomposition itself. When it needs to get the threshold value, Eigen calls `threshold()`. By default, this uses a formula to automatically determine a reasonable threshold. Once you have called the present method `setThreshold(const RealScalar&)`, your value is used instead.

Parameters: threshold The new value to use as the threshold.

A pivot will be considered nonzero if its absolute value is strictly greater than where `maxpivot` is the biggest pivot. If you want to come back to the default behavior, call `setThreshold(Default_t)` References `FullPivLU`; `MatrixType` *i*::`threshold()`.

`FullPivLU& setThreshold` ( `Default_t` )

Allows to come back to the default behavior, letting Eigen use its default formula for determining the threshold. You should pass the special object `Eigen::Default` as parameter here. `lu.setThreshold(Eigen::Default);` See the documentation of `setThreshold(const RealScalar&)`.

const internal::solve\_retval`FullPivLU`, `Rhs` *i* **solve** ( const `MatrixBase` *i*; `Rhs` *i* & `b` ) const

Returns a solution *x* to the equation  $Ax = b$ , where *A* is the matrix of which *\*this* is the LU decomposition. Parameters: *b* the right-hand-side of the equation to solve. Can be a vector or a matrix, the only requirement in order for the equation to make sense is that `b.rows() == A.rows()`, where *A* is the matrix of which *\*this* is the LU decomposition.

Returns a solution. This method just tries to find as good a solution as possible. If you want to check whether a solution exists or if it is accurate, just call this function to get a result and then compute the error of this result, or use `MatrixBase::isApprox()` directly, for instance like this:

```
bool a\_solution\_exists = (A*result).isApprox(b, precision);
```

This method avoids dividing by zero, so that the non-existence of a solution doesn't by itself mean that you'll get inf or nan values. If there exists more than one solution, this method will arbitrarily choose one. If you need a complete analysis of the space of solutions, take the one solution obtained by this method and add to it elements of the kernel, as determined by `kernel()`.

Example:

---

```
Matrix<float,2,3> m = Matrix<float,2,3>::Random();
Matrix2f y = Matrix2f::Random();
cout << "Here is the matrix m:" << endl << m << endl;
cout << "Here is the matrix y:" << endl << y << endl;
Matrix<float,3,2> x = m.fullPivLu().solve(y);
if((m*x).isApprox(y))
{ cout << "Here is a solution x to the equation mx=y:" << endl << x << endl;}
else cout << "The equation mx=y does not have any solution." << endl;
```

---

Output:

```
Here is the matrix m:
0.68  0.566  0.823
-0.211  0.597 -0.605
Here is the matrix y:
-0.33 -0.444
0.536  0.108
Here is a solution x to the equation mx=y:
0      0
0.291 -0.216
-0.6 -0.391
```

RealScalar **threshold** ( ) const

Returns the threshold that will be used by certain methods such as `rank()`. See the documentation of `setThreshold(const RealScalar&)`.

#### 24.4.2 Linear Solver

---

Function **SolveFullPivLU**(*A* As `mpNum[,]`, *B* As `mpNum[,]`) As `mpNum[]`

---

The function `SolveFullPivLU` returns the solution  $x$  of  $Ax = b$  , based on a LU decomposition with full pivoting.

**Parameters:**

*A*: A square real matrix.

*B*: A real vector or matrix.

---

Function **cplxSolveFullPivLU**(*A* As `mpNum[,]`, *B* As `mpNum[,]`) As `mpNum[]`

---

The function `cplxSolveFullPivLU` returns the solution  $x$  of  $Ax = b$  , based on a LU decomposition with full pivoting.

**Parameters:**

*A*: A square complex matrix.

*B*: A complex vector or matrix.

#### 24.4.3 Matrix Inversion

---

Function **InvertFullPivLU**(*A* As `mpNum[,]`) As `mpNum[]`

---

The function `InvertFullPivLU` returns  $A^{-1}$ , the inverse of  $A$ , based on a LU decomposition with full pivoting.

**Parameter:**

$A$ : A square real matrix.

---

Function **`cplxInvertFullPivLU(A As mpNum[,]) As mpNum[]`**

---

The function `cplxInvertFullPivLU` returns  $A^{-1}$ , the inverse of  $A$ , based on a LU decomposition with full pivoting.

**Parameter:**

$A$ : A square complex matrix.

#### 24.4.4 Determinant

---

Function **`DetFullPivLU(A As mpNum[,]) As mpNum`**

---

The function `DetFullPivLU` returns  $|A|$ , the determinant of  $A$ , based on a LU decomposition with full pivoting.

**Parameter:**

$A$ : A square real matrix.

---

Function **`cplxDetFullPivLU(A As mpNum[,]) As mpNum`**

---

The function `cplxDetFullPivLU` returns  $|A|$ , the determinant of  $A$ , based on a LU decomposition with full pivoting.

**Parameter:**

$A$ : A square complex matrix.

## 24.5 QR Decomposition without Pivoting

### 24.5.1 Decomposition

---

Function **DecompQR**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function **DecompQR** returns the QR decomposition without pivoting of  $A = QR$ .

**Parameters:**

*A*: the square real matrix of which we are computing the *LU* decomposition.

*B*: A vector or matrix of real numbers.

*Output*: A string specifying the output options.

---

Function **cplxDecompQR**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function **cplxDecompQR** returns the QR decomposition without pivoting of  $A = QR$ .

**Parameters:**

*A*: the square complex matrix of which we are computing the *LU* decomposition.

*B*: A vector or complex of real numbers.

*Output*: A string specifying the output options.

This class performs a QR decomposition of a matrix *A* into matrices *Q* and *R* such that

$$A = QR \quad (24.5.1)$$

by using Householder transformations. Here, *Q* a unitary matrix and *R* an upper triangular matrix. The result is stored in a compact way compatible with LAPACK. Note that no pivoting is performed. This is not a rank-revealing decomposition. If you want that feature, use **FullPivHouseholderQR** or **ColPivHouseholderQR** instead.

This Householder QR decomposition is faster, but less numerically stable and less feature-rich than **FullPivHouseholderQR** or **ColPivHouseholderQR**.

Member Function DocumentationMatrixType::RealScalar **absDeterminant** ( ) const

Returns the absolute value of the determinant of the matrix of which *\*this* is the QR decomposition. It has only linear complexity (that is,  $O(n)$  where *n* is the dimension of the square matrix) as the QR decomposition has already been computed. Note: This is only for square matrices. Warning: a determinant can be very big or small, so for matrices of large enough dimension, there is a risk of overflow/underflow. One way to work around that is to use **logAbsDeterminant()** instead. See Also **logAbsDeterminant()**, **MatrixBase::determinant()**

HouseholderQR::MatrixType *i* & **compute** ( const MatrixType & *matrix* )

Performs the QR factorization of the given matrix *matrix*. The result of the factorization is stored into *\*this*, and a reference to *\*this* is returned. See Also: class **HouseholderQR**, **HouseholderQR(const MatrixType&)**

const HCoeffsType& **hCoeffs** ( ) const

Returns a const reference to the vector of Householder coefficients used to represent the factor *Q*. For advanced uses only.

HouseholderSequenceType **householderQ** ( void ) const

This method returns an expression of the unitary matrix *Q* as a sequence of Householder transformations. The returned expression can directly be used to perform matrix products. It can also

be assigned to a dense Matrix object. Here is an example showing how to recover the full or thin matrix  $Q$ , as well as how to perform matrix products using operator\*:

Example:

---

```
MatrixXf A(MatrixXf::Random(5,3)), thinQ(MatrixXf::Identity(5,3)),
Q;A.setRandom();
HouseholderQR<MatrixXf> qr(A);
Q = qr.householderQ();
thinQ = qr.householderQ() * thinQ;
std::cout << "The complete unitary matrix Q is:\n" << Q << "\n\n";
std::cout << "The thin matrix Q is:\n" << thinQ << "\n\n";
```

---

Output:

The complete unitary matrix  $Q$  is:

```
-0.676  0.0793  0.713 -0.0788 -0.147
-0.221 -0.322  -0.37  -0.366 -0.759
-0.353 -0.345  -0.214  0.841 -0.0518
0.582 -0.462   0.555  0.176 -0.329
-0.174 -0.747 -0.00907 -0.348  0.539
```

The thin matrix  $Q$  is:

```
-0.676  0.0793  0.713
-0.221 -0.322  -0.37
-0.353 -0.345  -0.214
0.582 -0.462   0.555
-0.174 -0.747 -0.00907
```

MatrixType::RealScalar **logAbsDeterminant** ( ) const

Returns the natural log of the absolute value of the determinant of the matrix of which \*this is the QR decomposition. It has only linear complexity (that is,  $O(n)$  where  $n$  is the dimension of the square matrix) as the QR decomposition has already been computed. Note: This is only for square matrices. This method is useful to work around the risk of overflow/underflow that's inherent to determinant computation. See Also `absDeterminant()`, `MatrixBase::determinant()`

const MatrixType& **matrixQR** ( ) const

Returns a reference to the matrix where the Householder QR decomposition is stored in a LAPACK-compatible way.

const internal::solve\_retval<HouseholderQR, Rhs> **solve** ( const MatrixBase<Rhs> & b) const  
 This method finds a solution  $x$  to the equation  $Ax = b$ , where  $A$  is the matrix of which \*this is the QR decomposition, if any exists. Parameters:  $b$  the right-hand-side of the equation to solve. Returns a solution. Note: The case where  $b$  is a matrix is not yet implemented. Also, this code is space inefficient. This method just tries to find as good a solution as possible. If you want to check whether a solution exists or if it is accurate, just call this function to get a result and then compute the error of this result, or use `MatrixBase::isApprox()` directly, for instance like this:

```
bool a\_solution\_exists = (A*result).isApprox(b, precision);
```

This method avoids dividing by zero, so that the non-existence of a solution doesn't by itself mean that you'll get inf or nan values. If there exists more than one solution, this method will arbitrarily choose one.

Example:

---

```

typedef Matrix<float,3,3> Matrix3x3;
Matrix3x3 m = Matrix3x3::Random();
Matrix3f y = Matrix3f::Random();
cout << "Here is the matrix m:" << endl << m << endl;
cout << "Here is the matrix y:" << endl << y << endl;
Matrix3f x;
x = m.householderQr().solve(y);
assert(y.isApprox(m*x));
cout << "Here is a solution x to the equation mx=y:" << endl << x << endl;

```

---

Output:

Here is the matrix m:

0.68 0.597 -0.33  
-0.211 0.823 0.536  
0.566 -0.605 -0.444

Here is the matrix y:

0.108 -0.27 0.832  
-0.0452 0.0268 0.271  
0.258 0.904 0.435

Here is a solution x to the equation mx=y:

0.609 2.68 1.67  
-0.231 -1.57 0.0713  
0.51 3.51 1.05

## 24.5.2 Linear Solver

---

Function **SolveQR(A As mpNum[,], B As mpNum[,]) As mpNum[]**

---

The function **SolveQR** returns the solution  $x$  of  $Ax = b$  , based on a QR decomposition without pivoting.

**Parameters:**

*A*: A square real matrix.

*B*: A real vector or matrix.

---

Function **cplxSolveQR(A As mpNum[,], B As mpNum[,]) As mpNum[]**

---

The function **cplxSolveQR** returns the solution  $x$  of  $Ax = b$  , based on a QR decomposition without pivoting.

**Parameters:**

*A*: A square complex matrix.

*B*: A complex vector or matrix.

## 24.5.3 Matrix Inversion

---

Function **InvertQR(A As mpNum[,]) As mpNum[]**

---

The function **InvertQR** returns  $A^{-1}$ , the inverse of  $A$ , based on a QR decomposition without pivoting.

**Parameter:**

$A$ : A square real matrix.

---

**Function `cplxInvertQR(A As mpNum[,]) As mpNum[]`**

---

The function `cplxInvertQR` returns  $A^{-1}$ , the inverse of  $A$ , based on a QR decomposition without pivoting.

**Parameter:**

$A$ : A square complex matrix.

#### 24.5.4 Determinant

---

**Function `DetQR(A As mpNum[,]) As mpNum`**

---

The function `DetQR` returns  $|A|$ , the determinant of  $A$ , based on a QR decomposition without pivoting.

**Parameter:**

$A$ : A square real matrix.

---

**Function `cplxDetQR(A As mpNum[,]) As mpNum`**

---

The function `cplxDetQR` returns  $|A|$ , the determinant of  $A$ , based on a QR decomposition without pivoting.

**Parameter:**

$A$ : A square complex matrix.

#### 24.5.5 Example

Example:

Example

## 24.6 QR Decomposition with column Pivoting

### 24.6.1 Decomposition

---

Function **DecompColPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function *DecompColPivQR* returns the QR decomposition with column-pivoting of  $A = QR$ .

**Parameters:**

*A*: the square real matrix of which we are computing the *LU* decomposition.

*B*: A vector or matrix of real numbers.

*Output*: A string specifying the output options.

---

Function **cplxDecompColPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function *cplxDecompColPivQR* returns the QR decomposition with column-pivoting of  $A = QR$ .

**Parameters:**

*A*: the square complex matrix of which we are computing the *LU* decomposition.

*B*: A vector or complex of real numbers.

*Output*: A string specifying the output options.

This class performs a rank-revealing QR decomposition of a matrix *A* into matrices *P*, *Q* and *R* such that

$$AP = QR \quad (24.6.1)$$

by using Householder transformations. Here, *P* is a permutation matrix, *Q* a unitary matrix and *R* an upper triangular matrix.

This decomposition performs column pivoting in order to be rank-revealing and improve numerical stability.

It is slower than *HouseholderQR*, and faster than *FullPivHouseholderQR*.

Member Function DocumentationMatrixType::RealScalar **absDeterminant** ( ) const

Returns the absolute value of the determinant of the matrix of which *\*this* is the QR decomposition. It has only linear complexity (that is,  $O(n)$  where *n* is the dimension of the square matrix) as the QR decomposition has already been computed. Note: This is only for square matrices. Warning: a determinant can be very big or small, so for matrices of large enough dimension, there is a risk of overflow/underflow. One way to work around that is to use *logAbsDeterminant()* instead. See Also *logAbsDeterminant()*, *MatrixBase::determinant()*

const PermutationType& **colsPermutation** ( ) const

Returns a const reference to the column permutation matrix

ColPivHouseholderQR<sub>i</sub> MatrixType<sub>i</sub> & **compute** ( const MatrixType & *matrix* )

Performs the QR factorization of the given matrix *matrix*. The result of the factorization is stored into *\*this*, and a reference to *\*this* is returned. See Also class *ColPivHouseholderQR*, *ColPivHouseholderQR*(const MatrixType&)

Index **dimensionOfKernel** ( ) const

Returns the dimension of the kernel of the matrix of which *\*this* is the QR decomposition. Note: This method has to determine which pivots should be considered nonzero. For that, it uses

the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `ColPivHouseholderQR::MatrixType::rank()`.

`const HCoeffsType& hCoeffs() const`

Returns a const reference to the vector of Householder coefficients used to represent the factor `Q`. For advanced uses only.

`ColPivHouseholderQR::MatrixType::HouseholderSequenceType householderQ() const`  
Returns the matrix `Q` as a sequence of householder transformations

`ComputationInfo info() const`

Reports whether the QR factorization was successful. Note: This function always returns `Success`. It is provided for compatibility with other factorization routines.

`const internal::solve_retval::ColPivHouseholderQR, typename MatrixType::IdentityReturnType::inverse() const`

Returns the inverse of the matrix of which `*this` is the QR decomposition. Note: If this matrix is not invertible, the returned matrix has undefined coefficients. Use `isInvertible()` to first determine whether this matrix is invertible.

`bool isInjective() const`

Returns true if the matrix of which `*this` is the QR decomposition represents an injective linear map, i.e. has trivial kernel; false otherwise. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `ColPivHouseholderQR::MatrixType::rank()`.

`bool isInvertible() const`

Returns true if the matrix of which `*this` is the QR decomposition is invertible. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `ColPivHouseholderQR::MatrixType::isInjective()`, and `ColPivHouseholderQR::MatrixType::isSurjective()`.

`bool isSurjective() const`

Returns true if the matrix of which `*this` is the QR decomposition represents a surjective linear map; false otherwise. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `ColPivHouseholderQR::MatrixType::rank()`.

`MatrixType::RealScalar logAbsDeterminant() const`

Returns the natural log of the absolute value of the determinant of the matrix of which `*this` is the QR decomposition. It has only linear complexity (that is,  $O(n)$  where  $n$  is the dimension of the square matrix) as the QR decomposition has already been computed. Note: This is only for square matrices. This method is useful to work around the risk of overflow/underflow that's inherent to determinant computation. See Also `absDeterminant()`, `MatrixBase::determinant()`

`const MatrixType& matrixQR() const`

Returns a reference to the matrix where the Householder QR decomposition is stored

`const MatrixType& matrixR() const`

Returns a reference to the matrix where the result Householder QR is stored Warning: The strict lower part of this matrix contains internal values. Only the upper triangular part should be referenced. To get it, use `matrixR().template triangularView<Upper>()` For rank-deficient matrices, use `matrixR().topLeftCorner(rank(), rank()).template triangularView<Upper>()` `RealScalar`

**maxPivot ( ) const**

Returns the absolute value of the biggest pivot, i.e. the biggest diagonal coefficient of R.

**Index nonzeroPivots ( ) const**

Returns the number of nonzero pivots in the QR decomposition. Here nonzero is meant in the exact sense, not in a fuzzy sense. So that notion isn't really intrinsically interesting, but it is still useful when implementing algorithms. See Also rank() .

**Index rank ( ) const**

Returns the rank of the matrix of which \*this is the QR decomposition. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling setThreshold(const RealScalar&). References ColPivHouseholderQR::MatrixType::threshold().

**ColPivHouseholderQR& setThreshold ( const RealScalar & threshold)**

Allows to prescribe a threshold to be used by certain methods, such as rank(), who need to determine when pivots are to be considered nonzero. This is not used for the QR decomposition itself. When it needs to get the threshold value, Eigen calls threshold(). By default, this uses a formula to automatically determine a reasonable threshold. Once you have called the present method setThreshold(const RealScalar&), your value is used instead. Parameters: threshold The new value to use as the threshold.

A pivot will be considered nonzero if its absolute value is strictly greater than where maxpivot is the biggest pivot. If you want to come back to the default behavior, call setThreshold(Default\_t) References ColPivHouseholderQR::MatrixType::threshold().

**ColPivHouseholderQR& setThreshold ( Default\_t )**

Allows to come back to the default behavior, letting Eigen use its default formula for determining the threshold. You should pass the special object Eigen::Default as parameter here. qr.setThreshold(Eigen::Default); See the documentation of setThreshold(const RealScalar&).

```
const internal::solve_retval<ColPivHouseholderQR, Rhs> solve ( const MatrixBase< Rhs > & b )
const
```

This method finds a solution x to the equation Ax=b, where A is the matrix of which \*this is the QR decomposition, if any exists. Parameters: b the right-hand-side of the equation to solve.

Returns a solution. Note: The case where b is a matrix is not yet implemented. Also, this code is space inefficient. This method just tries to find as good a solution as possible. If you want to check whether a solution exists or if it is accurate, just call this function to get a result and then compute the error of this result, or use MatrixBase::isApprox() directly, for instance like this:

```
bool a\_\_solution\_\_exists = (A*result).isApprox(b, precision);
```

This method avoids dividing by zero, so that the non-existence of a solution doesn't by itself mean that you'll get inf or nan values. If there exists more than one solution, this method will arbitrarily choose one.

Example:

---

```
Matrix3f m = Matrix3f::Random();
Matrix3f y = Matrix3f::Random();
cout << "Here is the matrix m:" << endl << m << endl;
cout << "Here is the matrix y:" << endl << y << endl;Matrix3f x;
x = m.colPivHouseholderQr().solve(y);
assert(y.isApprox(m*x));
```

---

```
cout << "Here is a solution x to the equation mx=y:" << endl << x << endl;
```

---

Output:

Here is the matrix m:

0.68 0.597 -0.33  
-0.211 0.823 0.536  
0.566 -0.605 -0.444

Here is the matrix y:

0.108 -0.27 0.832  
-0.0452 0.0268 0.271  
0.258 0.904 0.435

Here is a solution x to the equation mx=y:

0.609 2.68 1.67  
-0.231 -1.57 0.0713  
0.51 3.51 1.05

RealScalar threshold ( ) const inline

Returns the threshold that will be used by certain methods such as rank(). See the documentation of setThreshold(const RealScalar&). Referenced by ColPivHouseholderQR $\langle$  MatrixType $\rangle$ ::rank(), and ColPivHouseholderQR $\langle$  MatrixType $\rangle$ ::setThreshold().

## 24.6.2 Linear Solver

---

Function **SolveColPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function **SolveColPivQR** returns the solution  $x$  of  $Ax = b$ , based on a QR decomposition with column-pivoting.

**Parameters:**

*A*: A square real matrix.

*B*: A real vector or matrix.

---

Function **cplxSolveColPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function **cplxSolveColPivQR** returns the solution  $x$  of  $Ax = b$ , based on a QR decomposition with column-pivoting.

**Parameters:**

*A*: A square complex matrix.

*B*: A complex vector or matrix.

## 24.6.3 Matrix Inversion

---

Function **InvertColPivQR**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function **InvertColPivQR** returns  $A^{-1}$ , the inverse of  $A$ , based on a QR decomposition with column-pivoting.

**Parameter:**

*A*: A square real matrix.

---

**Function `cplxInvertColPivQR(A As mpNum[,]) As mpNum[]`**

---

The function `cplxInvertColPivQR` returns  $A^{-1}$ , the inverse of  $A$ , based on a QR decomposition with column-pivoting.

**Parameter:**

$A$ : A square complex matrix.

#### 24.6.4 Determinant

---

**Function `DetColPivQR(A As mpNum[,]) As mpNum`**

---

The function `DetColPivQR` returns  $|A|$ , the determinant of  $A$ , based on a QR decomposition with column-pivoting.

**Parameter:**

$A$ : A square real matrix.

---

**Function `cplxDetColPivQR(A As mpNum[,]) As mpNum`**

---

The function `cplxDetColPivQR` returns  $|A|$ , the determinant of  $A$ , based on a QR decomposition with column-pivoting.

**Parameter:**

$A$ : A square complex matrix.

#### 24.6.5 Example

Example:

Example

## 24.7 QR Decomposition with full Pivoting

### 24.7.1 Decomposition

---

Function **DecompFullPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function **DecompFullPivQR** returns the QR decomposition with full pivoting of  $AP = QR$ .

**Parameters:**

*A*: the square real matrix of which we are computing the *LU* decomposition.

*B*: A vector or matrix of real numbers.

*Output*: A string specifying the output options.

---

Function **cplxDecompFullPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,], *Output* As *String*) As *mpNumList*

---

The function **cplxDecompFullPivQR** returns the QR decomposition with full pivoting of  $AP = QR$ .

**Parameters:**

*A*: the square complex matrix of which we are computing the *LU* decomposition.

*B*: A vector or complex of real numbers.

*Output*: A string specifying the output options.

This class performs a rank-revealing QR decomposition of a matrix *A* into matrices *P*, *Q* and *R* such that

$$AP = QR \quad (24.7.1)$$

by using Householder transformations. Here, *P* is a permutation matrix, *Q* a unitary matrix and *R* an upper triangular matrix.

This decomposition performs a very prudent full pivoting in order to be rank-revealing and achieve optimal numerical stability. The trade-off is that it is slower than **HouseholderQR** and **ColPivHouseholderQR**.

Member Function Documentation *MatrixType*::*RealScalar* **absDeterminant** ( ) const

Returns the absolute value of the determinant of the matrix of which *\*this* is the QR decomposition. It has only linear complexity (that is,  $O(n)$  where *n* is the dimension of the square matrix) as the QR decomposition has already been computed. Note: This is only for square matrices. Warning: a determinant can be very big or small, so for matrices of large enough dimension, there is a risk of overflow/underflow. One way to work around that is to use **logAbsDeterminant()** instead. See Also **logAbsDeterminant()**, *MatrixBase*::*determinant()*

const *PermutationType*& **colsPermutation** ( ) const

Returns a const reference to the column permutation matrix

*FullPivHouseholderQR*::*MatrixType* *j* & **compute** ( const *MatrixType* & *matrix* )

Performs the QR factorization of the given matrix *matrix*. The result of the factorization is stored into *\*this*, and a reference to *\*this* is returned. See Also class **FullPivHouseholderQR**, **FullPivHouseholderQR**(const *MatrixType*&)

Index **dimensionOfKernel** ( ) const

Returns the dimension of the kernel of the matrix of which *\*this* is the QR decomposition. Note: This method has to determine which pivots should be considered nonzero. For that, it uses

the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivHouseholderQR::MatrixType::rank()`.

**const HCoeffsType& hCoeffs () const**

Returns a const reference to the vector of Householder coefficients used to represent the factor  $Q$ . For advanced uses only.

**const internal::solve\_retval<FullPivHouseholderQR, typename MatrixType::IdentityReturnType> inverse () const**

Returns the inverse of the matrix of which `*this` is the QR decomposition. Note: If this matrix is not invertible, the returned matrix has undefined coefficients. Use `isInvertible()` to first determine whether this matrix is invertible.

**bool isInjective () const**

Returns true if the matrix of which `*this` is the QR decomposition represents an injective linear map, i.e. has trivial kernel; false otherwise. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivHouseholderQR::MatrixType::rank()`.

**bool isInvertible () const**

Returns true if the matrix of which `*this` is the QR decomposition is invertible. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivHouseholderQR::MatrixType::isInjective()`, and `FullPivHouseholderQR::MatrixType::isSurjective()`.

**bool isSurjective () const**

Returns true if the matrix of which `*this` is the QR decomposition represents a surjective linear map; false otherwise. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivHouseholderQR::MatrixType::rank()`. Referenced by `FullPivHouseholderQR::MatrixType::isInvertible()`.

**MatrixType::RealScalar logAbsDeterminant () const**

Returns the natural log of the absolute value of the determinant of the matrix of which `*this` is the QR decomposition. It has only linear complexity (that is,  $O(n)$  where  $n$  is the dimension of the square matrix) as the QR decomposition has already been computed. Note: This is only for square matrices. This method is useful to work around the risk of overflow/underflow that's inherent to determinant computation. See Also `absDeterminant()`, `MatrixBase::determinant()`

**FullPivHouseholderQR::MatrixType::MatrixQReturnType matrixQ ( void ) const**

Returns Expression object representing the matrix  $Q$

**const MatrixType& matrixQR () const**

Returns a reference to the matrix where the Householder QR decomposition is stored

**RealScalar maxPivot () const**

Returns the absolute value of the biggest pivot, i.e. the biggest diagonal coefficient of  $U$ .

**Index nonzeroPivots () const**

Returns the number of nonzero pivots in the QR decomposition. Here nonzero is meant in the exact sense, not in a fuzzy sense. So that notion isn't really intrinsically interesting, but it is still useful when implementing algorithms. See Also `rank()`.

Index **rank ( ) const**

Returns the rank of the matrix of which *\*this* is the QR decomposition. Note: This method has to determine which pivots should be considered nonzero. For that, it uses the threshold value that you can control by calling `setThreshold(const RealScalar&)`. References `FullPivHouseholderQR::MatrixType::threshold()`.

`const IntDiagSizeVectorType& rowsTranspositions ( ) const`

Returns a const reference to the vector of indices representing the rows transpositions

`FullPivHouseholderQR& setThreshold ( const RealScalar & threshold)`

Allows to prescribe a threshold to be used by certain methods, such as `rank()`, who need to determine when pivots are to be considered nonzero. This is not used for the QR decomposition itself. When it needs to get the threshold value, Eigen calls `threshold()`. By default, this uses a formula to automatically determine a reasonable threshold. Once you have called the present method `setThreshold(const RealScalar&)`, your value is used instead. Parameters `threshold` The new value to use as the threshold.

A pivot will be considered nonzero if its absolute value is strictly greater than where `maxpivot` is the biggest pivot. If you want to come back to the default behavior, call `setThreshold(Default_t)` References `FullPivHouseholderQR::MatrixType::threshold()`.

`FullPivHouseholderQR& setThreshold ( Default_t )`

Allows to come back to the default behavior, letting Eigen use its default formula for determining the threshold. You should pass the special object `Eigen::Default` as parameter here. `qr.setThreshold(Eigen::Default);` See the documentation of `setThreshold(const RealScalar&)`.

`const internal::solve_retval<FullPivHouseholderQR, Rhs> solve ( const MatrixBase<Rhs> & b ) const`

This method finds a solution `x` to the equation `Ax=b`, where `A` is the matrix of which *\*this* is the QR decomposition. Parameters: `b` the right-hand-side of the equation to solve.

Returns the exact or least-square solution if the rank is greater or equal to the number of columns of `A`, and an arbitrary solution otherwise. Note: The case where `b` is a matrix is not yet implemented. Also, this code is space inefficient. This method just tries to find as good a solution as possible. If you want to check whether a solution exists or if it is accurate, just call this function to get a result and then compute the error of this result, or use `MatrixBase::isApprox()` directly, for instance like this:

```
bool a\_solution\_exists = (A*result).isApprox(b, precision);
```

This method avoids dividing by zero, so that the non-existence of a solution doesn't by itself mean that you'll get `inf` or `nan` values. If there exists more than one solution, this method will arbitrarily choose one.

Example:

---

```
Matrix3f m = Matrix3f::Random();
Matrix3f y = Matrix3f::Random();
cout << "Here is the matrix m:" << endl << m << endl;
cout << "Here is the matrix y:" << endl << y << endl;
Matrix3f x;x = m.fullPivHouseholderQr().solve(y);
assert(y.isApprox(m*x));
cout << "Here is a solution x to the equation mx=y:" << endl << x << endl;
```

---

**Output:**

Here is the matrix m:

0.68 0.597 -0.33  
-0.211 0.823 0.536  
0.566 -0.605 -0.444

Here is the matrix y:

0.108 -0.27 0.832  
-0.0452 0.0268 0.271  
0.258 0.904 0.435

Here is a solution x to the equation mx=y:

0.609 2.68 1.67  
-0.231 -1.57 0.0713  
0.51 3.51 1.05

RealScalar **threshold** ( ) const

Returns the threshold that will be used by certain methods such as rank(). See the documentation of setThreshold(const RealScalar&). Referenced by FullPivHouseholderQR $\langle$  MatrixType  $\rangle$ ::rank(), and FullPivHouseholderQR $\langle$  MatrixType  $\rangle$ ::setThreshold().

## 24.7.2 Linear Solver

---

Function **SolveFullPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function **SolveFullPivQR** returns the solution *x* of  $Ax = b$  , based on a QR decomposition with full pivoting.

**Parameters:**

*A*: A square real matrix.

*B*: A real vector or matrix.

---

Function **cplxSolveFullPivQR**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function **cplxSolveFullPivQR** returns the solution *x* of  $Ax = b$  , based on a QR decomposition with full pivoting.

**Parameters:**

*A*: A square complex matrix.

*B*: A complex vector or matrix.

## 24.7.3 Matrix Inversion

---

Function **InvertFullPivQR**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function **InvertFullPivQR** returns  $A^{-1}$ , the inverse of *A*, based on a QR decomposition with full pivoting.

**Parameter:**

*A*: A square real matrix.

---

Function **cplxInvertFullPivQR**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function `cplxInvertFullPivQR` returns  $A^{-1}$ , the inverse of  $A$ , based on a QR decomposition with full pivoting.

**Parameter:**

$A$ : A square complex matrix.

#### 24.7.4 Determinant

---

Function **DetFullPivQR**( $A$  As `mpNum[,]`) As `mpNum`

---

The function `DetFullPivQR` returns  $|A|$ , the determinant of  $A$ , based on a QR decomposition with full pivoting.

**Parameter:**

$A$ : A square real matrix.

---

Function **cplxDetFullPivQR**( $A$  As `mpNum[,]`) As `mpNum`

---

The function `cplxDetFullPivQR` returns  $|A|$ , the determinant of  $A$ , based on a QR decomposition with full pivoting.

**Parameter:**

$A$ : A square complex matrix.

#### 24.7.5 Example

Example:

Example

## 24.8 Singular Value Decomposition

Two-sided Jacobi SVD decomposition of a rectangular matrix.

Parameters

MatrixType the type of the matrix of which we are computing the SVD decomposition

QRPreconditioner this optional parameter allows to specify the type of QR decomposition that will be used internally for the R-SVD step for non-square matrices. See discussion of possible values below.

SVD decomposition consists in decomposing any  $n$ -by- $p$  matrix  $A$  as a product

$$A = USV^* \quad (24.8.1)$$

where  $U$  is a  $n$ -by- $n$  unitary,  $V$  is a  $p$ -by- $p$  unitary, and  $S$  is a  $n$ -by- $p$  real positive matrix which is zero outside of its main diagonal; the diagonal entries of  $S$  are known as the singular values of  $A$  and the columns of  $U$  and  $V$  are known as the left and right singular vectors of  $A$  respectively. Singular values are always sorted in decreasing order.

This JacobiSVD decomposition computes only the singular values by default. If you want  $U$  or  $V$ , you need to ask for them explicitly.

You can ask for only thin  $U$  or  $V$  to be computed, meaning the following. In case of a rectangular  $n$ -by- $p$  matrix, letting  $m$  be the smaller value among  $n$  and  $p$ , there are only  $m$  singular vectors; the remaining columns of  $U$  and  $V$  do not correspond to actual singular vectors. Asking for thin  $U$  or  $V$  means asking for only their  $m$  first columns to be formed. So  $U$  is then a  $n$ -by- $m$  matrix, and  $V$  is then a  $p$ -by- $m$  matrix. Notice that thin  $U$  and  $V$  are all you need for (least squares) solving.

This JacobiSVD class is a two-sided Jacobi R-SVD decomposition, ensuring optimal reliability and accuracy. The downside is that it's slower than bidiagonalizing SVD algorithms for large square matrices; however its complexity is still where  $n$  is the smaller dimension and  $p$  is the greater dimension, meaning that it is still of the same order of complexity as the faster bidiagonalizing R-SVD algorithms. In particular, like any R-SVD, it takes advantage of non-squareness in that its complexity is only linear in the greater dimension.

If the input matrix has inf or nan coefficients, the result of the computation is undefined, but the computation is guaranteed to terminate in finite (and reasonable) time.

The possible values for QRPreconditioner are:

- ColPivHouseholderQRPreconditioner is the default. In practice it's very safe. It uses column-pivoting QR.
- FullPivHouseholderQRPreconditioner, is the safest and slowest. It uses full-pivoting QR. Contrary to other QRs, it doesn't allow computing thin unitaries.
- HouseholderQRPreconditioner is the fastest, and less safe and accurate than the pivoting variants. It uses non-pivoting QR. This is very similar in safety and accuracy to the bidiagonalization process used by bidiagonalizing SVD algorithms (since bidiagonalization is inherently non-pivoting). However the resulting SVD is still more reliable than bidiagonalizing SVDs because the Jacobi-based iterative process is more reliable than the optimized bidiagonal SVD iterations.
- NoQRPreconditioner allows not to use a QR preconditioner at all. This is useful if you know that you will only be computing JacobiSVD decompositions of square matrices. Non-square matrices require a QR preconditioner. Using this option will result in faster compilation and smaller executable code. It won't significantly speed up computation, since JacobiSVD is always checking if QR preconditioning is needed before applying it anyway.

### 24.8.1 Decomposition

---

Function **DecompJacobiSVD**(**A** As *mpNum*[,], **B** As *mpNum*[,], **computationOptions** As *Integer*, **Output** As *String*) As *mpNumList*

---

The function **DecompJacobiSVD** returns the Cholesky decomposition  $A = LL^T = U^*U$  of a matrix.

**Parameters:**

**A:** the real matrix of which we are computing the  $LL^T$  Cholesky decomposition.

**B:** A vector or matrix of real numbers.

**computationOptions:** An optional parameter allowing to specify if you want full or thin U or V unitaries to be computed.

**Output:** A string specifying the output options.

---

Function **cplxDecompJacobiSVD**(**A** As *mpNum*[,], **B** As *mpNum*[,], **computationOptions** As *Integer*, **Output** As *String*) As *mpNumList*

---

The function **cplxDecompJacobiSVD** returns the Cholesky decomposition  $A = LL^T = U^*U$  of a matrix.

**Parameters:**

**A:** the complex matrix of which we are computing the  $LL^T$  Cholesky decomposition.

**B:** A vector or complex of real numbers.

**computationOptions:** An optional parameter allowing to specify if you want full or thin U or V unitaries to be computed.

**Output:** A string specifying the output options.

Member Function DocumentationJacobiSVD`MatrixType, QRPreconditioner & compute (const MatrixType & matrix, unsigned int computationOptions )`

Method performing the decomposition of given matrix using custom options. Parameters: matrix the matrix to decompose

computationOptions: optional parameter allowing to specify if you want full or thin U or V unitaries to be computed. By default, none is computed. This is a bit-field, the possible bits are ComputeFullU, ComputeThinU, ComputeFullV, ComputeThinV.

Thin unitaries are only available if your matrix type has a Dynamic number of columns (for example MatrixXf). They also are not available with the (non-default) FullPivHouseholderQR preconditioner. References JacobiRotation`Scalar &::transpose()`.

JacobiSVD`& compute ( const MatrixType & matrix)`

Method performing the decomposition of given matrix using current options. Parameters: matrix the matrix to decompose

This method uses the current computationOptions, as already passed to the constructor or to compute`(const MatrixType&, unsigned int)`.

bool `computeU ( ) const`

Returns true if U (full or thin) is asked for in this SVD decomposition

bool `computeV ( ) const`

Returns true if V (full or thin) is asked for in this SVD decomposition

const `MatrixUType& matrixU ( ) const`

Returns the U matrix. For the SVD decomposition of a n-by-p matrix, letting m be the minimum of n and p, the U matrix is n-by-n if you asked for ComputeFullU, and is n-by-m if you asked for ComputeThinU.

The m first columns of U are the left singular vectors of the matrix being decomposed. This method asserts that you asked for U to be computed.

const MatrixVType& **matrixV** ( ) const

Returns the V matrix. For the SVD decomposition of a n-by-p matrix, letting m be the minimum of n and p, the V matrix is p-by-p if you asked for ComputeFullV, and is p-by-m if you asked for ComputeThinV. The m first columns of V are the right singular vectors of the matrix being decomposed. This method asserts that you asked for V to be computed.

Index **nonzeroSingularValues** ( ) const

Returns the number of singular values that are not exactly 0

const SingularValuesType& **singularValues** ( ) const

Returns the vector of singular values. For the SVD decomposition of a n-by-p matrix, letting m be the minimum of n and p, the returned vector has size m. Singular values are always sorted in decreasing order.

const internal::solve\_retval<JacobiSVD, Rhs> **solve** ( const MatrixBase<Rhs> & b) const

Returns a (least squares) solution of using the current SVD decomposition of A. Parameters: b the right-hand-side of the equation to solve.

Note: Solving requires both U and V to be computed. Thin U and V are enough, there is no need for full U or V. SVD solving is implicitly least-squares. Thus, this method serves both purposes of exact solving and least-squares solving. In other words, the returned solution is guaranteed to minimize the Euclidean norm  $\|Ax - b\|$ .

## 24.8.2 Linear Solver

---

Function **SolveJacobiSVD**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function *SolveJacobiSVD* returns the solution *x* of  $Ax = b$  , based on a singular value decomposition.

**Parameters:**

*A*: A symmetric positive definite real matrix.

*B*: A real vector or matrix.

---

Function **cplxSolveJacobiSVD**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*[]

---

The function *cplxSolveJacobiSVD* returns the solution *x* of  $Ax = b$  , based on a singular value decomposition.

**Parameters:**

*A*: A symmetric positive definite complex matrix.

*B*: A complex vector or matrix.

## 24.8.3 Matrix Inversion

---

Function **InvertJacobiSVD**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function `InvertJacobiSVD` returns  $A^{-1}$ , the inverse of  $A$ , based on a singular value decomposition.

**Parameter:**

$A$ : A square real matrix.

---

Function **`cplxInvertJacobiSVD(A As mpNum[,]) As mpNum[]`**

---

The function `cplxInvertJacobiSVD` returns  $A^{-1}$ , the inverse of  $A$ , based on a singular value decomposition.

**Parameter:**

$A$ : A square complex matrix.

#### 24.8.4 Determinant

---

Function **`DetJacobiSVD(A As mpNum[,]) As mpNum`**

---

The function `DetJacobiSVD` returns  $|A|$ , the determinant of  $A$ , based on a singular value decomposition.

**Parameter:**

$A$ : A square real matrix.

---

Function **`cplxDetJacobiSVD(A As mpNum[,]) As mpNum`**

---

The function `cplxDetJacobiSVD` returns  $|A|$ , the determinant of  $A$ , based on a singular value decomposition.

**Parameter:**

$A$ : A square complex matrix.

#### 24.8.5 Example

Example:

Here's an example demonstrating basic usage:

---

```
MatrixXf m = MatrixXf::Random(3,2);
cout << "Here is the matrix m:" << endl << m << endl;
JacobiSVD<MatrixXf> svd(m, ComputeThinU | ComputeThinV);
cout << "Its singular values are:" << endl << svd.singularValues() << endl;
cout << "Its left singular vectors are the columns of the thin U matrix:" << endl <<
    svd.matrixU() << endl;
cout << "Its right singular vectors are the columns of the thin V matrix:" << endl <<
    svd.matrixV() << endl;
Vector3f rhs(1, 0, 0);
cout << "Now consider this rhs vector:" << endl << rhs << endl;
cout << "A least-squares solution of m*x = rhs is:" << endl << svd.solve(rhs) << endl;
```

---

Output:

Here is the matrix m:

0.68 0.597  
-0.211 0.823  
0.566 -0.605

Its singular values are:

1.19  
0.899

Its left singular vectors are the columns of the thin U matrix:

0.388 0.866  
0.712 -0.0634  
-0.586 0.496

Its right singular vectors are the columns of the thin V matrix:

-0.183 0.983  
0.983 0.183

Now consider this rhs vector:

1  
0  
0

A least-squares solution of  $m*x = rhs$  is:

0.888  
0.496

## 24.9 Householder Transformations

Reference

Detailed Description

This module provides Householder transformations.

HouseholderSequence

Convenience function for constructing a Householder sequence of Householder reflections acting on subspaces with decreasing size.

Returns A HouseholderSequence constructed from the specified arguments.

HouseholderSequence(OnTheRight)

Convenience function for constructing a Householder sequence.

Returns A HouseholderSequence constructed from the specified arguments.

This function differs from householderSequence() in that the template argument OnTheSide of the constructed HouseholderSequence is set to OnTheRight, instead of the default OnTheLeft.

### 24.9.1 Overview

This class represents a product sequence of Householder reflections where the first Householder reflection acts on the whole space, the second Householder reflection leaves the one-dimensional subspace spanned by the first unit vector invariant, the third Householder reflection leaves the two-dimensional subspace spanned by the first two unit vectors invariant, and so on up to the last reflection which leaves all but one dimensions invariant and acts only on the last dimension. Such sequences of Householder reflections are used in several algorithms to zero out certain parts of a matrix.

Indeed, the methods HessenbergDecomposition::matrixQ(), Tridiagonalization::matrixQ(), HouseholderQR::householderQ(), and ColPivHouseholderQR::householderQ() all return a HouseholderSequence.

More precisely, the Householder sequence represents an  $n \times n$  matrix  $H$  of the form  $H = \prod_{i=0}^{n-1} H_i$  where the  $i$ -th Householder reflection is  $H_i = I - h_i v_i v_i^*$ . The  $i$ -th Householder coefficient  $H_i$  is a scalar and the  $i$ -th Householder vector  $v_i$  is a vector of the form

$$v_i = \underbrace{[0, \dots, 0]}_{i-1 \text{ zeros}}, 1, \underbrace{*, \dots, *}_{n-i \text{ arbitrary entries}}]. \quad (24.9.1)$$

The last  $n - i$  entries of  $v_i$  are called the essential part of the Householder vector.

Typical usages are listed below, where  $H$  is a HouseholderSequence:

---

```
A.applyOnTheRight(H);           // A = A * H
A.applyOnTheLeft(H);           // A = H * A
A.applyOnTheRight(H.adjoint()); // A = A * H^*
A.applyOnTheLeft(H.adjoint()); // A = H^* * A
MatrixXd Q = H;                // conversion to a dense matrix
```

---

In addition to the adjoint, you can also apply the inverse (=adjoint), the transpose, and the conjugate operators.

### 24.9.2 Constructor

Parameters:

[in]  $v$  Matrix containing the essential parts of the Householder vectors  
 [in]  $h$  Vector containing the Householder coefficients

Constructs the Householder sequence with coefficients given by  $h$  and vectors given by  $v$ . The  $i$ -th Householder coefficient  $h_i$  is given by  $h(i)$  and the essential part of the  $i$ -th Householder vector  $v_i$  is given by  $v(k,i)$  with  $k > i$  (the subdiagonal part of the  $i$ -th column). If  $v$  has fewer columns than rows, then the Householder sequence contains as many Householder reflections as there are columns.

Example:

---

```
Matrix3d v = Matrix3d::Random();
cout << "The matrix v is:" << endl;
cout << v << endl;
Vector3d v0(1, v(1,0), v(2,0));
cout << "The first Householder vector is: v\0 = " << v0.transpose() << endl;
Vector3d v1(0, 1, v(2,1));
cout << "The second Householder vector is: v\1 = " << v1.transpose() << endl;
Vector3d v2(0, 0, 1);
cout << "The third Householder vector is: v\2 = " << v2.transpose() << endl;
Vector3d h = Vector3d::Random();
cout << "The Householder coefficients are: h = " << h.transpose() << endl;
Matrix3d H0 = Matrix3d::Identity() - h(0) * v0 * v0.adjoint();
cout << "The first Householder reflection is represented by H\0 = " << endl;
cout << H0 << endl;Matrix3d H1 = Matrix3d::Identity() - h(1) * v1 * v1.adjoint();
cout << "The second Householder reflection is represented by H\1 = " << endl;cout <<
H1 << endl;
Matrix3d H2 = Matrix3d::Identity() - h(2) * v2 * v2.adjoint();
cout << "The third Householder reflection is represented by H\2 = " << endl;
cout << H2 << endl;cout << "Their product is H\0 H\1 H\2 = " << endl;
cout << H0 * H1 * H2 << endl;HouseholderSequence<Matrix3d, Vector3d> hhSeq(v, h);
Matrix3d hhSeqAsMatrix(hhSeq);
cout << "If we construct a HouseholderSequence from v and h" << endl;
cout << "and convert it to a matrix, we get:" << endl;cout << hhSeqAsMatrix << endl;
```

---

Output:

The matrix v is:

```
0.68  0.597 -0.33
-0.211 0.823  0.536
0.566 -0.605 -0.444
```

The first Householder vector is: v\0 = 1 -0.211 0.566

The second Householder vector is: v\1 = 0 1 -0.605

The third Householder vector is: v\2 = 0 0 1

The Householder coefficients are: h = 0.108 -0.0452 0.258

The first Householder reflection is represented by H\0 =

```
0.892 0.0228 -0.0611
0.0228 0.995 0.0129
-0.0611 0.0129 0.965
```

The second Householder reflection is represented by H\1 =

```
1 0 0
0 1.05 -0.0273
0 -0.0273 1.02
```

The third Householder reflection is represented by  $H\_2 =$

```
1      0      0
0      1      0
0      0  0.742
```

Their product is  $H\_0 H\_1 H\_2 =$

```
0.892  0.0255 -0.0466
0.0228  1.04 -0.0105
-0.0611 -0.0129  0.728
```

If we construct a `HouseholderSequence` from `v` and `h` and convert it to a matrix, we get:

```
0.892  0.0255 -0.0466
0.0228  1.04 -0.0105
-0.0611 -0.0129  0.728
```

### 24.9.3 Member Function Documentation

**Index `cols` ( void ) const**

Number of columns of transformation viewed as a matrix.

Returns Number of columns. This equals the dimension of the space that the transformation acts on.

**const `HouseholderSequence`**

Returns a reference to the derived object

**const `EssentialVectorType essentialVector ( Index k ) const`**

Essential part of a Householder vector.

Parameters: [in] `k` Index of Householder reflection

Returns: Vector containing non-trivial entries of  $k$ -th Householder vector

This function returns the essential part of the Householder vector  $v_i$ . This is a vector of length  $n - i$  containing the last  $n - i$  entries of the vector

$$v_i = \left[ \underbrace{0, \dots, 0}_{i-1 \text{ zeros}}, \underbrace{1, \dots, *}_{n-i \text{ arbitrary entries}} \right]. \quad (24.9.2)$$

The index  $i$  equals  $k + \text{shift}()$ , corresponding to the  $k$ -th column of the matrix `v` passed to the constructor.

**`Matrix_type_times_scalar_type`**

Computes the product of a Householder sequence with a matrix.

Parameters: [in] other Matrix being multiplied.

Returns Expression object representing the product. This function computes  $HM$  where  $H$  is the Householder sequence represented by `*this` and  $M$  is the matrix `other`.

**Index `rows ( void ) const`**

Number of rows of transformation viewed as a matrix.

Returns Number of rows

This equals the dimension of the space that the transformation acts on.

`HouseholderSequence& setLength ( Index length)`

Sets the length of the Householder sequence.

Parameters: [in] `length` New value for the length.

By default, the length  $n$  of the Householder sequence  $H = H_0 H_1 \dots H_{n-1}$  is set to the number of columns of the matrix  $v$  passed to the constructor, or the number of rows if that is smaller. After this function is called, the length equals length.

`HouseholderSequence& setShift ( Index shift)`

Sets the shift of the Householder sequence.

Parameters: [in] shift New value for the shift.

By default, a `HouseholderSequence` object represents  $H = H_0 H_1 \dots H_{n-1}$  and the  $i$ -th column of the matrix  $v$  passed to the constructor corresponds to the  $i$ -th Householder reflection. After this function is called, the object represents  $H = H_{\text{shift}} H_{\text{shift} + 1} \dots H_{n-1}$  and the  $i$ -th column of  $v$  corresponds to the  $(\text{shift}+i)$ -th Householder reflection.

`HouseholderSequence& setTrans ( bool trans)`

Sets the transpose flag.

Parameters: [in] trans New value of the transpose flag.

By default, the transpose flag is not set. If the transpose flag is set, then this object represents  $H^T = H_{n-1}^T \dots H_1^T H_0^T$  instead of  $H = H_0 H_1 \dots H_{n-1}$ .

`Index size ( ) const`

Returns the number of coefficients, which is `rows() * cols()`.

# Chapter 25

## Eigensystems, (based on Eigen)

Book reference: [Golub & Van Loan \(1996\)](#)

### 25.1 Symmetric/Hermitian Eigensystems

A matrix  $A$  is selfadjoint if it equals its adjoint. For real matrices, this means that the matrix is symmetric: it equals its transpose. This class computes the eigenvalues and eigenvectors of a selfadjoint matrix. These are the scalars  $\lambda$  and vectors  $v$  such that  $Av = \lambda v$ . The eigenvalues of a selfadjoint matrix are always real. If  $D$  is a diagonal matrix with the eigenvalues on the diagonal, and  $V$  is a matrix with the eigenvectors as its columns, then  $A = VDV^{-1}$  (for selfadjoint matrices, the matrix  $V$  is always invertible). This is called the eigendecomposition.

The algorithm exploits the fact that the matrix is selfadjoint, making it faster and more accurate than the general purpose eigenvalue algorithms implemented in EigenSolver and ComplexEigenSolver.

Only the lower triangular part of the input matrix is referenced.

Call the function `compute()` to compute the eigenvalues and eigenvectors of a given matrix. Alternatively, you can use the `SelfAdjointEigenSolver(const MatrixType, int)` constructor which computes the eigenvalues and eigenvectors at construction time. Once the eigenvalue and eigenvectors are computed, they can be retrieved with the `eigenvalues()` and `eigenvectors()` functions. The documentation for `SelfAdjointEigenSolver(const MatrixType, int)` contains an example of the typical use of this class.

To solve the generalized eigenvalue problem  $Av = \lambda Bv$  and the likes, see the class `GeneralizedSelfAdjointEigenSolver`.

#### 25.1.1 Real Symmetric Matrices

---

##### Function `EigenSymm(A As mpNum[,]) As mpNum`

---

The function `EigenSymm` returns the eigenvalues of a real symmetric matrix.

##### Parameter:

$A$ : the real matrix of which we are computing the eigenvalues.

---

##### Function `EigenSymmv(A As mpNum[,]) As mpNum`

---

The function `EigenSymmv` returns the eigenvalues and eigenvectors of a real symmetric matrix.

**Parameter:**

**A:** the real matrix of which we are computing the eigenvalues.

Member Function Documentation `SelfAdjointEigenSolver::MatrixType & compute ( const MatrixType & matrix, int options = ComputeEigenvectors )`

Computes eigendecomposition of given matrix. Parameters: [in] matrix Selfadjoint matrix whose eigendecomposition is to be computed. Only the lower triangular part of the matrix is referenced. [in] options Can be ComputeEigenvectors (default) or EigenvaluesOnly.

Returns Reference to `*this`.

This function computes the eigenvalues of matrix. The `eigenvalues()` function can be used to retrieve them. If options equals `ComputeEigenvectors`, then the eigenvectors are also computed and can be retrieved by calling `eigenvectors()`.

This implementation uses a symmetric QR algorithm. The matrix is first reduced to tridiagonal form using the Tridiagonalization class. The tridiagonal matrix is then brought to diagonal form with implicit symmetric QR steps with Wilkinson shift. Details can be found in Section 8.3 of [Golub & Van Loan \(1996\)](#).

The cost of the computation is about  $9n^3$  if the eigenvectors are required and  $4n^3/3$  if they are not required.

This method reuses the memory in the `SelfAdjointEigenSolver` object that was allocated when the object was constructed, if the size of the matrix does not change.

Example:

---

```
SelfAdjointEigenSolver<MatrixXf> es(4);
MatrixXf X = MatrixXf::Random(4,4);
MatrixXf A = X + X.transpose();
es.compute(A);
cout << "The eigenvalues of A are: " << es.eigenvalues().transpose() << endl;
es.compute(A + MatrixXf::Identity(4,4)); // re-use es to compute eigenvalues of A+I
cout << "The eigenvalues of A+I are: " << es.eigenvalues().transpose() << endl;
```

---

**Output:**

```
The eigenvalues of A are: -1.58 -0.473 1.32 2.46
The eigenvalues of A+I are: -0.581 0.527 2.32 3.46
```

See Also `SelfAdjointEigenSolver(const MatrixType&, int)` References `Eigen::ComputeEigenvectors`, `Eigen::NoConvergence`, and `Eigen::Success`. Referenced by `SelfAdjointEigenSolver::MatrixType &::SelfAdjointEigenSolver()`.

`SelfAdjointEigenSolver::MatrixType & computeDirect ( const MatrixType & matrix, int options = ComputeEigenvectors )`

Computes eigendecomposition of given matrix using a direct algorithm. This is a variant of `compute(const MatrixType&, int options)` which directly solves the underlying polynomial equation. Currently only 3x3 matrices for which the sizes are known at compile time are supported (e.g., `Matrix3d`). This method is usually significantly faster than the QR algorithm but it might also be less accurate. It is also worth noting that for 3x3 matrices it involves trigonometric operations which are not necessarily available for all scalar types. See Also `compute(const MatrixType&, int options)`

`const RealVectorType& eigenvalues ( ) const`

Returns the eigenvalues of given matrix. Returns A const reference to the column vector containing the eigenvalues. Precondition: The eigenvalues have been computed before. The eigenvalues

are repeated according to their algebraic multiplicity, so there are as many eigenvalues as rows in the matrix. The eigenvalues are sorted in increasing order.

Example:

---

```
MatrixXd ones = MatrixXd::Ones(3,3);
SelfAdjointEigenSolver<MatrixXd> es(ones);
cout << "The eigenvalues of the 3x3 matrix of ones are:"
<< endl << es.eigenvalues() << endl;
```

---

Output:

```
The eigenvalues of the 3x3 matrix of ones are:
-3.09e-16
0
3
```

See Also `eigenvectors()`, `MatrixBase::eigenvalues()`

`const MatrixType& eigenvectors ( ) const`

Returns the eigenvectors of given matrix. Returns A const reference to the matrix whose columns are the eigenvectors. Precondition: The eigenvectors have been computed before.

Column  $k$  of the returned matrix is an eigenvector corresponding to eigenvalue number  $k$  as returned by `eigenvalues()`. The eigenvectors are normalized to have (Euclidean) norm equal to one. If this object was used to solve the eigenproblem for the selfadjoint matrix  $A$ , then the matrix returned by this function is the matrix  $V$  in the eigendecomposition  $A = VDV^{-1}$ .

Example:

---

```
MatrixXd ones = MatrixXd::Ones(3,3);
SelfAdjointEigenSolver<MatrixXd> es(ones);
cout << "The first eigenvector of the 3x3 matrix of ones is:"
<< endl << es.eigenvectors().col(1) << endl;
```

---

Output:

```
The first eigenvector of the 3x3 matrix of ones is:
0
-0.707
0.707
```

See Also `eigenvalues()`

`ComputationInfo info ( ) const`

Reports whether previous computation was successful. Returns: Success if computation was successful, NoConvergence otherwise.

`MatrixType operatorInverseSqrt ( ) const`

---

### Function **MatSymmInverseSqrt**(*A* As *mpNum*[,]) As *mpNum*

---

The function `MatSymmInverseSqrt` returns the inverse matrix square root of a real symmetric matrix.

**Parameter:**

*A*: the real matrix of which we are computing the eigenvalues.

Computes the inverse square root of the matrix. Returns the inverse positive-definite square root of the matrix Precondition: The eigenvalues and eigenvectors of a positive-definite matrix have been computed before. This function uses the eigendecomposition  $A = VDV^{-1}$  to compute the inverse square root as  $VD^{-1/2}V^{-1}$ . This is cheaper than first computing the square root with `operatorSqrt()` and then its inverse with `MatrixBase::inverse()`.

Example:

---

```
MatrixXd X = MatrixXd::Random(4,4);
MatrixXd A = X * X.transpose();
cout << "Here is a random positive-definite matrix, A:" << endl << A << endl << endl;
SelfAdjointEigenSolver<MatrixXd> es(A);
cout << "The inverse square root of A is: " << endl;
cout << es.operatorInverseSqrt() << endl;
cout << "We can also compute it with operatorSqrt() and inverse(). That yields: " <<
     endl;
cout << es.operatorSqrt().inverse() << endl;
```

---

Output:

Here is a random positive-definite matrix, A:  
1.41 -0.697 -0.111 0.508  
-0.697 0.423 0.0991 -0.4  
-0.111 0.0991 1.25 0.902  
0.508 -0.4 0.902 1.4

The inverse square root of A is:

1.88 2.78 -0.546 0.605  
2.78 8.61 -2.3 2.74  
-0.546 -2.3 1.92 -1.36  
0.605 2.74 -1.36 2.18

We can also compute it with `operatorSqrt()` and `inverse()`. That yields:

1.88 2.78 -0.546 0.605  
2.78 8.61 -2.3 2.74  
-0.546 -2.3 1.92 -1.36  
0.605 2.74 -1.36 2.18

See Also `operatorSqrt()`, `MatrixBase::inverse()`, `MatrixFunctions` Module

---

MatrixType **operatorSqrt** ( ) const

---

Function **MatSymmSqrt**(*A* As *mpNum*[,]) As *mpNum*

---

The function `MatSymmSqrt` returns the matrix square root of a real symmetric matrix.

**Parameter:**

*A*: the real matrix of which we are computing the eigenvalues.

Computes the positive-definite square root of the matrix. Returns the positive-definite square root of the matrix Precondition: The eigenvalues and eigenvectors of a positive-definite matrix have been computed before. The square root of a positive-definite matrix *A* is the positive-definite matrix whose square equals *A*. This function uses the eigendecomposition  $A = VDV^{-1}$  to compute the square root as  $A^{1/2} = VD^{1/2}V^{-1}$ .

Example:

---

```
MatrixXd X = MatrixXd::Random(4,4);
MatrixXd A = X * X.transpose();
cout << "Here is a random positive-definite matrix, A:" << endl << A << endl << endl;
SelfAdjointEigenSolver<MatrixXd> es(A);
MatrixXd sqrtA = es.operatorSqrt();
cout << "The square root of A is: " << endl << sqrtA << endl;
cout << "If we square this, we get: " << endl << sqrtA*sqrtA << endl;
```

---

Output:

Here is a random positive-definite matrix, A:

```
1.41 -0.697 -0.111 0.508
-0.697 0.423 0.0991 -0.4
-0.111 0.0991 1.25 0.902
0.508 -0.4 0.902 1.4
```

The square root of A is:

```
1.09 -0.432 -0.0685 0.2
-0.432 0.379 0.141 -0.269
-0.0685 0.141 1 0.468
0.2 -0.269 0.468 1.04
If we square this, we get:
1.41 -0.697 -0.111 0.508
-0.697 0.423 0.0991 -0.4
-0.111 0.0991 1.25 0.902
0.508 -0.4 0.902 1.4
```

See Also: operatorInverseSqrt(), MatrixFunctions Module

Member Data Documentation const int **m\_maxIterations**

Maximum number of iterations. The algorithm terminates if it does not converge within m\_maxIterations \* n iterations, where n denotes the size of the matrix. This value is currently set to 30 (copied from LAPACK).

### 25.1.2 Complex Hermitian Matrices

---

Function **cplxEigenHerm(A As mpNum[,]) As mpNum**

---

The function cplxEigenHerm returns the eigenvalues of a complex hermitian matrix.

**Parameter:**

A: the complex hermitian matrix of which we are computing the eigenvalues.

---

Function **cplxEigenHermv(A As mpNum[,]) As mpNum**

---

The function cplxEigenHermv returns the eigenvalues and eigenvectors of a complex hermitian matrix.

**Parameter:**

A: the complex hermitian matrix of which we are computing the eigenvalues.

### 25.1.2.1 Example

Example:

---

```
MatrixXd X = MatrixXd::Random(5,5);
MatrixXd A = X + X.transpose();
cout << "Here is a random symmetric 5x5 matrix, A:" << endl << A << endl << endl;
SelfAdjointEigenSolver<MatrixXd> es(A);
cout << "The eigenvalues of A are:" << endl << es.eigenvalues() << endl;
cout << "The matrix of eigenvectors, V, is:" << endl << es.eigenvectors() << endl <<
    endl;
double lambda = es.eigenvalues()[0];
cout << "Consider the first eigenvalue, lambda = " << lambda << endl;
VectorXd v = es.eigenvectors().col(0);
cout << "If v is the corresponding eigenvector, then lambda * v = " << endl << lambda
    * v << endl;
cout << "... and A * v = " << endl << A * v << endl << endl;
MatrixXd D = es.eigenvalues().asDiagonal();
MatrixXd V = es.eigenvectors();
cout << "Finally, V * D * V^(-1) = " << endl << V * D * V.inverse() << endl;
```

---

Output:

Here is a random symmetric 5x5 matrix, A:

```
1.36 -0.816 0.521 1.43 -0.144
-0.816 -0.659 0.794 -0.173 -0.406
0.521 0.794 -0.541 0.461 0.179
1.43 -0.173 0.461 -1.43 0.822
-0.144 -0.406 0.179 0.822 -1.37
```

The eigenvalues of A are:

```
-2.65
-1.77
-0.745
0.227
2.29
```

The matrix of eigenvectors, V, is:

```
0.326 -0.0984 -0.347 0.0109 0.874
0.207 -0.642 -0.228 -0.662 -0.232
-0.0495 0.629 0.164 -0.74 0.164
-0.721 -0.397 0.402 -0.115 0.385
0.573 -0.156 0.799 0.0256 0.0858
```

Consider the first eigenvalue, lambda = -2.65

If v is the corresponding eigenvector, then lambda \* v =  
-0.865  
-0.55  
0.131  
1.91  
-1.52  
... and A \* v =

-0.865  
-0.55  
0.131  
1.91  
-1.52

Finally,  $V * D * V^{-1} =$   
1.36 -0.816 0.521 1.43 -0.144  
-0.816 -0.659 0.794 -0.173 -0.406  
0.521 0.794 -0.541 0.461 0.179  
1.43 -0.173 0.461 -1.43 0.822  
-0.144 -0.406 0.179 0.822 -1.37

## 25.2 General (Nonsymmetric) Eigensystems

Computes eigenvalues and eigenvectors of general matrices.

This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the eigendecomposition; this is expected to be an instantiation of the Matrix class template. Currently, only real matrices are supported.

The eigenvalues and eigenvectors of a matrix  $A$  are scalars  $\lambda$  and vectors  $v$  such that  $Av = \lambda v$ . If  $D$  is a diagonal matrix with the eigenvalues on the diagonal, and  $V$  is a matrix with the eigenvectors as its columns, then  $AV = VD$ . The matrix  $V$  is almost always invertible, in which case we have  $A = VDV^{-1}$ . This is called the eigendecomposition. The eigenvalues and eigenvectors of a matrix may be complex, even when the matrix is real. However, we can choose real matrices  $V$  and  $D$  satisfying  $AV = VD$ , just like the eigendecomposition, if the matrix  $D$  is not required to be diagonal, but if it is allowed to have blocks of the form

$$\begin{pmatrix} u & v \\ -v & u \end{pmatrix} \quad (25.2.1)$$

(where  $u$  and  $v$  are real numbers) on the diagonal. These blocks correspond to complex eigenvalue pairs  $u \pm iv$ . We call this variant of the eigendecomposition the pseudo-eigendecomposition.

Call the function compute() to compute the eigenvalues and eigenvectors of a given matrix. Alternatively, you can use the EigenSolver(const MatrixType, bool) constructor which computes the eigenvalues and eigenvectors at construction time. Once the eigenvalue and eigenvectors are computed, they can be retrieved with the eigenvalues() and eigenvectors() functions. The pseudoEigenvalueMatrix() and pseudoEigenvectors() methods allow the construction of the pseudo-eigendecomposition.

The documentation for EigenSolver(const MatrixType, bool) contains an example of the typical use of this class.

See Also

MatrixBase::eigenvalues(), class ComplexEigenSolver, class SelfAdjointEigenSolver

### 25.2.1 Real Nonsymmetric Matrices

---

#### Function **EigenNonsymm**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function EigenNonsymm returns the eigenvalues of a real general (non-symmetric) matrix.

**Parameter:**

*A*: the real general (non-symmetric) matrix of which we are computing the eigenvalues.

---

#### Function **EigenNonsymmv**(*A* As *mpNum*[,]) As *mpNumList*[2]

---

The function EigenNonsymmv returns the eigenvalues and eigenvectors of a real general (non-symmetric) matrix.

**Parameter:**

*A*: the real general (non-symmetric) matrix of which we are computing the eigenvalues.

---

#### Function **PseudoEigenNonsymm**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function `PseudoEigenNonsymm` returns the pseudoeigenvalues of a real general (non-symmetric) matrix.

**Parameter:**

`A`: the real general (non-symmetric) matrix of which we are computing the pseudoeigenvalues.

---

**Function `PseudoEigenNonsymmv(A As mpNum[,]) As mpNumList[2]`**

---

The function `PseudoEigenNonsymmv` returns the pseudoeigenvalues and pseudoeigenvectors of a real general (non-symmetric) matrix.

**Parameter:**

`A`: the real general (non-symmetric) matrix of which we are computing the pseudoeigenvalues and pseudoeigenvectors.

Member Function DocumentationEigenSolver`<MatrixType & compute ( const MatrixType & matrix, bool computeEigenvectors = true)`

Computes eigendecomposition of given matrix. Parameters:

[in] `matrix` Square matrix whose eigendecomposition is to be computed.

[in] `computeEigenvectors` If true, both the eigenvectors and the eigenvalues are computed; if false, only the eigenvalues are computed.

Returns: Reference to `*this`

This function computes the eigenvalues of the real matrix `matrix`. The `eigenvalues()` function can be used to retrieve them. If `computeEigenvectors` is true, then the eigenvectors are also computed and can be retrieved by calling `eigenvectors()`.

The matrix is first reduced to real Schur form using the `RealSchur` class. The Schur decomposition is then used to compute the eigenvalues and eigenvectors. The cost of the computation is dominated by the cost of the Schur decomposition, which is very approximately  $25n^3$  (where  $n$  is the size of the matrix) if `computeEigenvectors` is true, and  $10n^3$  if `computeEigenvectors` is false. This method reuses of the allocated data in the `EigenSolver` object.

Example:

---

```
EigenSolver<MatrixXf> es;
MatrixXf A = MatrixXf::Random(4,4);
es.compute(A, /* computeEigenvectors = */ false);
cout << "The eigenvalues of A are: " << es.eigenvalues().transpose() << endl;
es.compute(A + MatrixXf::Identity(4,4), false); // re-use es to compute eigenvalues
of A+I
cout << "The eigenvalues of A+I are: " << es.eigenvalues().transpose() << endl;
```

---

**Output:**

The eigenvalues of A are: (0.755,0.528) (0.755,-0.528) (-0.323,0.0965) (-0.323,-0.0965)

The eigenvalues of A+I are: (1.75,0.528) (1.75,-0.528) (0.677,0.0965) (0.677,-0.0965)

const EigenvalueType& `eigenvalues () const`

Returns the eigenvalues of given matrix. Returns: A const reference to the column vector containing the eigenvalues. Precondition: Either the constructor `EigenSolver(const MatrixType&,bool)` or the member function `compute(const MatrixType&, bool)` has been called before. The eigenvalues are repeated according to their algebraic multiplicity, so there are as many eigenvalues as rows in the matrix. The eigenvalues are not sorted in any particular order.

Example:

---

```
MatrixXd ones = MatrixXd::Ones(3,3);
EigenSolver<MatrixXd> es(ones, false);
cout << "The eigenvalues of the 3x3 matrix of ones are:" << endl << es.eigenvalues()
    << endl;
```

---

Output:

```
The eigenvalues of the 3x3 matrix of ones are:
(-5.31e-17,0)
(3,0)
(0,0)
```

**EigenSolver***j* MatrixType *j*::EigenvectorsType **eigenvectors** ( ) const

Returns the eigenvectors of given matrix. Returns Matrix whose columns are the (possibly complex) eigenvectors. Precondition: Either the constructor EigenSolver(const MatrixType&,bool) or the member function compute(const MatrixType&, bool) has been called before, and computeEigenvectors was set to true (the default). Column *k* of the returned matrix is an eigenvector corresponding to eigenvalue number *k* as returned by eigenvalues(). The eigenvectors are normalized to have (Euclidean) norm equal to one. The matrix returned by this function is the matrix *V* in the eigendecomposition  $A = VDV^{-1}$ , if it exists.

Example:

---

```
MatrixXd ones = MatrixXd::Ones(3,3);
EigenSolver<MatrixXd> es(ones);
cout << "The first eigenvector of the 3x3 matrix of ones is:"
<< endl << es.eigenvectors().col(1) << endl;
```

---

Output:

```
The first eigenvector of the 3x3 matrix of ones is:
(0.577,0)
(0.577,0)
(0.577,0)
```

MatrixType **pseudoEigenvalueMatrix** ( ) const

Returns the block-diagonal matrix in the pseudo-eigendecomposition. Returns A block-diagonal matrix.

Precondition: Either the constructor EigenSolver(const MatrixType&,bool) or the member function compute(const MatrixType&, bool) has been called before. The matrix *D* returned by this function is real and block-diagonal. The blocks on the diagonal are either 1-by-1 or 2-by-2 blocks of the form  $\begin{pmatrix} u & v \\ -v & u \end{pmatrix}$ . These blocks are not sorted in any particular order. The matrix *D* and the matrix *V* returned by pseudoEigenvectors() satisfy  $AV = VD$ .

See Also **pseudoEigenvectors()** for an example, **eigenvalues()**

const MatrixType& **pseudoEigenvectors** ( ) const

Returns the pseudo-eigenvectors of given matrix. Returns Const reference to matrix whose columns are the pseudo-eigenvectors.

Precondition: Either the constructor EigenSolver(const MatrixType&,bool) or the member function compute(const MatrixType&, bool) has been called before, and computeEigenvectors was set to true (the default). The real matrix  $V$  returned by this function and the block-diagonal matrix  $D$  returned by pseudoEigenvalueMatrix() satisfy  $AV = VD$ .

Example:

---

```
MatrixXd A = MatrixXd::Random(6,6);
cout << "Here is a random 6x6 matrix, A:" << endl << A << endl << endl;
EigenSolver<MatrixXd> es(A);
MatrixXd D = es.pseudoEigenvalueMatrix();
MatrixXd V = es.pseudoEigenvectors();
cout << "The pseudo-eigenvalue matrix D is:" << endl << D << endl;
cout << "The pseudo-eigenvector matrix V is:" << endl << V << endl;
cout << "Finally, V * D * V^(-1) = " << endl << V * D * V.inverse() << endl;
```

---

Output:

Here is a random 6x6 matrix, A:

```
0.68  -0.33  -0.27  -0.717  -0.687  0.0259
-0.211  0.536  0.0268  0.214  -0.198  0.678
0.566  -0.444  0.904  -0.967  -0.74   0.225
0.597  0.108  0.832  -0.514  -0.782  -0.408
0.823 -0.0452  0.271  -0.726  0.998  0.275
-0.605  0.258  0.435  0.608  -0.563  0.0486
```

The pseudo-eigenvalue matrix D is:

```
0.049  1.06  0  0  0  0
-1.06  0.049  0  0  0  0
0  0  0.967  0  0  0
0  0  0  0.353  0  0
0  0  0  0  0.618  0.129
0  0  0  0  -0.129  0.618
```

The pseudo-eigenvector matrix V is:

```
-0.571  -0.888  -0.066  -1.13  17.2  -3.54
0.263  -0.204  -0.869  0.21  9.73  10.7
-0.827  -0.352  0.209  0.0871  -9.75  -4.17
-1.15  0.0535  -0.0857  -0.971  9.36  -4.53
-0.485  0.258  0.436  0.337  -9.74  -2.21
0.206  0.353  -0.426 -0.00873  -0.942  2.98
```

Finally,  $V * D * V^(-1) =$

```
0.68  -0.33  -0.27  -0.717  -0.687  0.0259
-0.211  0.536  0.0268  0.214  -0.198  0.678
0.566  -0.444  0.904  -0.967  -0.74   0.225
0.597  0.108  0.832  -0.514  -0.782  -0.408
0.823 -0.0452  0.271  -0.726  0.998  0.275
-0.605  0.258  0.435  0.608  -0.563  0.0486
```

### 25.2.1.1 Example

Example:

---

```

MatrixXd A = MatrixXd::Random(6,6);
cout << "Here is a random 6x6 matrix, A:" << endl << A << endl << endl;
EigenSolver<MatrixXd> es(A);
cout << "The eigenvalues of A are:" << endl << es.eigenvalues() << endl;
cout << "The matrix of eigenvectors, V, is:" << endl << es.eigenvectors() << endl <<
    endl;
complex<double> lambda = es.eigenvalues()[0];
cout << "Consider the first eigenvalue, lambda = " << lambda << endl;
VectorXcd v = es.eigenvectors().col(0);
cout << "If v is the corresponding eigenvector, then lambda * v = " << endl << lambda
    * v << endl;
cout << "... and A * v = " << endl << A.cast<complex<double> >() * v << endl << endl;
MatrixXd D = es.eigenvalues().asDiagonal();MatrixXd V = es.eigenvectors();
cout << "Finally, V * D * V^(-1) = " << endl << V * D * V.inverse() << endl;

```

---

Output:

Here is a random 6x6 matrix, A:

```

0.68  -0.33  -0.27  -0.717  -0.687  0.0259
-0.211  0.536  0.0268  0.214  -0.198  0.678
0.566  -0.444  0.904  -0.967  -0.74   0.225
0.597  0.108  0.832  -0.514  -0.782  -0.408
0.823 -0.0452  0.271  -0.726  0.998  0.275
-0.605  0.258  0.435  0.608  -0.563  0.0486

```

The eigenvalues of A are:

```

(0.049,1.06)
(0.049,-1.06)
(0.967,0)
(0.353,0)
(0.618,0.129)
(0.618,-0.129)

```

The matrix of eigenvectors, V, is:

```

(-0.292,-0.454)  (-0.292,0.454)  (-0.0607,0)  (-0.733,0)  (0.59,-0.122)  (0.3
(0.134,-0.104)  (0.134,0.104)  (-0.799,0)  (0.136,0)  (0.335,0.368)  (0.3
(-0.422,-0.18)   (-0.422,0.18)   (0.192,0)  (0.0563,0)  (-0.335,-0.143)  (-0.
(-0.589,0.0274) (-0.589,-0.0274) (-0.0788,0)  (-0.627,0)  (0.322,-0.156)  (0.
(-0.248,0.132)   (-0.248,-0.132)  (0.401,0)  (0.218,0)  (-0.335,-0.076)  (-0.
(0.105,0.18)     (0.105,-0.18)   (-0.392,0)  (-0.00564,0)  (-0.0324,0.103)  (-0.032

```

Consider the first eigenvalue, lambda = (0.049,1.06)

If v is the corresponding eigenvector, then lambda \* v =

```

(0.466,-0.331)
(0.117,0.137)
(0.17,-0.456)
(-0.0578,-0.622)
(-0.152,-0.256)
(-0.186,0.12)
... and A * v =

```

```
(0.466,-0.331)
(0.117,0.137)
(0.17,-0.456)
(-0.0578,-0.622)
(-0.152,-0.256)
(-0.186,0.12)
```

Finally,  $V * D * V^{-1} =$

```
(0.68,1.9e-16)  (-0.33,4.82e-17)  (-0.27,-2.37e-16)  (-0.717,1.6e-16)  (-0.687,-2.2e-16)
(-0.211,2.22e-16)  (0.536,4.16e-17)  (0.0268,-2.98e-16)  (0.214,0)  (-0.198,6.1e-16)
(0.566,1.22e-15)  (-0.444,1.11e-16)  (0.904,-4.61e-16)  (-0.967,-3.61e-16)  (-0.74,7.1e-16)
(0.597,1.6e-15)  (0.108,1.84e-16)  (0.832,-5.6e-16)  (-0.514,-4.44e-16)  (-0.782,1.2e-16)
(0.823,-8.33e-16)  (-0.0452,-2.71e-16)  (0.271,5.53e-16)  (-0.726,7.77e-16)  (0.998,-2.1e-16)
(-0.605,1.03e-15)  (0.258,1.91e-16)  (0.435,-4.6e-16)  (0.608,-6.38e-16)  (-0.563,1.1e-16)
```

Computes eigenvalues and eigenvectors of general complex matrices.

This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the eigendecomposition; this is expected to be an instantiation of the Matrix class template.

The eigenvalues and eigenvectors of a matrix  $A$  are scalars  $\lambda$  and vectors  $v$  such that  $Av = \lambda v$ . If  $D$  is a diagonal matrix with the eigenvalues on the diagonal, and  $V$  is a matrix with the eigenvectors as its columns, then  $AV = VD$ . The matrix  $V$  is almost always invertible, in which case we have  $A = VDV^{-1}$ . This is called the eigendecomposition. The main function in this class is compute(), which computes the eigenvalues and eigenvectors of a given function. The documentation for that function contains an example showing the main features of the class.

See Also

class EigenSolver, class SelfAdjointEigenSolver

### 25.2.2 Complex Nonsymmetric Matrices

---

Function **cplxEigenNonsymm**(*A* As *mpNum*[,]) As *mpNum*[]

---

The function **cplxEigenNonsymm** returns the eigenvalues of a complex general (non-symmetric) matrix.

**Parameter:**

*A*: the complex general (non-symmetric) matrix of which we are computing the eigenvalues.

---

Function **cplxEigenNonsymmv**(*A* As *mpNum*[,]) As *mpNumList*[2]

---

The function **cplxEigenNonsymmv** returns the eigenvalues and eigenvectors of a complex general (non-symmetric) matrix.

**Parameter:**

*A*: the complex general (non-symmetric) matrix of which we are computing the eigenvalues.

Member Function DocumentationComplexEigenSolver::MatrixType *i* & compute ( const MatrixType & *matrix*, bool *computeEigenvectors* = true)

Computes eigendecomposition of given matrix. Parameters: [in] matrix Square matrix whose eigendecomposition is to be computed. [in] computeEigenvectors If true, both the eigenvectors and the eigenvalues are computed; if false, only the eigenvalues are computed.

Returns: Reference to `*this`

This function computes the eigenvalues of the complex matrix `matrix`. The `eigenvalues()` function can be used to retrieve them. If `computeEigenvectors` is true, then the eigenvectors are also computed and can be retrieved by calling `eigenvectors()`. The matrix is first reduced to Schur form using the `ComplexSchur` class. The Schur decomposition is then used to compute the eigenvalues and eigenvectors. The cost of the computation is dominated by the cost of the Schur decomposition, which is  $O(n^3)$  where  $n$  is the size of the matrix.

Example:

---

```
MatrixXcf A = MatrixXcf::Random(4,4);
cout << "Here is a random 4x4 matrix, A:" << endl << A << endl << endl;
ComplexEigenSolver<MatrixXcf> ces;
ces.compute(A);
cout << "The eigenvalues of A are:" << endl << ces.eigenvalues() << endl;
cout << "The matrix of eigenvectors, V, is:" << endl << ces.eigenvectors() << endl <<
endl;
complex<float> lambda = ces.eigenvalues()[0];
cout << "Consider the first eigenvalue, lambda = " << lambda << endl;
VectorXcf v = ces.eigenvectors().col(0);
cout << "If v is the corresponding eigenvector, then lambda * v = " << endl << lambda
* v << endl;
cout << "... and A * v = " << endl << A * v << endl << endl;
cout << "Finally, V * D * V^(-1) = "
<< endl
<< ces.eigenvectors() * ces.eigenvalues().asDiagonal() * ces.eigenvectors().inverse()
<< endl;
```

---

Output:

Here is a random 4x4 matrix, A:

(-0.211,0.68)	(0.108,-0.444)	(0.435,0.271)	(-0.198,-0.687)
(0.597,0.566)	(0.258,-0.0452)	(0.214,-0.717)	(-0.782,-0.74)
(-0.605,0.823)	(0.0268,-0.27)	(-0.514,-0.967)	(-0.563,0.998)
(0.536,-0.33)	(0.832,0.904)	(0.608,-0.726)	(0.678,0.0259)

The eigenvalues of A are:

(0.137,0.505)
(-0.758,1.22)
(1.52,-0.402)
(-0.691,-1.63)

The matrix of eigenvectors, V, is:

(-0.246,-0.106)	(0.418,0.263)	(0.0417,-0.296)	(-0.122,0.271)
(-0.205,-0.629)	(0.466,-0.457)	(0.244,-0.456)	(0.247,0.23)
(-0.432,-0.0359)	(-0.0651,-0.0146)	(-0.191,0.334)	(0.859,-0.0877)
(-0.301,0.46)	(-0.41,-0.397)	(0.623,0.328)	(-0.116,0.195)

Consider the first eigenvalue, lambda = (0.137,0.505)

If v is the corresponding eigenvector, then lambda \* v =

```

(0.0197,-0.139)
(0.29,-0.19)
(-0.0412,-0.223)
(-0.274,-0.0891)
... and A * v =
(0.0197,-0.139)
(0.29,-0.19)
(-0.0412,-0.223)
(-0.274,-0.0891)

Finally, V * D * V^(-1) =
(-0.211,0.68)  (0.108,-0.444)  (0.435,0.271)  (-0.198,-0.687)
(0.597,0.566)  (0.258,-0.0452)  (0.214,-0.717)  (-0.782,-0.74)
(-0.605,0.823)  (0.0268,-0.27)  (-0.514,-0.967)  (-0.563,0.998)
(0.536,-0.33)  (0.832,0.904)  (0.608,-0.726)  (0.678,0.0259)

```

References Eigen::Success. Referenced by ComplexEigenSolver $\langle$ MatrixType $\rangle$ ::ComplexEigenSolver().

const EigenvalueType& **eigenvalues** () const

Returns the eigenvalues of given matrix. Returns A const reference to the column vector containing the eigenvalues.

Precondition: Either the constructor ComplexEigenSolver(const MatrixType& matrix, bool) or the member function compute(const MatrixType& matrix, bool) has been called before to compute the eigendecomposition of a matrix. This function returns a column vector containing the eigenvalues. Eigenvalues are repeated according to their algebraic multiplicity, so there are as many eigenvalues as rows in the matrix. The eigenvalues are not sorted in any particular order. Example:

---

```

MatrixXcf ones = MatrixXcf::Ones(3,3);
ComplexEigenSolver<MatrixXcf> ces(ones, /* computeEigenvectors = */ false);
cout << "The eigenvalues of the 3x3 matrix of ones are:"
<< endl << ces.eigenvalues() << endl;

```

---

Output:

The eigenvalues of the 3x3 matrix of ones are:

```

(0,-0)
(0,0)
(3,0)

```

const EigenvectorType& **eigenvectors** () const

Returns the eigenvectors of given matrix. Returns A const reference to the matrix whose columns are the eigenvectors.

Precondition: Either the constructor ComplexEigenSolver(const MatrixType& matrix, bool) or the member function compute(const MatrixType& matrix, bool) has been called before to compute the eigendecomposition of a matrix, and computeEigenvectors was set to true (the default). This function returns a matrix whose columns are the eigenvectors. Column  $k$  is an eigenvector corresponding to eigenvalue number  $k$  as returned by eigenvalues(). The eigenvectors are normalized to have (Euclidean) norm equal to one. The matrix returned by this function is the matrix  $V$  in the eigendecomposition  $A = VDV^{-1}$ , if it exists.

Example:

---

```
MatrixXcf ones = MatrixXcf::Ones(3,3);
ComplexEigenSolver<MatrixXcf> ces(ones);
cout << "The first eigenvector of the 3x3 matrix of ones is:"
<< endl << ces.eigenvectors().col(1) << endl;
```

---

Output:

```
The first eigenvector of the 3x3 matrix of ones is:
(0.154,0)
(-0.772,0)
(0.617,0)
```

ComputationInfo **info** ( ) const

Reports whether previous computation was successful. Returns : Success if computation was successful, NoConvergence otherwise. References ComplexSchur<\_MatrixType>::info().

## 25.3 Generalized Eigensystems

Computes eigenvalues and eigenvectors of the generalized selfadjoint eigen problem. This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the eigendecomposition; this is expected to be an instantiation of the Matrix class template.

This class solves the generalized eigenvalue problem  $Av = \lambda Bv$ . In this case, the matrix  $A$  should be selfadjoint and the matrix  $B$  should be positive definite.

Only the lower triangular part of the input matrix is referenced.

Call the function compute() to compute the eigenvalues and eigenvectors of a given matrix. Alternatively, you can use the GeneralizedSelfAdjointEigenSolver(const MatrixType, const MatrixType, int) constructor which computes the eigenvalues and eigenvectors at construction time. Once the eigenvalue and eigenvectors are computed, they can be retrieved with the eigenvalues() and eigenvectors() functions.

GeneralizedSelfAdjointEigenSolver ( const MatrixType & matA, const MatrixType & matB, int options = ComputeEigenvectors—Ax\_lBx )

Constructor; computes generalized eigendecomposition of given matrix pencil. Parameters:

[in] matA Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

[in] matB Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

[in] options A or-ed set of flags ComputeEigenvectors,EigenvaluesOnly — Ax\_lBx,ABx\_lx,BAx\_lx. Default is ComputeEigenvectors—Ax\_lBx.

This constructor calls compute(const MatrixType&, const MatrixType&, int) to compute the eigenvalues and (if requested) the eigenvectors of the generalized eigenproblem  $Ax = \lambda Bx$  with matA the selfadjoint matrix  $A$  and matB the positive definite matrix  $B$ . Each eigenvector  $x$  satisfies the property  $x^T B x = 1$ . The eigenvectors are computed if options contains ComputeEigenvectors. In addition, the two following variants can be solved via options:

â€¢ ABx\_lx:  $ABx = \lambda x$

â€¢ BAx\_lx:  $BAx = \lambda x$ .

Example:

---

```
MatrixXd X = MatrixXd::Random(5,5);
MatrixXd A = X + X.transpose();
cout << "Here is a random symmetric matrix, A:" << endl << A << endl;
X = MatrixXd::Random(5,5);
MatrixXd B = X * X.transpose();
cout << "and a random positive-definite matrix, B:" << endl << B << endl << endl;
GeneralizedSelfAdjointEigenSolver<MatrixXd> es(A,B);
cout << "The eigenvalues of the pencil (A,B) are:" << endl << es.eigenvalues() << endl;
cout << "The matrix of eigenvectors, V, is:" << endl << es.eigenvectors() << endl << endl;
double lambda = es.eigenvalues()[0];
cout << "Consider the first eigenvalue, lambda = " << lambda << endl;
VectorXd v = es.eigenvectors().col(0);
cout << "If v is the corresponding eigenvector, then A * v = "
<< endl << A * v << endl;
cout << "... and lambda * B * v = " << endl << lambda * B * v << endl << endl;
```

---

Output:

Here is a random symmetric matrix, A:

```
1.36 -0.816  0.521   1.43 -0.144
-0.816 -0.659  0.794 -0.173 -0.406
 0.521  0.794 -0.541  0.461  0.179
 1.43 -0.173  0.461  -1.43  0.822
-0.144 -0.406  0.179  0.822 -1.37
```

and a random positive-definite matrix, B:

```
0.132  0.0109 -0.0512  0.0674 -0.143
 0.0109   1.68    1.13   -1.12  0.916
-0.0512   1.13    2.3    -2.14  1.86
 0.0674  -1.12   -2.14    2.69 -2.01
-0.143   0.916   1.86   -2.01  1.68
```

The eigenvalues of the pencil (A,B) are:

```
-227
-3.9
-0.837
0.101
54.2
```

The matrix of eigenvectors, V, is:

```
-14.2    1.03 -0.0766  0.0273   -8.36
-0.0546   0.115 -0.729  -0.478   0.374
 9.23   -0.624  0.0165  -0.499    3.01
 -7.88    -1.3  -0.225  -0.109   -3.85
-20.8   -0.805   0.567  0.0828   -8.73
```

Consider the first eigenvalue, lambda = -227

If v is the corresponding eigenvector, then A \* v =

```
-22.8
28.8
-19.8
-21.9
25.9
... and lambda * B * v =
-22.8
28.8
-19.8
-21.9
25.9
```

### 25.3.1 Real Generalized Symmetric-Definite Eigensystems

---

Function **EigenGensymm**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*

---

The function **EigenGensymm** returns the eigenvalues of a real Generalized Symmetric-Definite Eigensystem.

**Parameters:**

- A*: Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.  
*B*: Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

---

**Function `EigenGensymmv(A As mpNum[,], B As mpNum[,]) As mpNum`**

---

The function `EigenGensymmv` returns the eigenvalues and eigenvectors of a real Generalized Symmetric-Definite Eigensystem.

**Parameters:**

- A*: Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.  
*B*: Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

Member Function Documentation

`GeneralizedSelfAdjointEigenSolver<MatrixType>::MatrixType & compute ( const MatrixType & matA, const MatrixType & matB, int options = ComputeEigenvectors—Ax_lBx )`

Computes generalized eigendecomposition of given matrix pencil. Parameters [in] `matA` Self-adjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced. [in] `matB` Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced. [in] `options` A or-ed set of flags `ComputeEigenvectors`,`EigenvaluesOnly` — `Ax_lBx`,`ABx_lx`,`BAx_lx`. Default is `ComputeEigenvectors—Ax_lBx`.

Returns Reference to `*this` According to options, this function computes eigenvalues and (if requested) the eigenvectors of one of the following three generalized eigenproblems:

â€¢  $Ax = \lambda Bx$

â€¢  $ABx = \lambda x$

â€¢  $BAx = \lambda x$

with `matA` the selfadjoint matrix  $A$  and `matB` the positive definite matrix  $B$ . In addition, each eigenvector satisfies the property  $x^*Bx = 1$ . The `eigenvalues()` function can be used to retrieve the eigenvalues. If options contains `ComputeEigenvectors`, then the eigenvectors are also computed and can be retrieved by calling `eigenvectors()`.

The implementation uses LLT to compute the Cholesky decomposition  $B = LL^*$  and computes the classical eigendecomposition of the selfadjoint matrix  $L^{-1}A(L^*)^{-1}$  if options contains `Ax_lBx` and of  $L^*AL$  otherwise. This solves the generalized eigenproblem, because any solution of the generalized eigenproblem  $Ax = \lambda Bx$  corresponds to a solution  $L^{-1}A(L^*)^{-1}(L^*x) = \lambda(L^*x)$  of the eigenproblem for  $L^{-1}A(L^*)^{-1}$ . Similar statements can be made for the two other variants.

Example:

---

```
MatrixXd X = MatrixXd::Random(5,5);
MatrixXd A = X * X.transpose();
X = MatrixXd::Random(5,5);
MatrixXd B = X * X.transpose();
GeneralizedSelfAdjointEigenSolver<MatrixXd> es(A,B,EigenvaluesOnly);
cout << "The eigenvalues of the pencil (A,B) are:" << endl << es.eigenvalues() <<
    endl;
es.compute(B,A,false);
cout << "The eigenvalues of the pencil (B,A) are:" << endl << es.eigenvalues() <<
    endl;
```

---

Output:

The eigenvalues of the pencil (A,B) are:

0.0289  
0.299  
2.11  
8.64  
2.08e+03

The eigenvalues of the pencil (B,A) are:

0.000481  
0.116  
0.473  
3.34  
34.6

`SelfAdjointEigenSolver<MatrixType> & compute ( const MatrixType & matrix, int options = ComputeEigenvectors )`

Computes eigendecomposition of given matrix. Parameters [in] matrix Selfadjoint matrix whose eigendecomposition is to be computed. Only the lower triangular part of the matrix is referenced. [in] options Can be ComputeEigenvectors (default) or EigenvaluesOnly.

Returns Reference to `*this`

This function computes the eigenvalues of matrix. The `eigenvalues()` function can be used to retrieve them. If options equals `ComputeEigenvectors`, then the eigenvectors are also computed and can be retrieved by calling `eigenvectors()`.

This implementation uses a symmetric QR algorithm. The matrix is first reduced to tridiagonal form using the `Tridiagonalization` class. The tridiagonal matrix is then brought to diagonal form with implicit symmetric QR steps with Wilkinson shift. Details can be found in Section 8.3 of [Golub & Van Loan \(1996\)](#). The cost of the computation is about  $9n^3$  if the eigenvectors are required and  $4n^3/3$  if they are not required.

This method reuses the memory in the `SelfAdjointEigenSolver` object that was allocated when the object was constructed, if the size of the matrix does not change.

Example:

---

```
SelfAdjointEigenSolver<MatrixXf> es(4);
MatrixXf X = MatrixXf::Random(4,4);
MatrixXf A = X + X.transpose(); es.compute(A);
cout << "The eigenvalues of A are: " << es.eigenvalues().transpose() << endl;
es.compute(A + MatrixXf::Identity(4,4)); // re-use es to compute eigenvalues of A+I
cout << "The eigenvalues of A+I are: " << es.eigenvalues().transpose() << endl;
```

---

Output:

The eigenvalues of A are: -1.58 -0.473 1.32 2.46  
The eigenvalues of A+I are: -0.581 0.527 2.32 3.46

`SelfAdjointEigenSolver<MatrixType> & computeDirect ( const MatrixType & matrix, int options = ComputeEigenvectors )`

Computes eigendecomposition of given matrix using a direct algorithm. This is a variant of `compute(const MatrixType&, int options)` which directly solves the underlying polynomial equation. Currently only 3x3 matrices for which the sizes are known at compile time are supported (e.g., `Matrix3d`). This method is usually significantly faster than the QR algorithm but it might also be less accurate. It is also worth noting that for 3x3 matrices it involves trigonometric operations

which are not necessarily available for all scalar types. See Also `compute(const MatrixType&, int options)`

`const RealVectorType& eigenvalues ( ) const`

Returns the eigenvalues of given matrix. Returns A const reference to the column vector containing the eigenvalues. Precondition The eigenvalues have been computed before. The eigenvalues are repeated according to their algebraic multiplicity, so there are as many eigenvalues as rows in the matrix. The eigenvalues are sorted in increasing order.

Example:

---

```
MatrixXd ones = MatrixXd::Ones(3,3);
SelfAdjointEigenSolver<MatrixXd> es(ones);
cout << "The eigenvalues of the 3x3 matrix of ones are:"
<< endl << es.eigenvalues() << endl;
```

---

Output:

```
The eigenvalues of the 3x3 matrix of ones are:
-3.09e-16
0
3
```

`const MatrixType& eigenvectors ( ) const inline inherited`

Returns the eigenvectors of given matrix. Returns A const reference to the matrix whose columns are the eigenvectors. Precondition The eigenvectors have been computed before. Column  $k$  of the returned matrix is an eigenvector corresponding to eigenvalue number  $k$  as returned by `eigenvalues()`. The eigenvectors are normalized to have (Euclidean) norm equal to one. If this object was used to solve the eigenproblem for the selfadjoint matrix  $A$ , then the matrix returned by this function is the matrix  $V$  in the eigendecomposition  $A = VDV^{-1}$ .

Example:

---

```
MatrixXd ones = MatrixXd::Ones(3,3);
SelfAdjointEigenSolver<MatrixXd> es(ones);
cout << "The first eigenvector of the 3x3 matrix of ones is:"
<< endl << es.eigenvectors().col(1) << endl;
```

---

Output:

```
The first eigenvector of the 3x3 matrix of ones is:
0
-0.707
0.707
```

`ComputationInfo info ( ) const`

Reports whether previous computation was successful. Returns Success if computation was successful, NoConvergence otherwise.

Member Data Documentation `const int m_maxIterations static inherited`

Maximum number of iterations. The algorithm terminates if it does not converge within `m_maxIterations * n` iterations, where `n` denotes the size of the matrix. This value is currently set to 30 (copied from LAPACK).

### 25.3.2 Complex Hermitian Generalized Symmetric-Definite Eigensystems

---

#### Function **cplxEigenGenherm**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*

---

The function `cplxEigenGenherm` returns the eigenvalues of a Complex Hermitian Generalized Symmetric-Definite Eigensystem.

**Parameters:**

*A*: Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.  
*B*: Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

---

#### Function **cplxEigenGenhermv**(*A* As *mpNum*[,], *B* As *mpNum*[,]) As *mpNum*

---

The function `cplxEigenGenhermv` returns the eigenvalues and eigenvectors of a Complex Hermitian Generalized Symmetric-Definite Eigensystem.

**Parameters:**

*A*: Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.  
*B*: Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

GeneralizedSelfAdjointEigenSolver ( const MatrixType & matA, const MatrixType & matB, int options = ComputeEigenvectors—Ax\_lBx )

Constructor; computes generalized eigendecomposition of given matrix pencil. Parameters:  
[in] matA Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

[in] matB Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

[in] options A or-ed set of flags ComputeEigenvectors,EigenvaluesOnly — Ax\_lBx,ABx\_lx,BAx\_lx. Default is ComputeEigenvectors—Ax\_lBx.

This constructor calls `compute(const MatrixType&, const MatrixType&, int)` to compute the eigenvalues and (if requested) the eigenvectors of the generalized eigenproblem  $Ax = \lambda Bx$  with matA the selfadjoint matrix *A* and matB the positive definite matrix *B*. Each eigenvector *x* satisfies the property  $x^* B x = 1$ . The eigenvectors are computed if options contains ComputeEigenvectors. In addition, the two following variants can be solved via options:

â€¢ ABx\_lx:  $ABx = \lambda x$

â€¢ BAx\_lx:  $BAx = \lambda x$ .

### 25.3.3 Real Generalized Nonsymmetric Eigensystem

---

#### Function **EigenGenNonsymm**(*A* As *mpNum*[, *B* As *mpNum*[,]) As *mpNum*

---

The function `EigenGenNonsymm` returns the eigenvalues of a real Generalized Non-Symmetric Eigensystem.

**Parameters:**

*A*: Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.  
*B*: Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

---

#### Function **EigenGenNonsymmv**(*A* As *mpNum*[, *B* As *mpNum*[,]) As *mpNum*

---

The function `EigenGenNonsymmv` returns the eigenvalues and eigenvectors of a real Generalized Non-Symmetric Eigensystem.

**Parameters:**

*A*: Selfadjoint matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.  
*B*: Positive-definite matrix in matrix pencil. Only the lower triangular part of the matrix is referenced.

Computes the generalized eigenvalues and eigenvectors of a pair of general (nonsymmetric) matrices.

This is defined in the `Eigenvalues` module.

`MatrixType` the type of the matrices of which we are computing the eigen-decomposition; this is expected to be an instantiation of the `Matrix` class template. Currently, only real matrices are supported.

The generalized eigenvalues and eigenvectors of a matrix pair *A* and *B* are scalars  $\lambda$  and vectors *v* such that  $Av = \lambda Bv$ . If *D* is a diagonal matrix with the eigenvalues on the diagonal, and *V* is a matrix with the eigenvectors as its columns, then  $AV = BVD$ . The matrix *V* is almost always invertible, in which case we have  $A = BVDV^{-1}$ . This is called the generalized eigen-decomposition.

The generalized eigenvalues and eigenvectors of a matrix pair may be complex, even when the matrices are real. Moreover, the generalized eigenvalue might be infinite if the matrix *B* is singular. To workaround this difficulty, the eigenvalues are provided as a pair of complex  $\alpha$  and real  $\beta$  such that:  $\lambda_i = \alpha_i/\beta_i$ . If  $\beta_i$  is (nearly) zero, then one can consider the well defined left eigenvalue  $\mu = \beta_i/\alpha_i$  such that:  $\mu_i Av_i = Bv_i$ , or even  $\mu_i u_i^T A = u_i^T B$  where *u<sub>i</sub>* is called the left eigenvector.

Call the function `compute()` to compute the generalized eigenvalues and eigenvectors of a given matrix pair.

Alternatively, you can use the `GeneralizedEigenSolver(const MatrixType, const MatrixType, bool)` constructor which computes the eigenvalues and eigenvectors at construction time. Once the eigenvalue and eigenvectors are computed, they can be retrieved with the `eigenvalues()` and `eigenvectors()` functions.

#### 25.3.3.1 Member Function Documentation

`ComplexVectorType alphas( ) const`

Returns A const reference to the vectors containing the alpha values

This vector permits to reconstruct the *j*-th eigenvalues as `alphas(i)/betas(j)`.

VectorType **betas** ( ) const

Returns A const reference to the vectors containing the beta values. This vector permits to reconstruct the j-th eigenvalues as  $\text{alphas}(i)/\text{betas}(j)$ .

GeneralizedEigenSolver $\langle$ MatrixType $\rangle$  & **compute** ( const MatrixType & A, const MatrixType & B, bool computeEigenvectors = true )

Computes generalized eigendecomposition of given matrix. Parameters

[in] A Square matrix whose eigendecomposition is to be computed.

[in] B Square matrix whose eigendecomposition is to be computed.

[in] computeEigenvectors If true, both the eigenvectors and the eigenvalues are computed; if false, only the eigenvalues are computed.

Returns Reference to \*this

This function computes the eigenvalues of the real matrix matrix. The eigenvalues() function can be used to retrieve them. If computeEigenvectors is true, then the eigenvectors are also computed and can be retrieved by calling eigenvectors().

The matrix is first reduced to real generalized Schur form using the RealQZ class. The generalized Schur decomposition is then used to compute the eigenvalues and eigenvectors.

The cost of the computation is dominated by the cost of the generalized Schur decomposition.

This method reuses of the allocated data in the GeneralizedEigenSolver object.

EigenvalueType **eigenvalues** ( ) const

Returns an expression of the computed generalized eigenvalues. Returns An expression of the column vector containing the eigenvalues.

It is a shortcut for this->alphas().cwiseQuotient(this->betas()); Note that betas might contain zeros. It is therefore not recommended to use this function, but rather directly deal with the alphas and betas vectors.

Precondition:

Either the constructor GeneralizedEigenSolver(const MatrixType&,const MatrixType&,bool) or the member function compute(const MatrixType&,const MatrixType&,bool) has been called before.

The eigenvalues are repeated according to their algebraic multiplicity, so there are as many eigenvalues as rows in the matrix. The eigenvalues are not sorted in any particular order.

GeneralizedEigenSolver& **setMaxIterations** ( Index maxIters)

Sets the maximal number of iterations allowed.

### 25.3.3.2 Example

Here is an usage example of this class:

Example:

---

```
GeneralizedEigenSolver<MatrixXf> ges;
MatrixXf A = MatrixXf::Random(4,4);
MatrixXf B = MatrixXf::Random(4,4);
ges.compute(A, B);
cout << "The (complex) numerators of the generalized eigenvalues are: "
<< ges.alphas().transpose() << endl;
cout << "The (real) denominators of the generalized eigenvalues are: "
<< ges.betas().transpose() << endl;
cout << "The (complex) generalized eigenvalues are (alphas./beta): "
<< ges.eigenvalues().transpose() << endl;
```

---

Output:

The (complex) numerators of the generalized eigenvalues are:  
(0.644,0.795) (0.644,-0.795) (-0.398,0) (-1.12,0)

The (real) denominatore of the generalized eigenvalues are:  
1.51 1.51 -1.25 0.746

The (complex) generalized eigenvalues are (alphas./beta):  
(0.427,0.528) (0.427,-0.528) (0.318,-0) (-1.5,0)

## 25.4 Decompositions

### 25.4.1 Tridiagonalization

Tridiagonal decomposition of a selfadjoint matrix.

This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the tridiagonal decomposition; this is expected to be an instantiation of the Matrix class template.

This class performs a tridiagonal decomposition of a selfadjoint matrix  $A$  such that:  $A = QTQ^*$  where  $Q$  is unitary and  $T$  a real symmetric tridiagonal matrix. A tridiagonal matrix is a matrix which has nonzero elements only on the main diagonal and the first diagonal below and above it. The Hessenberg decomposition of a selfadjoint matrix is in fact a tridiagonal decomposition. This class is used in SelfAdjointEigenSolver to compute the eigenvalues and eigenvectors of a selfadjoint matrix.

Call the function compute() to compute the tridiagonal decomposition of a given matrix. Alternatively, you can use the Tridiagonalization(const MatrixType) constructor which computes the tridiagonal Schur decomposition at construction time. Once the decomposition is computed, you can use the matrixQ() and matrixT() functions to retrieve the matrices  $Q$  and  $T$  in the decomposition.

The documentation of Tridiagonalization(const MatrixType) contains an example of the typical use of this class.

Example:

---

```
MatrixXd X = MatrixXd::Random(5,5);
MatrixXd A = X + X.transpose();
cout << "Here is a random symmetric 5x5 matrix:" << endl << A << endl << endl;
Tridiagonalization<MatrixXd> triOfA(A);
MatrixXd Q = triOfA.matrixQ();
cout << "The orthogonal matrix Q is:" << endl << Q << endl;
MatrixXd T = triOfA.matrixT();
cout << "The tridiagonal matrix T is:" << endl << T << endl << endl;
cout << "Q * T * Q^T = " << endl << Q * T * Q.transpose() << endl;
```

---

Output:

Here is a random symmetric 5x5 matrix:

```
1.36 -0.816  0.521   1.43 -0.144
-0.816 -0.659  0.794 -0.173 -0.406
 0.521  0.794 -0.541  0.461  0.179
 1.43 -0.173  0.461  -1.43  0.822
-0.144 -0.406  0.179  0.822  -1.37
```

The orthogonal matrix Q is:

```
1       0       0       0       0
0   -0.471    0.127   -0.671   -0.558
0    0.301   -0.195    0.437   -0.825
0    0.825    0.0459   -0.563 -0.00872
0  -0.0832   -0.971   -0.202    0.0922
```

The tridiagonal matrix T is:

```
1.36   1.73      0      0      0
```

```

1.73 -1.2 -0.966 0 0
0 -0.966 -1.28 0.214 0
0 0 0.214 -1.69 0.345
0 0 0 0.345 0.164

Q * T * Q^T =
1.36 -0.816 0.521 1.43 -0.144
-0.816 -0.659 0.794 -0.173 -0.406
0.521 0.794 -0.541 0.461 0.179
1.43 -0.173 0.461 -1.43 0.822
-0.144 -0.406 0.179 0.822 -1.37

```

#### 25.4.1.1 Member Function Documentation

Member Function Documentation

Tridiagonalization& **compute** ( const MatrixType & matrix)

Computes tridiagonal decomposition of given matrix. Parameters: [in] matrix Selfadjoint matrix whose tridiagonal decomposition is to be computed.

Returns Reference to \*this

The tridiagonal decomposition is computed by bringing the columns of the matrix successively in the required form using Householder reflections. The cost is flops, where denotes the size of the given matrix. This method reuses of the allocated data in the Tridiagonalization object, if the size of the matrix does not change.

Example:

---

```

Tridiagonalization<MatrixXf> tri;
MatrixXf X = MatrixXf::Random(4,4);
MatrixXf A = X + X.transpose();
tri.compute(A);
cout << "The matrix T in the tridiagonal decomposition of A is: " << endl;
cout << tri.matrixT() << endl;
tri.compute(2*A); // re-use tri to compute eigenvalues of 2A
cout << "The matrix T in the tridiagonal decomposition of 2A is: " << endl;
cout << tri.matrixT() << endl;

```

---

Output:

The matrix T in the tridiagonal decomposition of A is:

```

1.36 -0.704 0 0
-0.704 0.0147 1.71 0
0 1.71 0.856 0.641
0 0 0.641 -0.506

```

The matrix T in the tridiagonal decomposition of 2A is:

```

2.72 -1.41 0 0
-1.41 0.0294 3.43 0
0 3.43 1.71 1.28
0 0 1.28 -1.01

```

Tridiagonalization<MatrixType>::DiagonalReturnType **diagonal** ( ) const

Returns the diagonal of the tridiagonal matrix T in the decomposition. Returns expression representing the diagonal of T Precondition Either the constructor `Tridiagonalization(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the tridiagonal decomposition of a matrix.

Example:

---

```
MatrixXcd X = MatrixXcd::Random(4,4);
MatrixXcd A = X + X.adjoint();
cout << "Here is a random self-adjoint 4x4 matrix:" << endl << A << endl << endl;
Tridiagonalization<MatrixXcd> triOfA(A);
MatrixXd T = triOfA.matrixT();
cout << "The tridiagonal matrix T is:" << endl << T << endl << endl;
cout << "We can also extract the diagonals of T directly ..." << endl;
VectorXd diag = triOfA.diagonal();
cout << "The diagonal is:" << endl << diag << endl;
VectorXd subdiag = triOfA.subDiagonal();
cout << "The subdiagonal is:" << endl << subdiag << endl;
```

---

Output:

```
Here is a random self-adjoint 4x4 matrix:
(-0.422,0)  (0.705,-1.01)  (-0.17,-0.552)  (0.338,-0.357)
(0.705,1.01)           (0.515,0)  (0.241,-0.446)  (0.05,-1.64)
(-0.17,0.552)  (0.241,0.446)           (-1.03,0)  (0.0449,1.72)
(0.338,0.357)  (0.05,1.64)  (0.0449,-1.72)           (1.36,0)
```

The tridiagonal matrix T is:

```
-0.422  -1.45      0      0
-1.45    1.01  -1.42      0
  0  -1.42      1.8  -1.2
  0      0  -1.2  -1.96
```

We can also extract the diagonals of T directly ...

The diagonal is:

```
-0.422
1.01
1.8
-1.96
```

The subdiagonal is:

```
-1.45
-1.42
-1.2
```

See Also `matrixT()`, `subDiagonal()`

`CoeffVectorType householderCoefficients ( ) const`

Returns the Householder coefficients. Returns a const reference to the vector of Householder coefficients

Precondition: Either the constructor `Tridiagonalization(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the tridiagonal decompo-

sition of a matrix. The Householder coefficients allow the reconstruction of the matrix  $Q$  in the tridiagonal decomposition from the packed data.

Example:

---

```
Matrix4d X = Matrix4d::Random(4,4);
Matrix4d A = X + X.transpose();
cout << "Here is a random symmetric 4x4 matrix:" << endl << A << endl;
Tridiagonalization<Matrix4d> triOfA(A);
Vector3d hc = triOfA.householderCoefficients();
cout << "The vector of Householder coefficients is:" << endl << hc << endl;
```

---

Output:

Here is a random symmetric 4x4 matrix:

```
1.36  0.612  0.122  0.326
0.612 -1.21   -0.222  0.563
0.122 -0.222 -0.0904  1.16
0.326  0.563   1.16   1.66
```

The vector of Householder coefficients is:

```
1.87
1.24
0
```

See Also `packedMatrix()`, Householder module

HouseholderSequenceType **matrixQ** ( ) const

Returns the unitary matrix  $Q$  in the decomposition.

Returns object representing the matrix  $Q$

Precondition:

Either the constructor `Tridiagonalization(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the tridiagonal decomposition of a matrix.

This function returns a light-weight object of template class `HouseholderSequence`. You can either apply it directly to a matrix or you can convert it to a matrix of type `MatrixType`.

See Also `Tridiagonalization(const MatrixType&)` for an example, `matrixT()`, class `HouseholderSequence`

`MatrixTReturnType matrixT` ( ) const

Returns an expression of the tridiagonal matrix  $T$  in the decomposition.

Returns expression object representing the matrix  $T$

Precondition:

Either the constructor `Tridiagonalization(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the tridiagonal decomposition of a matrix.

Currently, this function can be used to extract the matrix  $T$  from internal data and copy it to a dense matrix object. In most cases, it may be sufficient to directly use the packed matrix or the vector expressions returned by `diagonal()` and `subDiagonal()` instead of creating a new dense copy matrix with this function.

`const MatrixType& packedMatrix` ( ) const

Returns the internal representation of the decomposition.

Returns a const reference to a matrix with the internal representation of the decomposition.

Precondition:

Either the constructor `Tridiagonalization(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the tridiagonal decomposition of a matrix.

The returned matrix contains the following information:

the strict upper triangular part is equal to the input matrix  $A$ .

the diagonal and lower sub-diagonal represent the real tridiagonal symmetric matrix  $T$ .

the rest of the lower part contains the Householder vectors that, combined with Householder coefficients returned by `householderCoefficients()`, allows to reconstruct the matrix  $Q$  as  $Q = H_{N-1} \dots H_1 H_0$ . Here, the matrices  $H_i$  are the Householder transformations  $H_i = (I - h_i v_i v_i^T)$  where  $h_i$  is the  $i$ th Householder coefficient and  $v_i$  is the Householder vector defined by  $v_i = [0, \dots, 0, 1, M(i+2, i), \dots, M(N-1, i)]^T$  with  $M$  the matrix returned by this function. See LAPACK for further details on this packed storage.

Example:

---

```
Matrix4d X = Matrix4d::Random(4,4);
Matrix4d A = X + X.transpose();
cout << "Here is a random symmetric 4x4 matrix:" << endl << A << endl;
Tridiagonalization<Matrix4d> triOfA(A);
Matrix4d pm = triOfA.packedMatrix();
cout << "The packed matrix M is:" << endl << pm << endl;
cout << "The diagonal and subdiagonal corresponds to the matrix T, which is:"
<< endl << triOfA.matrixT() << endl;
```

---

Output:

Here is a random symmetric 4x4 matrix:

```
1.36  0.612  0.122  0.326
0.612  -1.21  -0.222  0.563
0.122  -0.222 -0.0904  1.16
0.326  0.563  1.16   1.66
```

The packed matrix  $M$  is:

```
1.36  0.612  0.122  0.326
-0.704 0.0147 -0.222  0.563
0.0925  1.71   0.856  1.16
0.248   0.785  0.641 -0.506
```

The diagonal and subdiagonal corresponds to the matrix  $T$ , which is:

```
1.36 -0.704      0      0
-0.704 0.0147    1.71    0
0     1.71   0.856  0.641
0     0     0.641 -0.506
```

See Also `householderCoefficients()`

`Tridiagonalization<MatrixType>::SubDiagonalReturnType subDiagonal( ) const`  
 Returns the subdiagonal of the tridiagonal matrix  $T$  in the decomposition.

Returns expression representing the subdiagonal of  $T$

Precondition: Either the constructor `Tridiagonalization(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the tridiagonal decomposition of a matrix.

## 25.4.2 Hessenberg Decomposition

Reduces a square matrix to Hessenberg form by an orthogonal similarity transformation. This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the Hessenberg decomposition. This class performs an Hessenberg decomposition of a matrix  $A$ .

In the real case, the Hessenberg decomposition consists of an orthogonal matrix  $Q$  and a Hessenberg matrix  $H$  such that  $A = QHQ^T$ . An orthogonal matrix is a matrix whose inverse equals its transpose ( $Q^{-1} = Q^T$ ). A Hessenberg matrix has zeros below the subdiagonal, so it is almost upper triangular.

The Hessenberg decomposition of a complex matrix is  $A = QHQ^*$  with  $Q$  unitary (that is,  $Q^{-1} = Q^*$ ).

Call the function `compute()` to compute the Hessenberg decomposition of a given matrix. Alternatively, you can use the `HessenbergDecomposition(const MatrixType)` constructor which computes the Hessenberg decomposition at construction time. Once the decomposition is computed, you can use the `matrixH()` and `matrixQ()` functions to construct the matrices  $H$  and  $Q$  in the decomposition.

The documentation for `matrixH()` contains an example of the typical use of this class.

See Also

class ComplexSchur, class Tridiagonalization, QR Module

### 25.4.2.1 Member Function Documentation

`HessenbergDecomposition& compute ( const MatrixType & matrix)`

Computes Hessenberg decomposition of given matrix. Parameters [in] `matrix` Square matrix whose Hessenberg decomposition is to be computed.

Returns Reference to `*this` The Hessenberg decomposition is computed by bringing the columns of the matrix successively in the required form using Householder reflections (see, e.g., Algorithm 7.4.2 in [Golub & Van Loan \(1996\)](#)). The cost is  $10n^3/3$  flops, where  $n$  denotes the size of the given matrix. This method reuses of the allocated data in the `HessenbergDecomposition` object.

Example:

---

```
MatrixXcf A = MatrixXcf::Random(4,4);
HessenbergDecomposition<MatrixXcf> hd(4);
hd.compute(A);
cout << "The matrix H in the decomposition of A is:" << endl << hd.matrixH() << endl;
hd.compute(2*A); // re-use hd to compute and store decomposition of 2A
cout << "The matrix H in the decomposition of 2A is:" << endl << hd.matrixH() << endl;
```

---

Output:

The matrix H in the decomposition of A is:

(-0.211,0.68)	(0.346,0.216)	(-0.688,0.00979)	(0.0451,0.584)
(-1.45,0)	(-0.0574,-0.0123)	(-0.196,0.385)	(0.395,0.389)
(0,0)	(1.68,0)	(-0.397,-0.552)	(0.156,-0.241)
(0,0)	(0,0)	(1.56,0)	(0.876,-0.423)

The matrix H in the decomposition of 2A is:

(-0.422,1.36)	(0.691,0.431)	(-1.38,0.0196)	(0.0902,1.17)
(-2.91,0)	(-0.115,-0.0246)	(-0.392,0.77)	(0.791,0.777)
(0,0)	(3.36,0)	(-0.795,-1.1)	(0.311,-0.482)
(0,0)	(0,0)	(3.12,0)	(1.75,-0.846)

```
const CoeffVectorType& householderCoefficients ( ) const
```

Returns the Householder coefficients. Returns a const reference to the vector of Householder coefficients Precondition Either the constructor HessenbergDecomposition(const MatrixType&) or the member function compute(const MatrixType&) has been called before to compute the Hessenberg decomposition of a matrix. The Householder coefficients allow the reconstruction of the matrix in the Hessenberg decomposition from the packed data. See Also packedMatrix(), Householder module

```
MatrixHReturnType matrixH ( ) const
```

Constructs the Hessenberg matrix H in the decomposition. Returns expression object representing the matrix H Precondition Either the constructor HessenbergDecomposition(const MatrixType&) or the member function compute(const MatrixType&) has been called before to compute the Hessenberg decomposition of a matrix. The object returned by this function constructs the Hessenberg matrix H when it is assigned to a matrix or otherwise evaluated. The matrix H is constructed from the packed matrix as returned by packedMatrix(): The upper part (including the subdiagonal) of the packed matrix contains the matrix H. It may sometimes be better to directly use the packed matrix instead of constructing the matrix H.

Example:

---

```
MatrixXf A = MatrixXf::Random(4,4);
cout << "Here is a random 4x4 matrix:" << endl << A << endl;
HessenbergDecomposition<MatrixXf> hessOfA(A);
MatrixXf H = hessOfA.matrixH();
cout << "The Hessenberg matrix H is:" << endl << H << endl;
MatrixXf Q = hessOfA.matrixQ();
cout << "The orthogonal matrix Q is:" << endl << Q << endl;
cout << "Q * H * Q^T is:" << endl << Q * H * Q.transpose() << endl;
```

---

Output:

```
Here is a random 4x4 matrix:
0.68  0.823 -0.444  -0.27
-0.211 -0.605  0.108  0.0268
0.566  -0.33  -0.0452  0.904
0.597  0.536  0.258  0.832
The Hessenberg matrix H is:
0.68 -0.691 -0.645  0.235
0.849  0.836 -0.419  0.794
0 -0.469 -0.547 -0.0731
0  0 -0.559 -0.107
The orthogonal matrix Q is:
1  0  0  0
0 -0.249 -0.958  0.144
0  0.667 -0.277 -0.692
0  0.703 -0.0761  0.707
Q * H * Q^T is:
0.68  0.823 -0.444  -0.27
-0.211 -0.605  0.108  0.0268
0.566  -0.33  -0.0452  0.904
0.597  0.536  0.258  0.832
```

See Also `matrixQ()`, `packedMatrix()`

HouseholderSequenceType **matrixQ** ( ) const

Reconstructs the orthogonal matrix  $Q$  in the decomposition. Returns object representing the matrix  $Q$  Precondition Either the constructor `HessenbergDecomposition(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the Hessenberg decomposition of a matrix. This function returns a light-weight object of template class `HouseholderSequence`. You can either apply it directly to a matrix or you can convert it to a matrix of type `MatrixType`. See Also `matrixH()` for an example, class `HouseholderSequence`

const `MatrixType& packedMatrix` ( ) const

Returns the internal representation of the decomposition. Returns a const reference to a matrix with the internal representation of the decomposition.

Precondition Either the constructor `HessenbergDecomposition(const MatrixType&)` or the member function `compute(const MatrixType&)` has been called before to compute the Hessenberg decomposition of a matrix.

The returned matrix contains the following information:

â€¢ the upper part and lower sub-diagonal represent the Hessenberg matrix  $H$

â€¢ the rest of the lower part contains the Householder vectors that, combined with Householder coefficients returned by `householderCoefficients()`, allows to reconstruct the matrix  $Q$  as  $Q = H_N - 1 \dots H_1 H_0$ . Here, the matrices  $H_i$  are the Householder transformations  $H_i = (I - h_i v_i v_i^T)$  where  $h_i$  is the  $i$ th Householder coefficient and  $v_i$  is the Householder vector defined by  $v_i = [0, \dots, 0, 1, M(i+2, i), \dots, M(N-1, i)]^T$  with  $M$  the matrix returned by this function. See LAPACK for further details on this packed storage.

Example:

---

```
Matrix4d A = Matrix4d::Random(4,4);
cout << "Here is a random 4x4 matrix:" << endl << A << endl;
HessenbergDecomposition<Matrix4d> hessOfA(A);
Matrix4d pm = hessOfA.packedMatrix();
cout << "The packed matrix M is:" << endl << pm << endl;
cout << "The upper Hessenberg part corresponds to the matrix H, which is:" << endl << hessOfA.matrixH() << endl;
Vector3d hc = hessOfA.householderCoefficients();
cout << "The vector of Householder coefficients is:" << endl << hc << endl;
```

---

Output:

Here is a random 4x4 matrix:

```
0.68  0.823 -0.444 -0.27
-0.211 -0.605  0.108  0.0268
0.566  -0.33 -0.0452  0.904
0.597  0.536  0.258  0.832
```

The packed matrix M is:

```
0.68 -0.691 -0.645  0.235
0.849  0.836 -0.419  0.794
-0.534 -0.469 -0.547 -0.0731
-0.563  0.344 -0.559 -0.107
```

The upper Hessenberg part corresponds to the matrix H, which is:

```
0.68 -0.691 -0.645  0.235
0.849  0.836 -0.419  0.794
```

```
0 -0.469 -0.547 -0.0731
0 0 -0.559 -0.107
```

The vector of Householder coefficients is:

```
1.25
1.79
0
```

See Also `householderCoefficients()`

### 25.4.3 Real QZ Decomposition

Performs a real QZ decomposition of a pair of square matrices.

This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the real QZ decomposition; this is expected to be an instantiation of the Matrix class template.

Given a real square matrices  $A$  and  $B$ , this class computes the real QZ decomposition:  $A = QSZ$ ,  $B = QTZ$  where  $Q$  and  $Z$  are real orthogonal matrixes,  $T$  is upper-triangular matrix, and  $S$  is upper quasi-triangular matrix. An orthogonal matrix is a matrix whose inverse is equal to its transpose,  $U^{-1} = U^T$ . A quasi-triangular matrix is a block-triangular matrix whose diagonal consists of 1-by-1 blocks and 2-by-2 blocks where further reduction is impossible due to complex eigenvalues.

The eigenvalues of the pencil  $A - zB$  can be obtained from 1x1 and 2x2 blocks on the diagonals of  $S$  and  $T$ .

Call the function compute() to compute the real QZ decomposition of a given pair of matrices. Alternatively, you can use the RealQZ(const MatrixType B, const MatrixType B, bool computeQZ) constructor which computes the real QZ decomposition at construction time. Once the decomposition is computed, you can use the matrixS(), matrixT(), matrixQ() and matrixZ() functions to retrieve the matrices S, T, Q and Z in the decomposition. If computeQZ==false, some time is saved by not computing matrices Q and Z.

Example:

---

```
MatrixXf A = MatrixXf::Random(4,4);
MatrixXf B = MatrixXf::Random(4,4);
RealQZ<MatrixXf> qz(4); // preallocate space for 4x4 matrices
qz.compute(A,B); // A = Q S Z, B = Q T Z// print original matrices and result of
                  // decomposition
cout << "A:\n" << A << "\n" << "B:\n" << B << "\n";
cout << "S:\n" << qz.matrixS() << "\n" << "T:\n" << qz.matrixT() << "\n";
cout << "Q:\n" << qz.matrixQ() << "\n" << "Z:\n" << qz.matrixZ() << "\n";// verify
                  // precision
cout << "\nErrors:" << "\n|A-QSZ|: "
<< (A-qz.matrixQ()*qz.matrixS()*qz.matrixZ()).norm()
<< ", |B-QTZ|: " << (B-qz.matrixQ()*qz.matrixT()*qz.matrixZ()).norm()
<< "\n|QQ* - I|: " << (qz.matrixQ()*qz.matrixQ().adjoint() -
MatrixXf::Identity(4,4)).norm()
<< ", |ZZ* - I|: " << (qz.matrixZ()*qz.matrixZ().adjoint() -
MatrixXf::Identity(4,4)).norm() << "\n";
```

---

Output:

```
A:
0.68  0.823 -0.444 -0.27
-0.211 -0.605  0.108  0.0268
0.566  -0.33 -0.0452  0.904
0.597  0.536   0.258  0.832

B:
0.271 -0.967 -0.687  0.998
0.435 -0.514 -0.198 -0.563
-0.717 -0.726 -0.74  0.0259
0.214  0.608 -0.782  0.678
```

```

S:
0.927 -0.928  0.643 -0.227
-0.594   0.36  0.146 -0.606
0       0 -0.398 -0.164
0       0       0 -1.12

T:
1.51  0.278 -0.238  0.501
0 -1.04  0.519 -0.239
0       0 -1.25  0.438
0       0       0  0.746

Q:
0.603  0.011  0.552  0.576
-0.142  0.243  0.761 -0.585
0.092 -0.958  0.152 -0.223
0.78   0.149 -0.306 -0.526

Z:
0.284    0.26  -0.696   0.606
-0.918 -0.108  -0.38   0.0406
-0.269    0.783  0.462    0.32
-0.0674 -0.555   0.398   0.727

Errors:
|A-QSZ|: 1.13e-06, |B-QTZ|: 1.81e-06
|QQ* - I|: 1.01e-06, |ZZ* - I|: 7.02e-07

```

Note The implementation is based on the algorithm in [Golub & Van Loan \(1996\)](#), and [Moler & Stewart \(1973\)](#).

#### 25.4.3.1 Member Function Documentation

`RealQZj` `MatrixType & compute ( const MatrixType & A, const MatrixType & B, bool computeQZ = true )`

Computes QZ decomposition of given matrix. Parameters [in] A Matrix A. [in] B Matrix B. [in] computeQZ If false, A and Z are not computed.

Returns Reference to `*this` References `Eigen::NoConvergence`, and `Eigen::Success`. Referenced by `RealQZj` `MatrixType &::RealQZ()`.

`ComputationInfo info () const`

Reports whether previous computation was successful. Returns `Success` if computation was successful, `NoConvergence` otherwise.

`const MatrixType& matrixQ () const`

Returns matrix Q in the QZ decomposition. Returns A const reference to the matrix Q.

`const MatrixType& matrixS () const`

Returns matrix S in the QZ decomposition. Returns A const reference to the matrix S.

`const MatrixType& matrixT () const`

Returns matrix S in the QZ decomposition. Returns A const reference to the matrix S.

`const MatrixType& matrixZ () const`

Returns matrix Z in the QZ decomposition. Returns A const reference to the matrix Z.

RealQZ& **setMaxIterations** ( Index maxIters)

Sets the maximal number of iterations allowed to converge to one eigenvalue or decouple the problem. Referenced by GeneralizedEigenSolver<\_MatrixType>::setMaxIterations().

### 25.4.4 Real Schur Decomposition

Performs a real Schur decomposition of a square matrix. This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the real Schur decomposition; this is expected to be an instantiation of the Matrix class template.

Given a real square matrix  $A$ , this class computes the real Schur decomposition:  $A = UTU^T$  where  $U$  is a real orthogonal matrix and  $T$  is a real quasi-triangular matrix. An orthogonal matrix is a matrix whose inverse is equal to its transpose,  $U^{-1} = U^T$ . A quasi-triangular matrix is a block-triangular matrix whose diagonal consists of 1-by-1 blocks and 2-by-2 blocks with complex eigenvalues. The eigenvalues of the blocks on the diagonal of  $T$  are the same as the eigenvalues of the matrix  $A$ , and thus the real Schur decomposition is used in EigenSolver to compute the eigendecomposition of a matrix.

Call the function compute() to compute the real Schur decomposition of a given matrix. Alternatively, you can use the RealSchur(const MatrixType, bool) constructor which computes the real Schur decomposition at construction time. Once the decomposition is computed, you can use the matrixU() and matrixT() functions to retrieve the matrices  $U$  and  $T$  in the decomposition.

The documentation of RealSchur(const MatrixType, bool) contains an example of the typical use of this class.

See Also

class ComplexSchur, class EigenSolver, class ComplexEigenSolver

Example:

---

```
MatrixXd A = MatrixXd::Random(6,6);
cout << "Here is a random 6x6 matrix, A:" << endl << A << endl << endl;
RealSchur<MatrixXd> schur(A);
cout << "The orthogonal matrix U is:" << endl << schur.matrixU() << endl;
cout << "The quasi-triangular matrix T is:" << endl << schur.matrixT() << endl <<
endl;
MatrixXd U = schur.matrixU();
MatrixXd T = schur.matrixT();
cout << "U * T * U^T = " << endl << U * T * U.transpose() << endl;
```

---

Output:

Here is a random 6x6 matrix, A:

```
0.68  -0.33  -0.27  -0.717  -0.687  0.0259
-0.211  0.536  0.0268  0.214  -0.198  0.678
0.566  -0.444  0.904  -0.967  -0.74   0.225
0.597  0.108  0.832  -0.514  -0.782  -0.408
0.823 -0.0452  0.271  -0.726  0.998  0.275
-0.605  0.258  0.435  0.608  -0.563  0.0486
```

The orthogonal matrix U is:

```
0.348  -0.754  0.00435  -0.351  0.0145  0.432
-0.16   -0.266  -0.747   0.457  -0.366  0.0571
0.505  -0.157  0.0746   0.644  0.518  -0.177
0.703  0.324  -0.409  -0.349  -0.187  -0.275
0.296  0.372   0.24    0.324  -0.379  0.684
-0.126  0.305  -0.46   -0.161  0.647  0.485
```

The quasi-triangular matrix T is:

```

-0.2   -1.83   0.864   0.271   1.09   0.14
0.647   0.298  -0.0536   0.676  -0.288   0.023
0       0       0.967  -0.201  -0.429   0.847
0       0       0       0.353   0.602   0.694
0       0       0       0       0.572  -1.03
0       0       0       0       0.0184  0.664

U * T * U^T =
0.68   -0.33   -0.27  -0.717  -0.687  0.0259
-0.211   0.536   0.0268   0.214  -0.198   0.678
0.566   -0.444   0.904  -0.967  -0.74    0.225
0.597   0.108   0.832  -0.514  -0.782  -0.408
0.823  -0.0452   0.271  -0.726   0.998   0.275
-0.605   0.258   0.435   0.608  -0.563  0.0486

```

#### 25.4.4.1 Member Function Documentation

Member Function Documentation `RealSchur<MatrixType>::compute` ( `const MatrixType & matrix, bool computeU = true` )

Computes Schur decomposition of given matrix.

Parameters

[in] `matrix` Square matrix whose Schur decomposition is to be computed.

[in] `computeU` If true, both `T` and `U` are computed; if false, only `T` is computed.

Returns Reference to `*this`

The Schur decomposition is computed by first reducing the matrix to Hessenberg form using the class `HessenbergDecomposition`. The Hessenberg matrix is then reduced to triangular form by performing Francis QR iterations with implicit double shift. The cost of computing the Schur decomposition depends on the number of iterations; as a rough guide, it may be taken to be flops if `computeU` is true and flops if `computeU` is false.

Example:

---

```

MatrixXf A = MatrixXf::Random(4,4);
RealSchur<MatrixXf> schur(4);
schur.compute(A, /* computeU = */ false);
cout << "The matrix T in the decomposition of A is:" << endl << schur.matrixT() <<
    endl;
schur.compute(A.inverse(), /* computeU = */ false);
cout << "The matrix T in the decomposition of A^(-1) is:" << endl << schur.matrixT() <<
    endl;

```

---

Output:

The matrix T in the decomposition of A is:

```

0.523 -0.698  0.148  0.742
0.475  0.986 -0.793  0.721
0       0       -0.28   -0.77
0       0       0.0145 -0.367

```

The matrix T in the decomposition of A<sup>-1</sup> is:

```

-3.06 -4.57 -6.05  5.39
0.168 -2.62 -3.33  3.86

```

```
0      0 0.434  0.56
0      0 -1.06  1.35
```

See Also `compute(const MatrixType&, bool, Index)` Referenced by `RealSchur`; `MatrixType &::RealSchur()`.

`RealSchur & computeFromHessenberg ( const HessMatrixType & matrixH, const OrthMatrixType & matrixQ, bool computeU )`

Computes Schur decomposition of a Hessenberg matrix  $H = ZTZ^T$ . Parameters

[in] `matrixH` Matrix in Hessenberg form  $H$

[in] `matrixQ` orthogonal matrix  $Q$  that transform a matrix  $A$  to  $H : A = QHQ^T$

`computeU` Computes the matrix  $U$  of the Schur vectors

Returns Reference to `*this` This routine assumes that the matrix is already reduced in Hessenberg form `matrixH` using either the class `HessenbergDecomposition` or another mean. It computes the upper quasi-triangular matrix  $T$  of the Schur decomposition of  $H$ . When `computeU` is true, this routine computes the matrix  $U$  such that  $A = UTU^T = (QZ)T(QZ)^T = QHQ^T$  where  $A$  is the initial matrix. NOTE `Q` is referenced if `computeU` is true; so, if the initial orthogonal matrix is not available, the user should give an identity matrix (`Q.setIdentity()`) See Also `compute(const MatrixType&, bool)`

`ComputationInfo info () const`

Reports whether previous computation was successful. Returns `Success` if computation was successful, `NoConvergence` otherwise.

`const MatrixType& matrixT () const`

Returns the quasi-triangular matrix in the Schur decomposition. Returns `A` const reference to the matrix  $T$ . Precondition Either the constructor `RealSchur(const MatrixType&, bool)` or the member function `compute(const MatrixType&, bool)` has been called before to compute the Schur decomposition of a matrix. See Also `RealSchur(const MatrixType&, bool)` for an example

`const MatrixType& matrixU () const`

Returns the orthogonal matrix in the Schur decomposition. Returns `A` const reference to the matrix  $U$ . Precondition Either the constructor `RealSchur(const MatrixType&, bool)` or the member function `compute(const MatrixType&, bool)` has been called before to compute the Schur decomposition of a matrix, and `computeU` was set to true (the default value). See Also `RealSchur(const MatrixType&, bool)` for an example

`RealSchur & setMaxIterations ( Index maxIters )`

Sets the maximum number of iterations allowed. If not specified by the user, the maximum number of iterations is `m_maxIterationsPerRow` times the size of the matrix. Referenced by `EigenSolver &::MatrixType &::setMaxIterations()`.

Member Data Documentation `const int m_maxIterationsPerRow`

Maximum number of iterations per row. If not otherwise specified, the maximum number of iterations is this number times the size of the matrix. It is currently set to 40.

## 25.4.5 Complex Schur Decomposition

Performs a complex Schur decomposition of a real or complex square matrix.

This is defined in the Eigenvalues module.

MatrixType the type of the matrix of which we are computing the Schur decomposition; this is expected to be an instantiation of the Matrix class template.

Given a real or complex square matrix  $A$ , this class computes the Schur decomposition:  $A = UTU^*$  where  $U$  is a unitary complex matrix, and  $T$  is a complex upper triangular matrix. The diagonal of the matrix  $T$  corresponds to the eigenvalues of the matrix  $A$ .

Call the function `compute()` to compute the Schur decomposition of a given matrix. Alternatively, you can use the `ComplexSchur(const MatrixType, bool)` constructor which computes the Schur decomposition at construction time. Once the decomposition is computed, you can use the `matrixU()` and `matrixT()` functions to retrieve the matrices  $U$  and  $V$  in the decomposition.

See Also

class `RealSchur`, class `EigenSolver`, class `ComplexEigenSolver`

### 25.4.5.1 Member Function Documentation

`ComplexSchur(MatrixType & compute ( const MatrixType & matrix, bool computeU = true )`

Computes Schur decomposition of given matrix. Parameters [in] matrix Square matrix whose Schur decomposition is to be computed. [in] `computeU` If true, both  $T$  and  $U$  are computed; if false, only  $T$  is computed.

Returns Reference to `*this`

The Schur decomposition is computed by first reducing the matrix to Hessenberg form using the class `HessenbergDecomposition`. The Hessenberg matrix is then reduced to triangular form by performing QR iterations with a single shift. The cost of computing the Schur decomposition depends on the number of iterations; as a rough guide, it may be taken on the number of iterations; as a rough guide, it may be taken to be complex flops, or complex flops if `computeU` is false.

Example:

---

```
MatrixXcf A = MatrixXcf::Random(4,4);
ComplexSchur<MatrixXcf> schur(4);
schur.compute(A);
cout << "The matrix T in the decomposition of A is:" << endl << schur.matrixT() <<
    endl;
schur.compute(A.inverse());
cout << "The matrix T in the decomposition of A^(-1) is:" << endl << schur.matrixT()
    << endl;
```

---

Output:

The matrix T in the decomposition of A is:

(-0.691,-1.63)	(0.763,-0.144)	(-0.104,-0.836)	(-0.462,-0.378)
(0,0)	(-0.758,1.22)	(-0.65,-0.772)	(-0.244,0.113)
(0,0)	(0,0)	(0.137,0.505)	(0.0687,-0.404)
(0,0)	(0,0)	(0,0)	(1.52,-0.402)

The matrix T in the decomposition of A<sup>-1</sup> is:

(0.501,-1.84)	(-1.01,-0.984)	(0.636,1.3)	(-0.676,0.352)
(0,0)	(-0.369,-0.593)	(0.0733,0.18)	(-0.0658,-0.0263)
(0,0)	(0,0)	(-0.222,0.521)	(-0.191,0.121)

(0,0)	(0,0)	(0,0)	(0.614,0.162)
-------	-------	-------	---------------

See Also `compute(const MatrixType&, bool, Index)` References `ComplexSchur` `_MatrixType` `z::computeFromHessenberg()`, and `Eigen::Success`. Referenced by `ComplexSchur` `_MatrixType` `z::ComplexSchur()`.

`ComplexSchur& computeFromHessenberg ( const HessMatrixType & matrixH, const OrthMatrixType & matrixQ, bool computeU = true )`

Compute Schur decomposition from a given Hessenberg matrix.

Parameters

[in] `matrixH` Matrix in Hessenberg form  $H$

[in] `matrixQ` orthogonal matrix  $Q$  that transform a matrix  $A$  to  $H$  :  $A = QHQ^T$

`computeU` Computes the matrix  $U$  of the Schur vectors

Returns Reference to `*this`

This routine assumes that the matrix is already reduced in Hessenberg form `matrixH` using either the class `HessenbergDecomposition` or another mean. It computes the upper quasi-triangular matrix `T` of the Schur decomposition of `H`. When `computeU` is true, this routine computes the matrix `U` such that  $A = UTU^T = (QZ)T(QZ)^T = QHQ^T$  where `A` is the initial matrix

NOTE `Q` is referenced if `computeU` is true; so, if the initial orthogonal matrix is not available, the user should give an identity matrix (`Q.setIdentity()`) See Also `compute(const MatrixType&, bool)` Referenced by `ComplexSchur` `_MatrixType` `z::compute()`.

ComputationInfo `info () const`

Reports whether previous computation was successful. Returns `Success` if computation was successful, `NoConvergence` otherwise. Referenced by `ComplexEigenSolver` `_MatrixType` `z::info()`.

`const ComplexMatrixType& matrixT () const`

Returns the triangular matrix in the Schur decomposition.

Returns `A` const reference to the matrix `T`.

It is assumed that either the constructor `ComplexSchur(const MatrixType& matrix, bool computeU)` or the member function `compute(const MatrixType& matrix, bool computeU)` has been called before to compute the Schur decomposition of a matrix.

Note that this function returns a plain square matrix. If you want to reference only the upper triangular part, use: `schur.matrixT().triangularView<Upper>()`

Example:

---

```
MatrixXcf A = MatrixXcf::Random(4,4);
cout << "Here is a random 4x4 matrix, A:"
<< endl << A << endl << endl;ComplexSchur<MatrixXcf> schurOfA(A, false); // false
    means do not compute U
cout << "The triangular matrix T is:"
<< endl << schurOfA.matrixT() << endl;
```

---

Output:

```
Here is a random 4x4 matrix, A:
(-0.211,0.68)  (0.108,-0.444)  (0.435,0.271)  (-0.198,-0.687)
(0.597,0.566)  (0.258,-0.0452)  (0.214,-0.717)  (-0.782,-0.74)
(-0.605,0.823)  (0.0268,-0.27)  (-0.514,-0.967)  (-0.563,0.998)
(0.536,-0.33)   (0.832,0.904)   (0.608,-0.726)  (0.678,0.0259)
```

The triangular matrix T is:

(-0.691,-1.63)	(0.763,-0.144)	(-0.104,-0.836)	(-0.462,-0.378)
(0,0)	(-0.758,1.22)	(-0.65,-0.772)	(-0.244,0.113)
(0,0)	(0,0)	(0.137,0.505)	(0.0687,-0.404)
(0,0)	(0,0)	(0,0)	(1.52,-0.402)

const ComplexMatrixType& **matrixU** ( ) const

Returns the unitary matrix in the Schur decomposition. Returns A const reference to the matrix U. It is assumed that either the constructor ComplexSchur(const MatrixType& matrix, bool computeU) or the member function compute(const MatrixType& matrix, bool computeU) has been called before to compute the Schur decomposition of a matrix, and that computeU was set to true (the default value).

Example:

---

```
MatrixXcf A = MatrixXcf::Random(4,4);
cout << "Here is a random 4x4 matrix, A:" << endl << A << endl << endl;
ComplexSchur<MatrixXcf> schurOfA(A);
cout << "The unitary matrix U is:" << endl << schurOfA.matrixU() << endl;
```

---

Output:

Here is a random 4x4 matrix, A:

(-0.211,0.68)	(0.108,-0.444)	(0.435,0.271)	(-0.198,-0.687)
(0.597,0.566)	(0.258,-0.0452)	(0.214,-0.717)	(-0.782,-0.74)
(-0.605,0.823)	(0.0268,-0.27)	(-0.514,-0.967)	(-0.563,0.998)
(0.536,-0.33)	(0.832,0.904)	(0.608,-0.726)	(0.678,0.0259)

The unitary matrix U is:

(-0.122,0.271)	(0.354,0.255)	(-0.7,0.321)	(0.0909,-0.346)
(0.247,0.23)	(0.435,-0.395)	(0.184,-0.38)	(0.492,-0.347)
(0.859,-0.0877)	(0.00469,0.21)	(-0.256,0.0163)	(0.133,0.355)
(-0.116,0.195)	(-0.484,-0.432)	(-0.183,0.359)	(0.559,0.231)

ComplexSchur& **setMaxIterations** ( Index maxIters)

Sets the maximum number of iterations allowed. If not specified by the user, the maximum number of iterations is m\_maxIterationsPerRow times the size of the matrix. Referenced by ComplexEigenSolver<\_MatrixType>::setMaxIterations().

Member Data Documentationconst int m  
textbf{maxIterationsPerRow}

Maximum number of iterations per row. If not otherwise specified, the maximum number of iterations is this number times the size of the matrix. It is currently set to 30.

## 25.5 Matrix Functions

Matrix functions are defined as follows. Suppose that  $f$  is an entire function (that is, a function on the complex plane that is everywhere complex differentiable). Then its Taylor series

$$f(0) + f'(0)x + \frac{f''(0)}{2}x^2 + \frac{f'''(0)}{3!}x^3 + \dots \quad (25.5.1)$$

converges to  $f(x)$ . In this case, we can define the matrix function by the same series:

$$f(M) = f(0) + f'(0)M + \frac{f''(0)}{2}M^2 + \frac{f'''(0)}{3!}M^3 + \dots \quad (25.5.2)$$

### 25.5.1 Matrix Square Root

---

#### Function **MatSqrt**(*M* As *mpNum*[,]) As *mpNum*

---

The function **MatSqrt** returns an expression representing the matrix square root of the real matrix *M*.

**Parameter:**

*M*: the real matrix of which we are computing the matrix square root.

---

#### Function **cplxMatSqrt**(*M* As *mpNum*[,]) As *mpNum*

---

The function **cplxMatSqrt** returns an expression representing the matrix square root of the complex matrix *M*.

**Parameter:**

*M*: the complex matrix of which we are computing the matrix square root.

Compute the matrix square root.

Parameters

[in] *M* invertible matrix whose square root is to be computed.

Returns: expression representing the matrix square root of *M*.

The matrix square root of *M* is the matrix  $M^{1/2}$  whose square is the original matrix; so if  $S = M^{1/2}$  then  $S^2 = M$ .

In the real case, the matrix *M* should be invertible and it should have no eigenvalues which are real and negative (pairs of complex conjugate eigenvalues are allowed). In that case, the matrix has a square root which is also real, and this is the square root computed by this function.

The matrix square root is computed by first reducing the matrix to quasi-triangular form with the real Schur decomposition. The square root of the quasi-triangular matrix can then be computed directly. The cost is approximately  $25n^3$  real flops for the real Schur decomposition and  $n^3$  real flops for the remainder (though the computation time in practice is likely more than this indicates).

Details of the algorithm can be found in [Higham \(1987\)](#).

If the matrix is positive-definite symmetric, then the square root is also positive-definite symmetric. In this case, it is best to use `SelfAdjointEigenSolver::operatorSqrt()` to compute it.

In the complex case, the matrix *M* should be invertible; this is a restriction of the algorithm.

The square root computed by this algorithm is the one whose eigenvalues have an argument in the interval  $(-\frac{1}{2}\pi, \frac{1}{2}\pi]$ . This is the usual branch cut.

The computation is the same as in the real case, except that the complex Schur decomposition is used to reduce the matrix to a triangular matrix. The theoretical cost is the same. Details are in [Björck & Hammarling \(1983\)](#).

Example: The following program checks that the square root of

$$\begin{pmatrix} \cos\left(\frac{1}{3}\pi\right) & -\sin\left(\frac{1}{3}\pi\right) \\ \sin\left(\frac{1}{3}\pi\right) & \cos\left(\frac{1}{3}\pi\right) \end{pmatrix} \quad (25.5.3)$$

corresponding to a rotation over 60 degrees, is a rotation over 30 degrees:

$$\begin{pmatrix} \cos\left(\frac{1}{6}\pi\right) & -\sin\left(\frac{1}{6}\pi\right) \\ \sin\left(\frac{1}{6}\pi\right) & \cos\left(\frac{1}{6}\pi\right) \end{pmatrix} \quad (25.5.4)$$


---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>

using namespace Eigen;

int main()
{
const double pi = std::acos(-1.0);

MatrixXd A(2,2);
A << cos(pi/3), -sin(pi/3),
sin(pi/3), cos(pi/3);
std::cout << "The matrix A is:\n" << A << "\n\n";
std::cout << "The matrix square root of A is:\n" << A.sqrt() << "\n\n";
std::cout << "The square of the last matrix is:\n"
<< A.sqrt() * A.sqrt() << "\n";
}
```

---

Output:

The matrix A is:

```
0.5 -0.866025
0.866025      0.5
```

The matrix square root of A is:

```
0.866025      -0.5
0.5 0.866025
```

The square of the last matrix is:

```
0.5 -0.866025
0.866025      0.5
```

## 25.5.2 Matrix Exponential

---

### Function **MatExp**(*M* As *mpNum*[,]) As *mpNum*

---

The function **MatExp** returns an expression representing the matrix exponential of the real matrix *M*.

**Parameter:**

*M*: the real matrix of which we are computing the matrix exponential.

---

### Function **cplxMatExp**(*M* As *mpNum*[,]) As *mpNum*

---

The function **cplxMatExp** returns an expression representing the matrix exponential of the complex matrix *M*.

**Parameter:**

*M*: the complex matrix of which we are computing the matrix exponential.

Compute the matrix exponential.

Parameters: [in] *M* matrix whose exponential is to be computed.

Returns: expression representing the matrix exponential of *M*.

The matrix exponential of *M* is defined by

$$\exp(M) = \sum_{k=0}^{\infty} \frac{M^k}{k!}. \quad (25.5.5)$$

The matrix exponential can be used to solve linear ordinary differential equations: the solution of  $y' = My$  with the initial condition  $y(0) = y_0$  is given by  $y(t) = \exp(M)y_0$ . The cost of the computation is approximately  $20n^3$  for matrices of size *n*. The number 20 depends weakly on the norm of the matrix.

The matrix exponential is computed using the scaling-and-squaring method combined with Padé approximation. The matrix is first rescaled, then the exponential of the reduced matrix is computed approximant, and then the rescaling is undone by repeated squaring. The degree of the Padé approximant is chosen such that the approximation error is less than the round-off error. However, errors may accumulate during the squaring phase.

Details of the algorithm can be found in [Higham \(2005\)](#).

Example: The following program checks that

$$\exp \begin{pmatrix} 0 & \frac{1}{4}\pi & 0 \\ -\frac{1}{4}\pi & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} & 0 \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (25.5.6)$$

This corresponds to a rotation of  $\frac{1}{4}\pi$  radians around the z-axis.

---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>

using namespace Eigen;

int main()
{
    const double pi = std::acos(-1.0);
```

```

MatrixXd A(3,3);
A << 0, -pi/4, 0,
pi/4, 0, 0,
0, 0, 0;
std::cout << "The matrix A is:\n" << A << "\n\n";
std::cout << "The matrix exponential of A is:\n"
<< A.exp() << "\n\n";
}

```

---

Output:

The matrix A is:

```

0 -0.785398      0
0.785398      0      0
0      0      0

```

The matrix exponential of A is:

```

0.707107 -0.707107      0
0.707107  0.707107      0
0      0      1

```

Note: M has to be a matrix of real or complex.

### 25.5.3 Matrix Logarithm

---

#### Function **MatLog**(*M* As *mpNum*[,]) As *mpNum*

---

The function **MatLog** returns an expression representing the matrix logarithm of the real matrix M.

**Parameter:**

*M*: the real matrix of which we are computing the matrix logarithm.

---

#### Function **cplxMatLog**(*M* As *mpNum*[,]) As *mpNum*

---

The function **cplxMatLog** returns an expression representing the matrix logarithm of the complex matrix M.

**Parameter:**

*M*: the complex matrix of which we are computing the matrix logarithm.

Compute the matrix logarithm.

Parameters: [in] M invertible matrix whose logarithm is to be computed.

Returns: expression representing the matrix logarithm root of M.

The matrix logarithm of *M* is a matrix *X* such that  $\exp(X) = M$  where  $\exp$  denotes the matrix exponential. As for the scalar logarithm, the equation  $\exp(X) = M$  may have multiple solutions; this function returns a matrix whose eigenvalues have imaginary part in the interval  $(-\pi, \pi]$ .

In the real case, the matrix *M* should be invertible and it should have no eigenvalues which are real and negative (pairs of complex conjugate eigenvalues are allowed). In the complex case, it only needs to be invertible.

This function computes the matrix logarithm using the Schur-Parlett algorithm as implemented by `MatrixBase::matrixFunction()`. The logarithm of an atomic block is computed by `MatrixLogarithmAtomic`, which uses direct computation for 1-by-1 and 2-by-2 blocks and an inverse scaling-and-squaring algorithm for bigger blocks, with the square roots computed by `MatrixBase::sqrt()`. Details of the algorithm can be found in Section 11.6.2 of [Higham \(2008\)](#).

Example: The following program checks that

$$\log \begin{pmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} & 0 \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{4}\pi & 0 \\ -\frac{1}{4}\pi & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad (25.5.7)$$

This corresponds to a rotation of  $\frac{1}{4}\pi$  radians around the z-axis. This is the inverse of the example used in the documentation of `exp()`.

---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>

using namespace Eigen;

int main()
{
using std::sqrt;
MatrixXd A(3,3);
A << 0.5*sqrt(2), -0.5*sqrt(2), 0,
0.5*sqrt(2), 0.5*sqrt(2), 0,
0, 0, 1;
std::cout << "The matrix A is:\n" << A << "\n\n";
std::cout << "The matrix logarithm of A is:\n" << A.log() << "\n";
}
```

---

Output:

The matrix A is:

```
0.707107 -0.707107 0
0.707107 0.707107 0
0 0 1
```

The matrix logarithm of A is:

```
-1.11022e-16 -0.785398 0
0.785398 -1.11022e-16 0
0 0 0
```

### 25.5.4 Matrix raised to arbitrary real power

---

#### Function **MatPow**(*M* As *mpNum*[*..*], *p* As *mpNum*) As *mpNum*

---

The function **MatPow** returns an expression representing the matrix power of the real matrix *M*.

**Parameters:**

*M*: *M* base of the matrix power, should be a square matrix.

*p*: exponent of the matrix power, should be real.

---

#### Function **cplxMatPow**(*M* As *mpNum*[*..*], *p* As *mpNum*) As *mpNum*

---

The function **cplxMatPow** returns an expression representing the matrix power of the complex matrix *M*.

**Parameters:**

*M*: *M* base of the matrix power, should be a square matrix.

*p*: exponent of the matrix power, should be real.

MatrixBase::pow()

Compute the matrix raised to arbitrary real power. const MatrixPowerReturnValue; MatrixBase::pow(RealScalar *p*) constParameters [in] *M* base of the matrix power, should be a square matrix. [in] *p* exponent of the matrix power, should be real.

The matrix power  $M^p$  is defined as  $\exp(p \log(M))$ , where  $\exp$  denotes the matrix exponential, and  $\log$  denotes the matrix logarithm.

The matrix *M* should meet the conditions to be an argument of matrix logarithm. If *p* is not of the real scalar type of *M*, it is casted into the real scalar type of *M*.

This function computes the matrix power using the Schur-Padé algorithm as implemented by class **MatrixPower**. The exponent is split into integral part and fractional part, where the fractional part is in the interval  $(-1, 1)$ . The main diagonal and the first super-diagonal is directly computed.

Details of the algorithm can be found in [Higham & Lin \(2011\)](#).

Example: The following program checks that

$$\begin{pmatrix} \cos(1) & -\sin(1) & 0 \\ \sin(1) & \cos(1) & 0 \\ 0 & 0 & 1 \end{pmatrix}^{\frac{1}{4}\pi} = \begin{pmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} & 0 \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (25.5.8)$$

This corresponds to  $\frac{1}{4}\pi$  rotations of 1 radian around the z-axis.

---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>

using namespace Eigen;

int main()
{
    const double pi = std::acos(-1.0);
    Matrix3d A;
    A << cos(1), -sin(1), 0,
       sin(1), cos(1), 0,
       0, 0, 1;
    std::cout << "The matrix A is:\n" << A << "\n\n"
```

---

```
"The matrix power A^(pi/4) is:\n" << A.pow(pi/4) << std::endl;
return 0;
}
```

---

Output:

The matrix A is:

```
0.540302 -0.841471      0
0.841471  0.540302      0
0          0              1
```

The matrix power A^(pi/4) is:

```
0.707107 -0.707107      0
0.707107  0.707107      0
0          0              1
```

MatrixBase::pow() is user-friendly. However, there are some circumstances under which you should use class MatrixPower directly. MatrixPower can save the result of Schur decomposition, so it's better for computing various powers for the same matrix. Example:

---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>

using namespace Eigen;

int main()
{
Matrix4cd A = Matrix4cd::Random();
MatrixPower<Matrix4cd> Apow(A);

std::cout << "The matrix A is:\n" << A << "\n\n"
"A^3.1 is:\n" << Apow(3.1) << "\n\n"
"A^3.3 is:\n" << Apow(3.3) << "\n\n"
"A^3.7 is:\n" << Apow(3.7) << "\n\n"
"A^3.9 is:\n" << Apow(3.9) << std::endl;
return 0;
}
```

---

Output:

The matrix A is:

```
(-0.211234,0.680375)  (0.10794,-0.444451)  (0.434594,0.271423) (-0.198111,-0.686642)
(0.59688,0.566198)   (0.257742,-0.0452059) (0.213938,-0.716795) (-0.782382,-0.740419)
(-0.604897,0.823295) (0.0268018,-0.270431) (-0.514226,-0.967399) (-0.563486,0.997849)
(0.536459,-0.329554) (0.83239,0.904459)   (0.608354,-0.725537) (0.678224,0.0258648)
```

A^3.1 is:

```
(2.80575,-0.607662) (-1.16847,-0.00660555) (-0.760385,1.01461)  (-0.38073,-0.106512)
(1.4041,-3.61891)    (1.00481,0.186263)    (-0.163888,0.449419)  (-0.388981,-1.22629)
(-2.07957,-1.58136)  (0.825866,2.25962)    (5.09383,0.155736)   (0.394308,-1.63034)
```

$$(-0.818997, 0.671026) \quad (2.11069, -0.00768024) \quad (-1.37876, 0.140165) \quad (2.50512, -0.854429)$$

$A^3.3$  is:

$$\begin{aligned} (2.83571, -0.238717) & \quad (-1.48174, -0.0615217) \quad (-0.0544396, 1.68092) \quad (-0.292699, -0.621726) \\ (2.0521, -3.58316) & \quad (0.87894, 0.400548) \quad (0.738072, -0.121242) \quad (-1.07957, -1.63492) \\ (-3.00106, -1.10558) & \quad (1.52205, 1.92407) \quad (5.29759, -1.83562) \quad (-0.532038, -1.50253) \\ (-0.491353, -0.4145) & \quad (2.5761, 0.481286) \quad (-1.21994, 0.0367069) \quad (2.67112, -1.06331) \end{aligned}$$

$A^3.7$  is:

$$\begin{aligned} (1.42126, 0.33362) & \quad (-1.39486, -0.560486) \quad (1.44968, 2.47066) \quad (-0.324079, -1.75879) \\ (2.65301, -1.82427) & \quad (0.357333, -0.192429) \quad (2.01017, -1.4791) \quad (-2.71518, -2.35892) \\ (-3.98544, 0.964861) & \quad (2.26033, 0.554254) \quad (3.18211, -5.94352) \quad (-2.22888, 0.128951) \\ (0.944969, -2.14683) & \quad (3.31345, 1.66075) \quad (-0.0623743, -0.848324) \quad (2.3897, -1.863) \end{aligned}$$

$A^3.9$  is:

$$\begin{aligned} (0.0720766, 0.378685) & \quad (-0.931961, -0.978624) \quad (1.9855, 2.34105) \quad (-0.530547, -2.17664) \\ (2.40934, -0.265286) & \quad (0.0299975, -1.08827) \quad (1.98974, -2.05886) \quad (-3.45767, -2.50235) \\ (-3.71666, 2.3874) & \quad (2.054, -0.303) \quad (0.844348, -7.29588) \quad (-2.59136, 1.57689) \\ (1.87645, -2.38798) & \quad (3.52111, 2.10508) \quad (0.799055, -1.6122) \quad (1.93452, -2.44408) \end{aligned}$$

### 25.5.5 Matrix General Function

---

Function **MatGeneralFunction**(*M* As *mpNum*[,], *f* As *mpFunction*) As *mpNum*

---

The function **MatGeneralFunction** returns an expression representing *f* applied to the real matrix *M*.

**Parameters:**

*M*: argument of matrix function, should be a square matrix.

*f*: *f* an entire function; *f*(*x*,*n*) should compute the *n*-th derivative of *f* at *x*.

---

Function **cplxMatGeneralFunction**(*M* As *mpNum*[,], *f* As *mpFunction*) As *mpNum*

---

The function **cplxMatGeneralFunction** returns an expression representing *f* applied to the complex matrix *M*.

**Parameters:**

*M*: argument of matrix function, should be a square matrix.

*f*: *f* an entire function; *f*(*x*,*n*) should compute the *n*-th derivative of *f* at *x*.

Compute a matrix function.

Parameters

[in] *M* argument of matrix function, should be a square matrix.

[in] *f* an entire function; *f*(*x*,*n*) should compute the *n*-th derivative of *f* at *x*.

Returns expression representing *f* applied to *M*.

Suppose that *M* is a matrix whose entries have type *Scalar*. Then, the second argument, *f*, should be a function with prototype

ComplexScalar *f*(ComplexScalar, int)

where *ComplexScalar* = `std::complex` if *Scalar* is real (e.g., `float` or `double`) and *ComplexScalar* = *Scalar* if *Scalar* is complex.

The return value of *f*(*x*,*n*) should be  $f^{(n)}(x)$ , the *n*-th derivative of *f* at *x*.

This routine uses the algorithm described in [Davies & Higham \(2003\)](#).

The actual work is done by the *MatrixFunction* class. Example: The following program checks that

$$\exp \begin{pmatrix} 0 & \frac{1}{4}\pi & 0 \\ -\frac{1}{4}\pi & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} & 0 \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (25.5.9)$$

This corresponds to a rotation of  $\frac{1}{4}\pi$  radians around the *z*-axis. This is the same example as used in the documentation of *exp()*.

---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>

using namespace Eigen;

std::complex<double> expfn(std::complex<double> x, int)
{
    return std::exp(x);
}
```

```

int main()
{
const double pi = std::acos(-1.0);

MatrixXd A(3,3);
A << 0, -pi/4, 0,
pi/4, 0, 0,
0, 0, 0;

std::cout << "The matrix A is:\n" << A << "\n\n";
std::cout << "The matrix exponential of A is:\n"
<< A.matrixFunction(expfn) << "\n\n";
}

```

---

Output:

The matrix A is:

```

0 -0.785398 0
0.785398 0 0
0 0 0

```

The matrix exponential of A is:

```

0.707107 -0.707107 0
0.707107 0.707107 0
0 0 1

```

Note that the function expfn is defined for complex numbers x, even though the matrix A is over the reals. Instead of expfn, we could also have used StdStemFunctions::exp:  
A.matrixFunction(StdStemFunctions::complex &exp, &B);

### 25.5.6 Matrix Sine

---

Function **MatSin(*M* As mpNum[,]) As mpNum**

---

The function MatSin returns an expression representing the matrix sine of the real matrix M.

**Parameter:**

*M*: the real matrix of which we are computing the matrix sine.

Function **cplxMatSin(*M* As mpNum[,]) As mpNum**

---

The function cplxMatSin returns an expression representing the matrix sine of the complex matrix M.

**Parameter:**

*M*: the complex matrix of which we are computing the matrix sine.

Compute the matrix sine.

Parameters: [in] M a square matrix.

Returns: expression representing  $\sin(M)$ .

This function calls matrixFunction() with StdStemFunctions::sin().  
 Example:

---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>

using namespace Eigen;

int main()
{
MatrixXd A = MatrixXd::Random(3,3);
std::cout << "A = \n" << A << "\n\n";

MatrixXd sinA = A.sin();
std::cout << "sin(A) = \n" << sinA << "\n\n";

MatrixXd cosA = A.cos();
std::cout << "cos(A) = \n" << cosA << "\n\n";

// The matrix functions satisfy sin^2(A) + cos^2(A) = I,
// like the scalar functions.
std::cout << "sin^2(A) + cos^2(A) = \n" << sinA*sinA + cosA*cosA << "\n\n";
}
```

---

Output:

```
A =
0.680375  0.59688 -0.329554
-0.211234  0.823295  0.536459
0.566198 -0.604897 -0.444451

sin(A) =
0.679919  0.4579 -0.400612
-0.227278  0.821913  0.5358
0.570141 -0.676728 -0.462398

cos(A) =
0.927728 -0.530361 -0.110482
0.00969246  0.889022 -0.137604
-0.132574 -0.04289  1.16475

sin^2(A) + cos^2(A) =
1  4.44089e-16  1.94289e-16
6.38378e-16           1  5.55112e-16
0 -6.10623e-16           1
```

### 25.5.7 Matrix Cosine

---

Function **MatCos**(*M* As *mpNum*[,]) As *mpNum*

---

The function **MatCos** returns an expression representing the matrix cosine of the real matrix *M*.

**Parameter:**

*M*: the real matrix of which we are computing the matrix cosine.

---

Function **cplxMatCos**(*M* As *mpNum*[,]) As *mpNum*

---

The function **cplxMatCos** returns an expression representing the matrix cosine of the complex matrix *M*.

**Parameter:**

*M*: the complex matrix of which we are computing the matrix cosine.

Compute the matrix cosine.

Parameters: [in] *M* a square matrix.

Returns expression representing  $\cos(M)$ .

This function calls **matrixFunction()** with **StdStemFunctions::cos()**.

See Also **sin()** for an example.

### 25.5.8 Matrix Hyperbolic Sine

---

Function **MatSinh**(*M* As *mpNum*[,]) As *mpNum*

---

The function **MatSinh** returns an expression representing the matrix hyperbolic sine of the real matrix *M*.

**Parameter:**

*M*: the real matrix of which we are computing the matrix hyperbolic sine.

---

Function **cplxMatSinh**(*M* As *mpNum*[,]) As *mpNum*

---

The function **cplxMatSinh** returns an expression representing the matrix hyperbolic sine of the complex matrix *M*.

**Parameter:**

*M*: the complex matrix of which we are computing the matrix hyperbolic sine.

Compute the matrix hyperbolic sine.

Parameters: [in] *M* a square matrix.

Returns: expression representing  $\sinh(M)$ .

This function calls **matrixFunction()** with **StdStemFunctions::sinh()**.

Example:

---

```
#include <unsupported/Eigen/MatrixFunctions>
#include <iostream>
```

```
using namespace Eigen;
```

```

int main()
{
MatrixXf A = MatrixXf::Random(3,3);
std::cout << "A = \n" << A << "\n\n";

MatrixXf sinhA = A.sinh();
std::cout << "sinh(A) = \n" << sinhA << "\n\n";

MatrixXf coshA = A.cosh();
std::cout << "cosh(A) = \n" << coshA << "\n\n";

// The matrix functions satisfy cosh^2(A) - sinh^2(A) = I,
// like the scalar functions.
std::cout << "cosh^2(A) - sinh^2(A) = \n" << coshA*coshA - sinhA*sinhA
<< "\n\n";
}

```

---

Output:

```

A =
0.680375  0.59688 -0.329554
-0.211234  0.823295  0.536459
0.566198 -0.604897 -0.444451

sinh(A) =
0.682534  0.739989 -0.256871
-0.194928  0.826512  0.537546
0.562584 -0.53163 -0.425199

cosh(A) =
1.07817   0.567068   0.132125
-0.00418614  1.11649   0.135361
0.128891  0.0659989  0.851201

cosh^2(A) - sinh^2(A) =
1           0   8.9407e-08
1.29454e-07           1 -2.98023e-08
0 -2.83122e-07           1

```

### 25.5.9 Matrix Hyperbolic Cosine

---

Function **MatCosh(*M* As *mpNum*[,]) As *mpNum***

---

The function MatCosh returns an expression representing the matrix hyperbolic cosine of the real matrix M.

**Parameter:**

*M*: the real matrix of which we are computing the matrix hyperbolic cosine.

---

**Function `cplxMatCosh(M As mpNum[,]) As mpNum`**

---

The function `cplxMatCosh` returns an expression representing the matrix hyperbolic cosine of the complex matrix  $M$ .

**Parameter:**

$M$ : the complex matrix of which we are computing the matrix hyperbolic cosine.

Compute the matrix hyperbolic cosine.

Parameters: [in]  $M$  a square matrix.

Returns expression representing  $\cosh(M)$ .

This function calls `matrixFunction()` with `StdStemFunctions::cosh()`.

See Also `sinh()` for an example.

# Chapter 26

## Polynomials (based on Eigen)

### 26.1 Polynomial Evaluation

The functions described here evaluate the polynomial  $c_0 + c_1x + c_2x^2 + \dots + c_{n-1}x^{n-1}$  using Horner's method for stability.

#### 26.1.1 Polynomial Evaluation, Real Coefficients and Argument

---

Function **PolynomialEvaluation**(*x* As mpNum, *c* As mpNum $\langle\langle\rangle\rangle$ ) As mpNum

---

The function **PolynomialEvaluation** returns the value of a polynomial for the real variable *x* with real coefficients *c*.

**Parameters:**

*x*: A real number.

*c*: A vector of real coefficients.

#### 26.1.2 Polynomial Evaluation, Complex Coefficients and Argument

---

Function **cplxPolynomialEvaluation**(*z* As mpNum, *c* As mpNum $\langle\langle\rangle\rangle$ ) As mpNum

---

The function **cplxPolynomialEvaluation** returns the value of a polynomial for the complex variable *z* with complex coefficients *c*.

**Parameters:**

*z*: A complex number.

*c*: A vector of complex coefficients.

#### 26.1.3 Examples

---

```
Sub DemoPolyComplexEvalComplex()
Dim c() As mp_complex, n As Long
Dim x As mp_complex, y As mp_complex
n = 4
x.Real = 3.54: x.Imag = 2.66
ReDim c(0 To n - 1)
c(0).Real = 2: c(1).Real = 5: c(2).Real = 4: c(3).Real = 7
```

```

c(0).Imag = 2: c(1).Imag = 5: c(2).Imag = 4: c(3).Imag = 7
y = mp_complex_poly_complex_eval(c(0), n, x)
Debug.Print "x: ", x.Real, x.Imag, "y:", y.Real, y.Imag
End Sub

```

---

The output of the program is,

```

x:      3.54 + 2.66i
y:    -830.84176 + 482.955648i

```

## 26.2 Quadratic Equations

### 26.2.1 Quadratic Equation, Real Coefficients and Zeros

---

Function **QuadraticEquation**(*a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*) As *mpNum*[]

---

The function **QuadraticEquation** returns a real vector containing the real roots of the quadratic equation.

**Parameters:**

- a*: A real number.
- b*: A real number.
- c*: A real number.

This function returns a real vector containing the real roots of the quadratic equation

$$a + bx + cx^2 = 0 \quad (26.2.1)$$

where the coefficients *a*, *b*, *c* are all real.

The roots are returned in ascending order. If no real roots exist, then the function returns NaN. 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 mp\_real 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.

### 26.2.2 Quadratic Equation, Complex Coefficients and Zeros

---

Function **cplxQuadraticEquation**(*a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*) As *mpNum*[]

---

The function **cplxQuadraticEquation** returns a complex vector containing the complex roots of the quadratic equation.

**Parameters:**

- a*: A real or complex number.
- b*: A real or complex number.
- c*: A real or complex number.

This function returns a complex vector containing the complex roots of the quadratic equation

$$a + bz + cz^2 = 0, \quad (26.2.2)$$

where the coefficients  $a, b, c$  can be either real or complex.

The roots are returned in ascending order, sorted first by their real components and then by their imaginary components.

## 26.3 Cubic Equations

### 26.3.1 Cubic Equation, Real Coefficients and Zeros

---

Function **CubicEquation**(*a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *d* As *mpNum*) As *mpNum*[]

---

The function **CubicEquation** returns a real vector containing the real roots of the cubic equation.

**Parameters:**

- a*: A real number.
- b*: A real number.
- c*: A real number.
- d*: A real number.

This function returns a real vector containing the real roots of the cubic equation

$$a + bx + cx^2 + dx^3 = 0 \quad (26.3.1)$$

where the coefficients  $a, b, c, d$  are all real.

The roots (either one or three) are returned 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.

### 26.3.2 Cubic Equation, Complex Coefficients and Zeros

---

Function **cplxCubicEquation**(*a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *d* As *mpNum*) As *mpNum*[]

---

The function **cplxCubicEquation** returns a complex vector containing the complex roots of the cubic equation.

**Parameters:**

- a*: A real or complex number.
- b*: A real or complex number.
- c*: A real or complex number.
- d*: A real or complex number.

This function returns a complex vector containing the complex roots of the cubic equation

$$a + bz + cz^2 + dz^3 = 0, \quad (26.3.2)$$

where the coefficients  $a, b, c, d$  can be either real or complex.

The roots are returned in ascending order, sorted first by their real components and then by their imaginary components.

## 26.4 Quartic Equations

### 26.4.1 Quartic Equation, Real Coefficients and Zeros

---

Function **QuarticEquation**(*a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *d* As *mpNum*, *e* As *mpNum*) As *mpNum*[]

---

The function **QuarticEquation** returns a real vector containing the real roots of the quartic equation.

**Parameters:**

- a*: A real number.
- b*: A real number.
- c*: A real number.
- d*: A real number.
- e*: A real number.

This function returns a real vector containing the real roots of the quartic equation

$$a + bx + cx^2 + dx^3 + ex^4 = 0 \quad (26.4.1)$$

where the coefficients *a*, *b*, *c*, *d*, *e* are all real.

The roots are returned in ascending order. If no real roots exist, then the function returns NaN.

### 26.4.2 Quartic Equation, Complex Coefficients and Zeros

---

Function **cplxQuarticEquation**(*a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *d* As *mpNum*, *e* As *mpNum*) As *mpNum*[]

---

The function **cplxQuarticEquation** returns a complex vector containing the complex roots of the quartic equation.

**Parameters:**

- a*: A real or complex number.
- b*: A real or complex number.
- c*: A real or complex number.
- d*: A real or complex number.
- e*: A real or complex number.

This function returns a complex vector containing the complex roots of the quartic equation

$$a + bz + cz^2 + dz^3 + ez^4 = 0 \quad (26.4.2)$$

where the coefficients *a*, *b*, *c*, *d*, *e* can be either real or complex.

The roots are returned in ascending order, sorted first by their real components and then by their imaginary components.

## 26.5 General Polynomial Equations

### 26.5.1 General Polynomial Equation, Real Coefficients and Zeros

---

Function **GeneralPolynomialEquation**(*a* As *mpNum*[]) As *mpNum*[]

---

The function `GeneralPolynomialEquation` returns a real vector containing the real roots of the general real polynomial.

**Parameter:**

*a*: The real coefficients of the polynomial.

This function computes the real roots of the general real polynomial

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \quad (26.5.1)$$

using balanced-QR reduction of the companion matrix (see [Edelman & Murakami \(1995\)](#)). The coefficient of the highest order term must be non-zero. The roots (if any) are returned as real vector.

## 26.5.2 General Polynomial Equation, Complex Coefficients and Zeros

---

### Function `cplxGeneralPolynomialEquation(c As mpNum[])` As mpNum[]

---

The function `cplxGeneralPolynomialEquation` returns a complex vector containing the complex roots of the general complex polynomial.

**Parameter:**

*c*: The complex coefficients of the polynomial.

This function computes the complex roots of the general complex polynomial

$$P(z) = c_0 + c_1z + c_2z^2 + \dots + c_{n-1}z^{n-1} \quad (26.5.2)$$

using balanced-QR reduction of the companion matrix (see [Edelman & Murakami \(1995\)](#)). The coefficient of the highest order term must be non-zero. The  $n - 1$  roots are returned as a complex vector. The function returns `mp_SUCCESS` if all the roots are found. If the QR reduction does not converge, the error handler is invoked with an error code of `mp_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. [Zeng \(2004, 2005\)](#))

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

$$1, e^{2\pi i/5}, e^{4\pi i/5}, e^{6\pi i/5}, e^{8\pi i/5}. \quad (26.5.3)$$

---

```

Sub DemoComplexSolve()
Dim a() As mp_real, z() As mp_complex, w As mp_poly_complex_workspace
Dim n As Long, i As Long, status As Long
n = 6
ReDim a(0 To n - 1)
ReDim z(0 To n - 2)
a(0) = -1: a(1) = 0: a(2) = 0: a(3) = 0: a(4) = 0: a(5) = 1
w = mp_poly_complex_workspace_alloc(n)
status = mp_poly_complex_solve(a(), n, w, z())
Call mp_poly_complex_workspace_free(w)
For i = 0 To n - 2
Debug.Print z(i).Real, z(i).Imag

```

```
Next i  
End Sub
```

---

The output of the program is

```
z0 = -0.809016994374947451 +0.587785252292473137  
z1 = -0.809016994374947451 -0.587785252292473137  
z2 = +0.309016994374947451 +0.951056516295153642  
z3 = +0.309016994374947451 -0.951056516295153642  
z4 = +1.0000000000000000 +0.0000000000000000
```

which agrees with the analytic result,  $z_n = e^{2n\pi i/5}$ .

# Chapter 27

## Fast Fourier Transform (based on Eigen)

### 27.1 Discrete Fourier Transforms

In this section, we provide precise mathematical definitions for the transforms that FFTW computes. These transform definitions are fairly standard, but some authors follow slightly different conventions for the normalization of the transform (the constant factor in front) and the sign of the complex exponent. We begin by presenting the one-dimensional (1d) transform definitions, and then give the straightforward extension to multi-dimensional transforms.

A good introduction is given in [Arndt \(2011\)](#), chapter 21.

Another Reference is [Kammler \(2008\)](#).

Based on the EIGEN implementation of KISSFFT.

#### 27.1.1 1d Complex Discrete Fourier Transform (DFT)

---

##### Function **FFTW\_FORWARD**(*X* As *mpNum[]*) As *mpNum[]*

---

The function FFTW\_FORWARD returns a complex vector containing the forward complex discrete Fourier transform of *X*.

**Parameter:**

*X*: A complex vector.

The forward (FFTW\_FORWARD) discrete Fourier transform (DFT) of a 1d complex array *X* of size *n* computes an array *Y*, where:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi j k \sqrt{-1}/n}. \quad (27.1.1)$$

##### Function **FFTW\_BACKWARD**(*X* As *mpNum[]*) As *mpNum[]*

---

The function FFTW\_BACKWARD returns a complex vector containing the backward complex discrete Fourier transform of *X*.

**Parameter:**

*X*: A complex vector.

The backward (FFTW\_BACKWARD) DFT computes:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{2\pi j k \sqrt{-1}/n}. \quad (27.1.2)$$

FFTW computes an unnormalized transform, in that there is no coefficient in front of the summation in the DFT. In other words, applying the forward and then the backward transform will multiply the input by  $n$ .

From above, an `FFTW_FORWARD` transform corresponds to a sign of  $-1$  in the exponent of the DFT. Note also that we use the standard “in-order” output ordering – the  $k$ -th output corresponds to the frequency  $k/n$  (or  $k/T$ , where  $T$  is your total sampling period).

For those who like to think in terms of positive and negative frequencies, this means that the positive frequencies are stored in the first half of the output and the negative frequencies are stored in backwards order in the second half of the output. (The frequency  $-k/n$  is the same as the frequency  $(n - k)/n$ .)

### 27.1.2 1d Real-data DFT

---

#### Function `FFTW_R2C(X As mpNum[])` As `mpNum[]`

---

The function `FFTW_R2C` returns a complex vector containing the forward complex discrete Fourier transform of  $X$ .

**Parameter:**

$X$ : A real vector.

The real-input (r2c) DFT in FFTW computes the *forward* transform  $Y$  of the size  $n$  real array  $X$ , exactly as defined above, i.e.

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi j k \sqrt{-1}/n}. \quad (27.1.3)$$

This output array  $Y$  can easily be shown to possess the “Hermitian” symmetry  $Y_k = Y_{n-k}^*$ , where we take  $Y$  to be periodic so that  $Y_n = Y_0$ . As a result of this symmetry, half of the output  $Y$  is redundant (being the complex conjugate of the other half), and so the 1d r2c transforms only output elements  $0 \dots n/2$  of  $Y$  ( $n/2 + 1$  complex numbers), where the division by 2 is rounded down. Moreover, the Hermitian symmetry implies that  $Y_0$  and, if  $n$  is even, the  $Y_{n/2}$  element, are purely real. So, for the R2HC r2r transform, these elements are not stored in the halfcomplex output format.

---

#### Function `FFTW_C2R(X As mpNum[])` As `mpNum[]`

---

The function `FFTW_C2R` returns a real vector containing the backward discrete Fourier transform of  $X$ .

**Parameter:**

$X$ : A complex hermitian vector.

The c2r and H2RC r2r transforms compute the backward DFT of the *complex* array  $X$  with Hermitian symmetry, stored in the r2c/R2HC output formats, respectively, where the backward transform is defined exactly as for the complex case:

$$Y_k = \sum_{j=0}^{n-1} X_j e^{2\pi j k \sqrt{-1}/n}. \quad (27.1.4)$$

The outputs  $Y$  of this transform can easily be seen to be purely real, and are stored as an array of real numbers. Like FFTW’s complex DFT, these transforms are unnormalized. In other words, applying the real-to-complex (forward) and then the complex-to-real (backward) transform will multiply the input by  $n$ .

### 27.1.3 1d Real-even DFTs (DCTs)

The Real-even symmetry DFTs in FFTW are exactly equivalent to the unnormalized forward (and backward) DFTs as defined above, where the input array  $X$  of length  $N$  is purely real and is also even symmetry. In this case, the output array is likewise real and even symmetry.

For the case of REDFT00, this even symmetry means that  $X_j = X_{N-j}$ , where we take  $X$  to be periodic so that  $X_N = X_0$ . Because of this redundancy, only the first  $n$  real numbers are actually stored, where  $N = 2(n - 1)$ .

The proper definition of even symmetry for REDFT10, REDFT01, and REDFT11 transforms is somewhat more intricate because of the shifts by 1/2 of the input and/or output. Because of the even symmetry, however, the sine terms in the DFT all cancel and the remaining cosine terms are written explicitly below. This formulation often leads people to call such a transform a discrete cosine transform (DCT), although it is really just a special case of the DFT.

In each of the definitions below, we transform a real array  $X$  of length  $n$  to a real array  $Y$  of length  $n$ :

#### 27.1.3.1 REDFT00 (DCT-I)

---

Function **FFTW\_REDFT00**( $X$  As *mpNum[]*) As *mpNum[]*

---

The function **FFTW\_REDFT00** returns a real vector containing the REDFT00 transform (type-I DCT) transform of  $X$ .

**Parameter:**

$X$ : A real vector.

An REDFT00 transform (type-I DCT) in FFTW is defined by:

$$Y_k = X_0 + (-1)^k X_{n-1} + 2 \sum_{j=1}^{n-2} X_j \cos[\pi j k / (n - 1)]. \quad (27.1.5)$$

Note that this transform is not defined for  $n = 1$ . For  $n = 2$ , the summation term above is dropped as you might expect.

#### 27.1.3.2 REDFT10 (DCT-II)

---

Function **FFTW\_REDFT10**( $X$  As *mpNum[]*) As *mpNum[]*

---

The function **FFTW\_REDFT10** returns a real vector containing the REDFT10 transform (type-II DCT) transform of  $X$ .

**Parameter:**

$X$ : A real vector.

An REDFT10 transform (type-II DCT, sometimes called "the" DCT) in FFTW is defined by:

$$Y_k = 2 \sum_{j=0}^{n-1} X_j \cos[\pi(j + 1/2)k / n]. \quad (27.1.6)$$

### 27.1.3.3 REDFT01 (DCT-III)

---

Function **FFTW\_REDFT01**(*X* As *mpNum[]*) As *mpNum[]*

---

The function FFTW\_REDFT01 returns a real vector containing the REDFT01 transform (type-III DCT) transform of *X*.

**Parameter:**

*X*: A real vector.

An REDFT01 transform (type-III DCT) in FFTW is defined by:

$$Y_k = X_0 + 2 \sum_{j=1}^{n-1} X_j \cos[\pi j(k + 1/2)/n]. \quad (27.1.7)$$

In the case of  $n = 1$ , this reduces to  $Y_0 = X_0$ . Up to a scale factor (see below), this is the inverse of REDFT10 ("the" DCT), and so the REDFT01 (DCT-III) is sometimes called the "IDCT".

### 27.1.3.4 REDFT11 (DCT-IV)

---

Function **FFTW\_REDFT11**(*X* As *mpNum[]*) As *mpNum[]*

---

The function FFTW\_REDFT11 returns a real vector containing the REDFT11 transform (type-IV DCT) transform of *X*.

**Parameter:**

*X*: A real vector.

An REDFT11 transform (type-IV DCT) in FFTW is defined by:

$$Y_k = 2 \sum_{j=0}^{n-1} X_j \cos[\pi(j + 1/2)(k + 1/2)/n]. \quad (27.1.8)$$

### 27.1.3.5 Inverses and Normalization

These definitions correspond directly to the unnormalized DFTs used elsewhere in FFTW (hence the factors of 2 in front of the summations). The unnormalized inverse of REDFT00 is REDFT00, of REDFT10 is REDFT01 and vice versa, and of REDFT11 is REDFT11. Each unnormalized inverse results in the original array multiplied by  $N$ , where  $N$  is the logical DFT size. For REDFT00,  $N = 2(n - 1)$  (note that  $n = 1$  is not defined); otherwise,  $N = 2n$ .

In defining the discrete cosine transform, some authors also include additional factors of  $\sqrt{2}$  (or its inverse) multiplying selected inputs and/or outputs. This is a mostly cosmetic change that makes the transform orthogonal, but sacrifices the direct equivalence to a symmetric DFT.

## 27.1.4 1d Real-odd DFTs (DSTs)

The Real-odd symmetry DFTs in FFTW are exactly equivalent to the unnormalized forward (and backward) DFTs as defined above, where the input array *X* of length  $N$  is purely real and is also odd symmetry. In this case, the output array is odd symmetry and purely imaginary.

For the case of RODFT00, this odd symmetry means that  $X_j = -X_{N-j}$ , where we take *X* to be periodic so that  $X_N = X_0$ . Because of this redundancy, only the first  $n$  real numbers starting at  $j = 1$  are actually stored (the  $j = 0$  element is zero), where  $N = 2(n + 1)$ .

The proper definition of odd symmetry for RODFT10, RODFT01, and RODFT11 transforms is somewhat more intricate because of the shifts by 1/2 of the input and/or output. Because of the odd symmetry, however, the cosine terms in the DFT all cancel and the remaining sine terms are written explicitly below. This formulation often leads people to call such a transform a discrete sine transform (DCT), although it is really just a special case of the DFT.

In each of the definitions below, we transform a real array  $X$  of length  $n$  to a real array  $Y$  of length  $n$ :

#### 27.1.4.1 RODFT00 (DST-I)

---

Function **FFTW\_RODFT00**( $X$  As *mpNum[]*) As *mpNum[]*

---

The function FFTW\_RODFT00 returns a real vector containing the RODFT00 transform (type-I DST) transform of  $X$ .

**Parameter:**

$X$ : A real vector.

An RODFT00 transform (type-I DST) in FFTW is defined by:

$$Y_k = 2 \sum_{j=0}^{n-1} X_j \sin[\pi(j+1)(k+1)/(n+1)]. \quad (27.1.9)$$

#### 27.1.4.2 RODFT10 (DST-II)

---

Function **FFTW\_RODFT10**( $X$  As *mpNum[]*) As *mpNum[]*

---

The function FFTW\_RODFT10 returns a real vector containing the RODFT10 transform (type-II DST) transform of  $X$ .

**Parameter:**

$X$ : A real vector.

An RODFT10 transform (type-II DST) in FFTW is defined by:

$$Y_k = 2 \sum_{j=0}^{n-1} X_j \sin[\pi(j+1/2)(k+1/2)/n]. \quad (27.1.10)$$

#### 27.1.4.3 RODFT01 (DST-III)

---

Function **FFTW\_RODFT01**( $X$  As *mpNum[]*) As *mpNum[]*

---

The function FFTW\_RODFT01 returns a real vector containing the RODFT01 transform (type-III DST) transform of  $X$ .

**Parameter:**

$X$ : A real vector.

An RODFT01 transform (type-III DST) in FFTW is defined by:

$$Y_k = (-1)^k X_{n-1} + 2 \sum_{j=0}^{n-2} X_j \sin[\pi(j+1)(k+1/2)/n]. \quad (27.1.11)$$

In the case of  $n = 1$ , this reduces to  $Y_0 = X_0$ .

#### 27.1.4.4 RODFT11 (DST-IV)

---

Function **FFTW\_RODFT11(*X* As *mpNum[]*) As *mpNum[]***

---

The function FFTW\_RODFT11 returns a real vector containing the RODFT11 transform (type-IV DST) transform of *X*.

**Parameter:**

*X*: A real vector.

An RODFT11 transform (type-IV DST) in FFTW is defined by:

$$Y_k = 2 \sum_{j=0}^{n-1} X_j \sin[\pi(j + 1/2)(k + 1/2)/n]. \quad (27.1.12)$$

#### 27.1.4.5 Inverses and Normalization

These definitions correspond directly to the unnormalized DFTs used elsewhere in FFTW (hence the factors of 2 in front of the summations). The unnormalized inverse of RODFT00 is RODFT00, of RODFT10 is RODFT01 and vice versa, and of RODFT11 is RODFT11. Each unnormalized inverse results in the original array multiplied by *N*, where *N* is the logical DFT size. For RODFT00, *N* = 2(*n* + 1); otherwise, *N* = 2*n*.

In defining the discrete sine transform, some authors also include additional factors of  $\sqrt{2}$  (or its inverse) multiplying selected inputs and/or outputs. This is a mostly cosmetic change that makes the transform orthogonal, but sacrifices the direct equivalence to a symmetric DFT.

# Chapter 28

## Minimization and Optimization: Procedures based on MINPACK

Reference for MINPACK: [Moré \*et al.\* \(1980\)](#).

### Detailed Description

```
#include <junsupported/Eigen/NonLinearOptimization>
```

This module provides implementation of two important algorithms in non linear optimization. In both cases, we consider a system of non linear functions. Of course, this should work, and even work very well if those functions are actually linear. But if this is so, you should probably better use other methods more fitted to this special case.

One algorithm allows to find an extremum of such a system (Levenberg Marquardt algorithm) and the second one is used to find a zero for the system (Powell hybrid "dogleg" method).

This code is a port of MINPACK . Minpack is a very famous, old, robust and well-known package, written in fortran. Those implementations have been carefully tuned, tested, and used for several decades.

The original fortran code was automatically translated using f2c in C, then c++, and then cleaned by several different authors.

Finally, we ported this code to Eigen, creating classes and API coherent with Eigen. When possible, we switched to Eigen implementation, such as most linear algebra (vectors, matrices, stable norms).

Doing so, we were very careful to check the tests we setup at the very beginning, which ensure that the same results are found.

### Tests

The tests are placed in the file `unsupported/test/NonLinear.cpp`.

There are two kinds of tests : those that come from examples bundled with cminpack. They guarantee we get the same results as the original algorithms (value for 'x', for the number of evaluations of the function, and for the number of evaluations of the jacobian if ever).

Other tests were added by myself at the very beginning of the process and check the results for levenberg-marquardt using the reference data on NIST. Since then i've carefully checked that the same results were obtained when modifying the code. Please note that we do not always get the exact same decimals as they do, but this is ok : they use 128bits float, and we do the tests using the C type 'double', which is 64 bits on most platforms (x86 and amd64, at least). I've performed those tests on several other implementations of levenberg-marquardt, and (c)minpack performs VERY well compared to those, both in accuracy and speed.

The documentation for running the tests is on the wiki <http://eigen.tuxfamily.org/index.php?title=Tests API : overview of methods>

Both algorithms can use either the jacobian (provided by the user) or compute an approximation by themselves (actually using Eigen Numerical differentiation module). The part of API referring to the latter use 'NumericalDiff' in the method names (exemple: LevenbergMarquardt.minimizeNumericalDiff() )

The methods LevenbergMarquardt.lmder1()/lmdif1()/lmstr1() and HybridNonLinearSolver.hybrj1()/hybrd are specific methods from the original minpack package that you probably should NOT use until you are porting a code that was previously using minpack. They just define a 'simple' API with default values for some parameters.

All algorithms are provided using Two APIs :

one where the user inits the algorithm, and uses '\*OneStep()' as much as he wants : this way the caller have control over the steps

one where the user just calls a method (optimize() or solve()) which will handle the loop: init + loop until a stop condition is met. Those are provided for convenience.

As an example, the method LevenbergMarquardt::minimize() is implemented as follow :

---

```
 Status LevenbergMarquardt<FunctorType,Scalar>::minimize(FVectorType &x, const int
 mode)
{
    Status status = minimizeInit(x, mode);
    do {
        status = minimizeOneStep(x, mode);
    } while (status==Running);
    return status;
}
```

---

The easiest way to understand how to use this module is by looking at the many examples in the file unsupported/test/NonLinearOptimization.cpp.

## 28.1 Multidimensional Rootfinding: Powell Hybrid

This is a modified version of PowellâŽs Hybrid method as implemented in the hybrj algorithm in minpack. The Hybrid algorithm retains the fast convergence of NewtonâŽs method but will also reduce the residual when NewtonâŽs 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  $|D(x - \bar{x})| < \delta$ , 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

$$dx = -\alpha J^{-1} f(x) - \beta \nabla |f(x)|^2. \quad (28.1.1)$$

staying inside the trust region,

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.

## 28.2 Nonlinear LeastSquares: Levenberg-Marquardt

The minimization algorithms described in this section make use of both the function and its derivative. They require an initial guess for the location of the minimum. 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 minimum for it to work.*

**lmsder** [Derivative Solver] This is a robust and efficient version of the Levenberg-Marquardt algorithm as implemented in the scaled lmder routine in minpack. Minpack was written by Jorge J. Moré, Burton S. Garbow and Kenneth E. Hillstrom.

The algorithm uses a generalized trust region to keep each step under control. In order to be accepted a proposed new position  $x_{\text{new}}$  must satisfy the condition  $|D(x_{\text{old}} - x_{\text{new}})| < \delta$ , 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 attempts to minimize the linear system  $|F + Jp|$  subject to the constraint  $|Dp| < \cdot$ . The solution to this constrained linear system is found using the Levenberg-Marquardt method.

The proposed step is now tested by evaluating the function at the resulting point,  $x_{\text{new}}$ . If the step reduces the norm of the function sufficiently, and follows the predicted behavior of the function within the trust region, then it is accepted and the size of the trust region is increased. If the proposed step fails to improve the solution, or differs significantly from the expected behavior within the trust region, then the size of the trust region is decreased and another trial step is computed. The algorithm also monitors the progress of the solution and returns an error if the changes in the solution are smaller than the machine precision. The possible error codes are,

# Chapter 29

## Procedures based on NLOPT

### 29.1 Overview

Reference to NLOPT is is [Johnson \(2012\)](#)

#### Nomenclature

Each algorithm in NLOpt is identified by a named constant, which is passed to the NLOpt routines in the various languages in order to select a particular algorithm. These constants are mostly of the form NLOPT\_G\_LN\_D\_xxxx, where G/L denotes global/local optimization and N/D denotes derivative-free/gradient-based algorithms, respectively.

For example, the NLOPT\_LN\_COBYLA constant refers to the COBYLA algorithm (described below), which is a local (L) derivative-free (N) optimization algorithm.

Two exceptions are the MLSL and augmented Lagrangian algorithms, denoted by NLOPT\_G\_MSL and NLOPT\_AUGLAG, since whether or not they use derivatives (and whether or not they are global, in AUGLAG's case) is determined by what subsidiary optimization algorithm is specified.

Many of the algorithms have several variants, which are grouped together below.

#### Comparing algorithms

For any given optimization problem, it is a good idea to compare several of the available algorithms that are applicable to that problem. In general, one often finds that the "best" algorithm strongly depends upon the problem at hand.

However, comparing algorithms requires a little bit of care because the function-value/parameter tolerance tests are not all implemented in exactly the same way for different algorithms. So, for example, the same fractional  $10^{-4}$  tolerance on the function value might produce a much more accurate minimum in one algorithm compared to another, and matching them might require some experimentation with the tolerances.

Instead, a more fair and reliable way to compare two different algorithms is to run one until the function value is converged to some value  $f_A$ , and then run the second algorithm with the  $\min f_{max}$  termination test set to  $\min f_{max} = f_A$ . That is, ask how long it takes for the two algorithms to reach the same function value.

Better yet, run some algorithm for a really long time until the minimum  $f_M$  is located to high precision. Then run the different algorithms you want to compare with the termination test:  $\min f_{max} = f_M + \Delta f$ . That is, ask how long it takes for the different algorithms to obtain the minimum to within an absolute tolerance  $\Delta f$ , for some  $\Delta f$ . (This is totally different from using the  $ftol_{abs}$  termination test, because the latter uses only a crude estimate of the error in the function values, and moreover the estimate varies between algorithms.)

## 29.2 Global optimization

All of the global-optimization algorithms currently require you to specify bound constraints on all the optimization parameters. Of these algorithms, only ISRES and ORIG\_DIRECT support nonlinear inequality constraints, and only ISRES supports nonlinear equality constraints. (However, any of them can be applied to nonlinearly constrained problems by combining them with the augmented Lagrangian method below.)

Something you should consider is that, after running the global optimization, it is often worthwhile to then use the global optimum as a starting point for a local optimization to "polish" the optimum to a greater accuracy. (Many of the global optimization algorithms devote more effort to searching the global parameter space than in finding the precise position of the local optimum accurately.)

### 29.2.1 DIRECT and DIRECT-L

DIRECT is the DIviding RECTangles algorithm for global optimization, described in [Jones \*et al.\* \(1993\)](#)

and DIRECT-L is the "locally biased" variant proposed by [Gablonsky & Kelley \(2001\)](#)

These are deterministic-search algorithms based on systematic division of the search domain into smaller and smaller hyperrectangles. The Gablonsky version makes the algorithm "more biased towards local search" so that it is more efficient for functions without too many local minima. NLopt contains several implementations of both of these algorithms. I would tend to try NLOPT\_GN\_DIRECT\_L first; YMMV.

First, it contains a from-scratch re-implementation of both algorithms, specified by the constants NLOPT\_GN\_DIRECT and NLOPT\_GN\_DIRECT\_L, respectively.

Second, there is a slightly randomized variant of DIRECT-L, specified by NLOPT\_GLOBAL\_DIRECT\_L\_RAND, which uses some randomization to help decide which dimension to halve next in the case of near-ties.

The DIRECT and DIRECT-L algorithms start by rescaling the bound constraints to a hypercube, which gives all dimensions equal weight in the search procedure. If your dimensions do not have equal weight, e.g. if you have a "long and skinny" search space and your function varies at about the same speed in all directions, it may be better to use unscaled variants of these algorithms, which are specified as NLOPT\_GLOBAL\_DIRECT\_NOSCAL, NLOPT\_GLOBAL\_DIRECT\_L\_NOSCAL, and NLOPT\_GLOBAL\_DIRECT\_L\_RAND\_NOSCAL, respectively. However, the unscaled variations make the most sense (if any) with the original DIRECT algorithm, since the design of DIRECT-L to some extent relies on the search region being a hypercube (which causes the subdivided hyperrectangles to have only a small set of side lengths).

Finally, NLopt also includes separate implementations based on the original Fortran code by Gablonsky *et al.* (1998-2001), which are specified as NLOPT\_GN\_ORIG\_DIRECT and NLOPT\_GN\_ORIG\_DIRECT\_L. These implementations have a number of hard-coded limitations on things like the number of function evaluations; I removed several of these limitations, but some remain. On the other hand, there seem to be slight differences between these implementations and mine; most of the time, the performance is roughly similar, but occasionally Gablonsky's implementation will do significantly better than mine or vice versa.

Most of the above algorithms only handle bound constraints, and in fact require finite bound constraints (they are not applicable to unconstrained problems). They do not handle arbitrary nonlinear constraints. However, the ORIG versions by Gablonsky *et al.* include some support for arbitrary nonlinear inequality constraints.

### 29.2.2 Controlled Random Search (CRS) with local mutation

My implementation of the "controlled random search" (CRS) algorithm (in particular, the CRS2 variant) with the "local mutation" modification, as defined by: [Kaelo & Ali \(2006\)](#).

The original CRS2 algorithm was described by: [Price \(1978, 1983\)](#)

The CRS algorithms are sometimes compared to genetic algorithms, in that they start with a random "population" of points, and randomly "evolve" these points by heuristic rules. In this case, the "evolution" somewhat resembles a randomized Nelder-Mead algorithm. The published results for CRS seem to be largely empirical; limited analytical results about its convergence were derived in [Hendrix \*et al.\* \(2001\)](#)

The initial population size for CRS defaults to  $10\sqrt{n+1}$  in  $n$  dimensions, but this can be changed with the `nlopt_set_stochastic_population` function; the initial population must be at least  $n+1$ .

Only bound-constrained problems are supported by this algorithm.

CRS2 with local mutation is specified in NLOpt as `NLOPT_GN_CRS2_LM`.

### 29.2.3 MLSL (Multi-Level Single-Linkage)

This is my implementation of the "Multi-Level Single-Linkage" (MLSL) algorithm for global optimization by a sequence of local optimizations from random starting points, proposed by: [Rinnooy Kan & Timmer \(1987a,b\)](#)

We also include a modification of MLSL use a Sobol' low-discrepancy sequence (LDS) instead of pseudorandom numbers, which was argued to improve the convergence rate by: [Kucherenko & Sytsko \(2005\)](#)

In either case, MLSL is a "multistart" algorithm: it works by doing a sequence of local optimizations (using some other local optimization algorithm) from random or low-discrepancy starting points. MLSL is distinguished, however by a "clustering" heuristic that helps it to avoid repeated searches of the same local optima, and has some theoretical guarantees of finding all local optima in a finite number of local minimizations.

The local-search portion of MLSL can use any of the other algorithms in NLOpt, and in particular can use either gradient-based (D) or derivative-free algorithms (N). The local search uses the derivative/nondervative algorithm set by `nlopt_opt_set_local_optimizer`.

LDS-based MLSL with is specified as `NLOPT_G_MSL_LDS`, while the original non-LDS original MLSL (using pseudo-random numbers, currently via the Mersenne twister algorithm) is indicated by `NLOPT_G_MSL`. In both cases, you must specify the local optimization algorithm (which can be gradient-based or derivative-free) via `nlopt_opt_set_local_optimizer`.

Note: If you do not set a stopping tolerance for your local-optimization algorithm, MLSL defaults to `ftol_rel=10^-15` and `xtol_rel=10^-7` for the local searches. Note that it is perfectly reasonable to set a relatively large tolerance for these local searches, run MLSL, and then at the end run another local optimization with a lower tolerance, using the MLSL result as a starting point, to "polish off" the optimum to high precision.

By default, each iteration of MLSL samples 4 random new trial points, but this can be changed with the `nlopt_set_population` function.

Only bound-constrained problems are supported by this algorithm.

### 29.2.4 StoGO

This is an algorithm adapted from the code downloaded from

StoGO global optimization library (link broken as of Nov. 2009, and the software seems absent from the author's web site) by Madsen et al. StoGO is a global optimization algorithm that works by systematically dividing the search space (which must be bound-constrained) into smaller hyper-rectangles via a branch-and-bound technique, and searching them by a gradient-based local-search algorithm (a BFGS variant), optionally including some randomness (hence the "Sto", which stands for "stochastic" I believe).

StoGO is written in C++, which means that it is only included when you compile the C++ algorithms enabled, in which case (on Unix) you must link to -lnlopt\_cxx instead of -lnlopt.

StoGO is specified within NLOpt by NLOPT\_GD\_STOGO, or NLOPT\_GD\_STOGO\_RAND for the randomized variant.

Some references on StoGO are: [Gudmundsson \(1998\)](#), [Madsen et al. \(1998\)](#), [Zertchaninov & Madsen \(1998\)](#)

Only bound-constrained problems are supported by this algorithm.

### 29.2.5 ISRES (Improved Stochastic Ranking Evolution Strategy)

This is my implementation of the "Improved Stochastic Ranking Evolution Strategy" (ISRES) algorithm for nonlinearly-constrained global optimization (or at least semi-global; although it has heuristics to escape local optima, I'm not aware of a convergence proof), based on the method described in: [Runarsson & Yao \(2005\)](#)

It is a refinement of an earlier method described in: [Runarsson & Yao \(2000\)](#)

This is an independent implementation by S. G. Johnson (2009) based on the papers above. Runarsson also has his own Matlab implementation available from his web page here.

The evolution strategy is based on a combination of a mutation rule (with a log-normal step-size update and exponential smoothing) and differential variation (a Nelder-Mead-like update rule). The fitness ranking is simply via the objective function for problems without nonlinear constraints, but when nonlinear constraints are included the stochastic ranking proposed by Runarsson and Yao is employed. The population size for ISRES defaults to  $20\sqrt{n+1}$  in  $n$  dimensions, but this can be changed with the nlopt\_set\_stochastic\_population function.

This method supports arbitrary nonlinear inequality and equality constraints in addition to the bound constraints, and is specified within NLOpt as NLOPT\_GN\_ISRES.

## 29.3 Local derivative-free optimization

Of these algorithms, only COBYLA currently supports arbitrary nonlinear inequality and equality constraints; the rest of them support bound-constrained or unconstrained problems only. (However, any of them can be applied to nonlinearly constrained problems by combining them with the augmented Lagrangian method below.)

### 29.3.1 COBYLA (Constrained Optimization BY Linear Approximations)

This is a derivative of Powell's implementation of the COBYLA (Constrained Optimization BY Linear Approximations) algorithm for derivative-free optimization with nonlinear inequality and equality constraints, by M. J. D. Powell, described in: [Powell \(1994\)](#) and reviewed in: [Powell \(1998\)](#)

It constructs successive linear approximations of the objective function and constraints via a simplex of  $n+1$  points (in  $n$  dimensions), and optimizes these approximations in a trust region at each step.

The original code itself was written in Fortran by Powell and was converted to C in 2004 by Jean-Sebastien Roy (js@jeannot.org) for the SciPy project. The version in NLOpt was based on Roy's C version, downloaded from:

<http://www.jeannot.org/js/code/index.en.html#COBYLA> NLOpt's version is slightly modified in a few ways. First, we incorporated all of the NLOpt termination criteria. Second, we added explicit support for bound constraints (although the original COBYLA could handle bound constraints as linear constraints, it would sometimes take a step that violated the bound constraints). Third, we allow COBYLA to increase the trust-region radius if the predicted improvement was approximately right and the simplex is OK, following a suggestion in the SAS manual for PROC NLP that seems to improve convergence speed. Fourth, we pseudo-randomize simplex steps in COBYLA algorithm, improving robustness by avoiding accidentally taking steps that don't improve conditioning (which seems to happen sometimes with active bound constraints); the algorithm remains deterministic (a deterministic seed is used), however. Also, we support unequal initial-step sizes in the different parameters (by the simple expedient of internally rescaling the parameters proportional to the initial steps), which is important when different parameters have very different scales.

(The underlying COBYLA code only supports inequality constraints. Equality constraints are automatically transformed into pairs of inequality constraints, which in the case of this algorithm seems not to cause problems.)

It is specified within NLOpt as NLOPT\_LN\_COBYLA.

### 29.3.2 BOBYQA

This is an algorithm derived from the BOBYQA subroutine of M. J. D. Powell, converted to C and modified for the NLOpt stopping criteria. BOBYQA performs derivative-free bound-constrained optimization using an iteratively constructed quadratic approximation for the objective function. See: [Powell \(2009\)](#)

(Because BOBYQA constructs a quadratic approximation of the objective, it may perform poorly for objective functions that are not twice-differentiable.)

The NLOpt BOBYQA interface supports unequal initial-step sizes in the different parameters (by the simple expedient of internally rescaling the parameters proportional to the initial steps), which is important when different parameters have very different scales.

This algorithm, specified in NLOpt as NLOPT\_LN\_BOBYQA, largely supersedes the NEWUOA algorithm below, which is an earlier version of the same idea by Powell.

### 29.3.3 NEWUOA + bound constraints

This is an algorithm derived from the NEWUOA subroutine of M. J. D. Powell, converted to C and modified for the NLOpt stopping criteria. I also modified the code to include a variant, NEWUOA-bound, that permits efficient handling of bound constraints. This algorithm is largely superseded by BOBYQA (above).

The original NEWUOA performs derivative-free unconstrained optimization using an iteratively constructed quadratic approximation for the objective function. See: [Powell \(2004\)](#)

(Because NEWUOA constructs a quadratic approximation of the objective, it may perform poorly for objective functions that are not twice-differentiable.)

The original algorithm is specified in NLOpt as NLOPT\_LN\_NEWUOA, and only supports unconstrained problems. For bound constraints, my variant is specified as NLOPT\_LN\_NEWUOA\_BOUND. In the original NEWUOA algorithm, Powell solved the quadratic subproblems (in routines TR-SAPP and BIGLAG) in a spherical trust region via a truncated conjugate-gradient algorithm. In my bound-constrained variant, we use the MMA algorithm for these subproblems to solve them with both bound constraints and a spherical trust region. In principle, we should also change the BIGDEN subroutine in a similar way (since BIGDEN also approximately solves a trust-region subproblem), but instead I just truncated its result to the bounds (which probably gives suboptimal convergence, but BIGDEN is called only very rarely in practice).

Shortly after my addition of bound constraints to NEWUOA, Powell released his own version of NEWUOA modified for bound constraints as well as some numerical-stability and convergence enhancements, called BOBYQA. NLOpt now incorporates BOBYQA as well, and it seems to largely supersede NEWUOA.

Note: NEWUOA requires the dimension  $n$  of the parameter space to be  $\geq 2$ , i.e. the implementation does not handle one-dimensional optimization problems.

### 29.3.4 PRAXIS (Principal AXIS)

"PRAXIS" gradient-free local optimization via the "principal-axis method" of Richard Brent, based on a C translation of Fortran code downloaded from Netlib:

<http://netlib.org/opt/praxis> The original Fortran code was written by Richard Brent and made available by the Stanford Linear Accelerator Center, dated 3/1/73. The appropriate reference seems to be: [Brent \(1972\)](#)

Specified in NLOpt as NLOPT\_LN\_PRAXIS

This algorithm was originally designed for unconstrained optimization. In NLOpt, bound constraints are "implemented" in PRAXIS by the simple expedient of returning infinity (Inf) when the constraints are violated (this is done automatically—*you don't have to do this in your own function*). This seems to work, more-or-less, but appears to slow convergence significantly. If you have bound constraints, you are probably better off using COBYLA or BOBYQA.

### 29.3.5 Nelder-Mead Simplex

My implementation of almost the original Nelder-Mead simplex algorithm (specified in NLOpt as NLOPT\_LN\_NELDERMEAD), as described in: [Nelder & Mead \(1965\)](#)

This method is simple and has demonstrated enduring popularity, despite the later discovery that it fails to converge at all for some functions (and examples may be constructed in which it converges to point that is not a local minimum). Anecdotal evidence suggests that it often performs well even for noisy and/or discontinuous objective functions. I would tend to recommend the Subplex method (below) instead, however.

The main change compared to the 1965 paper is that I implemented explicit support for bound constraints, using essentially the method proposed in: [Box \(1965\)](#)

and later reviewed in: [Richardson & Kuester \(1973\)](#)

Whenever a new point would lie outside the bound constraints, Box advocates moving it "just inside" the constraints by some fixed "small" distance of  $10\sqrt{\epsilon}$  or so. I couldn't see any advantage to using a fixed distance inside the constraints, especially if the optimum is on the constraint, so instead I move the point exactly onto the constraint in that case. The danger with implementing bound constraints in this way (or by Box's method) is that you may collapse the simplex into a lower-dimensional subspace. I'm not aware of a better way, however. In any case, this collapse

of the simplex is somewhat ameliorated by restarting, such as when Nelder-Mead is used within the Subplex algorithm below.

### 29.3.6 Sbplx (based on Subplex)

This is my re-implementation of Tom Rowan's "Subplex" algorithm. As Rowan expressed a preference that other implementations of his algorithm use a different name, I called my implementation "Sbplx" (referred to in NLOpt as NLOPT\_LN\_SBPLX).

Subplex (a variant of Nelder-Mead that uses Nelder-Mead on a sequence of subspaces) is claimed to be much more efficient and robust than the original Nelder-Mead, while retaining the latter's facility with discontinuous objectives, and in my experience these claims seem to be true in many cases. (However, I'm not aware of any proof that Subplex is globally convergent, and perhaps it may fail for some objectives like Nelder-Mead; YMMV.)

I used the description of Rowan's algorithm in his PhD thesis: [Rowan \(1990\)](#)

I would have preferred to use Rowan's original implementation, posted by him on Netlib: <http://www.netlib.org/opt/subplex.tgz> Unfortunately, the legality of redistributing or modifying this code is unclear, because it lacks anything resembling a license statement. After some friendly emails with Rowan in which he promised to consider providing a clear open-source/free-software license, I lost touch with him and his old email address now seems invalid.

Since the algorithm is not too complicated, however, I just rewrote it. There seem to be slight differences between the behavior of my implementation and his (probably due to different choices of initial subspace and other slight variations, where his paper was ambiguous), but the number of iterations to converge on my test problems seems to be quite close (within  $\pm 10\%$  of the number of function evaluations for most problems).

The only major difference between my implementation and Rowan's, as far as I can tell, is that I implemented explicit support for bound constraints (via the method in the Box paper as described above). This seems to be a big improvement in the case where the optimum lies against one of the constraints.

## 29.4 Local gradient-based optimization

Of these algorithms, only MMA and SLSQP support arbitrary nonlinear inequality constraints, and only SLSQP supports nonlinear equality constraints; the rest support bound-constrained or unconstrained problems only. (However, any of them can be applied to nonlinearly constrained problems by combining them with the augmented Lagrangian method below.)

### 29.4.1 MMA (Method of Moving Asymptotes) and CCSA

My implementation of the globally-convergent method-of-moving-asymptotes (MMA) algorithm for gradient-based local optimization, including nonlinear inequality constraints (but not equality constraints), specified in NLOpt as NLOPT\_LD\_MMA, as described in: [Svanberg \(2002\)](#)

This is an improved CCSA ("conservative convex separable approximation") variant of the original MMA algorithm published by Svanberg in 1987, which has become popular for topology optimization. (Note: "globally convergent" does not mean that this algorithm converges to the global optimum; it means that it is guaranteed to converge to some local minimum from any feasible starting point.)

At each point  $x$ , MMA forms a local approximation using the gradient of  $f$  and the constraint functions, plus a quadratic "penalty" term to make the approximations "conservative" (upper

bounds for the exact functions). The precise approximation MMA forms is difficult to describe in a few words, because it includes nonlinear terms consisting of a poles at some distance from  $x$  (outside of the current trust region), almost a kind of Pade approximant. The main point is that the approximation is both convex and separable, making it trivial to solve the approximate optimization by a dual method. Optimizing the approximation leads to a new candidate point  $x$ . The objective and constraints are evaluated at the candidate point. If the approximations were indeed conservative (upper bounds for the actual functions at the candidate point), then the process is restarted at the new  $x$ . Otherwise, the approximations are made more conservative (by increasing the penalty term) and re-optimized.

(If you contact Professor Svanberg, he has been willing in the past to graciously provide you with his original code, albeit under restrictions on commercial use or redistribution. The MMA implementation in NLOpt, however, is completely independent of Svanberg's, whose code we have not examined; any bugs are my own, of course.)

I also implemented another CCSA algorithm from the same paper, NLOPT\_LD\_CCSAQ: instead of constructing local MMA approximations, it constructs simple quadratic approximations (or rather, affine approximations plus a quadratic penalty term to stay conservative). This is the ccsa\_quadratic code. It seems to have similar convergence rates to MMA for most problems, which is not surprising as they are both essentially similar. However, for the quadratic variant I implemented the possibility of preconditioning: including a user-supplied Hessian approximation in the local model. It is easy to incorporate this into the proof in Svanberg's paper, and to show that global convergence is still guaranteed as long as the user's "Hessian" is positive semidefinite, and in practice it can greatly improve convergence if the preconditioner is a good approximation for the real Hessian (at least for the eigenvectors of the largest eigenvalues).

### 29.4.2 SLSQP

Specified in NLOpt as NLOPT\_LD\_SLSQP, this is a sequential quadratic programming (SQP) algorithm for nonlinearly constrained gradient-based optimization (supporting both inequality and equality constraints), based on the implementation by Dieter Kraft and described in:

[Kraft \(1988, 1994\)](#)

(I believe that SLSQP stands for something like "Sequential Least-Squares Quadratic Programming," because the problem is treated as a sequence of constrained least-squares problems, but such a least-squares problem is equivalent to a QP.) The algorithm optimizes successive second-order (quadratic/least-squares) approximations of the objective function (via BFGS updates), with first-order (affine) approximations of the constraints.

The Fortran code was obtained from the SciPy project, who are responsible for obtaining permission to distribute it under a free-software (3-clause BSD) license.

The code was modified for inclusion in NLOpt by S. G. Johnson in 2010, with the following changes. The code was converted to C and manually cleaned up. It was modified to be re-entrant (preserving the reverse-communication interface but explicitly saving the state in a data structure). The reverse-communication interface was wrapped with an NLOpt-style interface, with NLOpt stopping conditions. The inexact line search was modified to evaluate the functions including gradients for the first step, since this removes the need to evaluate the function+gradient a second time for the same point in the common case when the inexact line search concludes after a single step; this is motivated by the fact that NLOpt's interface combines the function and gradient computations. Since roundoff errors sometimes pushed SLSQP's parameters slightly outside the bound constraints (not allowed by NLOpt), we added checks to force the parameters within the bounds. We fixed a bug in the LSEI subroutine (use of uninitialized variables) for the case where the number of equality constraints equals the dimension of the problem. The LSQ

subroutine was modified to handle infinite lower/upper bounds (in which case those constraints are omitted).

Note: Because the SLSQP code uses dense-matrix methods (ordinary BFGS, not low-storage BFGS), it requires  $O(n^2)$  storage and  $O(n^3)$  time in  $n$  dimensions, which makes it less practical for optimizing more than a few thousand parameters

### 29.4.3 Low-storage BFGS

This algorithm in NLOpt (specified by NLOPT\_LD\_LBFGS), is based on a Fortran implementation of the low-storage BFGS algorithm written by Prof. Ladislav Luksan, and graciously posted online under the GNU LGPL at:

<http://www.uivt.cas.cz/luksan/subroutines.html> The original L-BFGS algorithm, based on variable-metric updates via Strang recurrences, was described by the papers:

[Nocedal \(1980\)](#) and [Liu & Nocedal \(1989\)](#).

I converted Prof. Luksan's code to C with the help of f2c, and made a few minor modifications (mainly to include the NLOpt termination criteria).

One of the parameters of this algorithm is the number M of gradients to "remember" from previous optimization steps: increasing M increases the memory requirements but may speed convergence. NLOpt sets M to a heuristic value by default, but this can be changed by the set\_vector\_storage function.

### 29.4.4 Preconditioned truncated Newton

This algorithm in NLOpt, is based on a Fortran implementation of a preconditioned inexact truncated Newton algorithm written by Prof. Ladislav Luksan, and graciously posted online under the GNU LGPL at:

<http://www.uivt.cas.cz/luksan/subroutines.html>

NLOpt includes several variations of this algorithm by Prof. Luksan. First, a variant preconditioned by the low-storage BFGS algorithm with steepest-descent restarting, specified as NLOPT\_LD\_TNEWTON\_PRECOND\_RESTART. Second, simplified versions NLOPT\_LD\_TNEWTON\_PR (same without restarting), NLOPT\_LD\_TNEWTON\_RESTART (same without preconditioning), and NLOPT\_LD\_TNEWTON (same without restarting or preconditioning).

The algorithms are based on the ones described by: [Dembo & Steihaug \(1982\)](#)

I converted Prof. Luksan's code to C with the help of f2c, and made a few minor modifications (mainly to include the NLOpt termination criteria).

One of the parameters of this algorithm is the number M of gradients to "remember" from previous optimization steps: increasing M increases the memory requirements but may speed convergence. NLOpt sets M to a heuristic value by default, but this can be changed by the set\_vector\_storage function.

### 29.4.5 Shifted limited-memory variable-metric

This algorithm in NLOpt, is based on a Fortran implementation of a shifted limited-memory variable-metric algorithm by Prof. Ladislav Luksan, and graciously posted online under the GNU LGPL at:

<http://www.uivt.cas.cz/luksan/subroutines.html> There are two variations of this algorithm: NLOPT\_LD\_VAR2, using a rank-2 method, and NLOPT\_LD\_VAR1, using a rank-1 method.

The algorithms are based on the ones described by: [Vlcek & Luksan \(2006\)](#)

I converted Prof. Luksan's code to C with the help of f2c, and made a few minor modifications (mainly to include the NLOpt termination criteria).

One of the parameters of this algorithm is the number  $M$  of gradients to "remember" from previous optimization steps: increasing  $M$  increases the memory requirements but may speed convergence. NLOpt sets  $M$  to a heuristic value by default, but this can be changed by the `set_vector_storage` function.

## 29.5 NLOPT: Augmented Lagrangian algorithm

### 29.5.1 Implementation

There is one algorithm in NLOpt that fits into all of the above categories, depending on what subsidiary optimization algorithm is specified, and that is the augmented Lagrangian method described in: [Conn \*et al.\* \(1991\)](#) and [Birgin & Martínez \(2008\)](#)

This method combines the objective function and the nonlinear inequality/equality constraints (if any) into a single function: essentially, the objective plus a "penalty" for any violated constraints. This modified objective function is then passed to another optimization algorithm with no nonlinear constraints. If the constraints are violated by the solution of this sub-problem, then the size of the penalties is increased and the process is repeated; eventually, the process must converge to the desired solution (if it exists).

The subsidiary optimization algorithm is specified by the `nlopt_set_local_optimizer` function, described in the NLOpt Reference. (Don't forget to set a stopping tolerance for this subsidiary optimizer!) Since all of the actual optimization is performed in this subsidiary optimizer, the subsidiary algorithm that you specify determines whether the optimization is gradient-based or derivative-free. In fact, you can even specify a global optimization algorithm for the subsidiary optimizer, in order to perform global nonlinearly constrained optimization (although specifying a good stopping criterion for this subsidiary global optimizer is tricky).

The augmented Lagrangian method is specified in NLOpt as `NLOPT_AUGLAG`. We also provide a variant, `NLOPT_AUGLAG_EQ`, that only uses penalty functions for equality constraints, while inequality constraints are passed through to the subsidiary algorithm to be handled directly; in this case, the subsidiary algorithm must handle inequality constraints (e.g. MMA or COBYLA). While NLOpt uses an independent re-implementation of the Birgin and Martínez algorithm, those authors provide their own free-software implementation of the method as part of the TANGO project, and implementations can also be found in semi-free packages like LANCELOT.

# Part VI

## Boost and related libraries

# Chapter 30

## RandomNumbers

Random numbers are required in a number of different problem domains, such as

- numerics (simulation, Monte-Carlo integration)
- games (non-deterministic enemy behavior)
- security (key generation)
- testing (random coverage in white-box tests)

The Boost Random Number Generator Library provides a framework for random number generators with well-defined properties so that the generators can be used in the demanding numerics and security domains. For a general introduction to random numbers in numerics, see [Press \*et al.\* \(2007\)](#), Chapter 7.

Depending on the requirements of the problem domain, different variations of random number generators are appropriate:

This is based on the Boost Random Library [Maurer & Watanabe \(2013\)](#).

### 30.1 Definitions

#### 30.1.1 Random Device

Class `random_device` models a non-deterministic random number generator . It uses one or more implementation-defined stochastic processes to generate a sequence of uniformly distributed non-deterministic random numbers. For those environments where a non-deterministic random number generator is not available, class `random_device` must not be implemented. See [Eastlake \*et al.\* \(1994\)](#) for further discussions.

Implementation Note for Windows: On the Windows operating system, `token` is interpreted as the name of a cryptographic service provider. By default `random_device` uses `MS_DEF_PROV`.

#### 30.1.2 Uniform Random Number Generator

A uniform random number generator is a `NumberGenerator` that provides a sequence of random numbers uniformly distributed on a given range. The range can be compile-time fixed or available (only) after run-time construction of the object. The tight lower bound of some (finite) set  $S$  is the (unique) member  $l$  in  $S$ , so that for all  $v$  in  $S$ ,  $l \leq v$  holds. Likewise, the tight upper bound of some (finite) set  $S$  is the (unique) member  $u$  in  $S$ , so that for all  $v$  in  $S$ ,  $v \leq u$  holds.

For integer generators (i.e. integer T), the generated values  $x$  fulfill  $\min() \leq x \leq \max()$ , for non-integer generators (i.e. non-integer T), the generated values  $x$  fulfill  $\min() \leq x < \max()$ .

### 30.1.3 Pseudo-Random Number Generator

A pseudo-random number generator is a `UniformRandomNumberGenerator` which provides a deterministic sequence of pseudo-random numbers, based on some algorithm and internal state. Linear congruential and inversive congruential generators are examples of such pseudo-random number generators. Often, these generators are very sensitive to their parameters. In order to prevent wrong implementations from being used, an external testsuite should check that the generated sequence and the validation value provided do indeed match. [Knuth \(1997\)](#) gives an extensive overview on pseudo-random number generation. The descriptions for the specific generators contain additional references.

## 30.2 The Random Number Generator Interface

### 30.2.1 Sampling

This is a place holder reference for Excel Sampling.

## 30.3 Random number generator algorithms

This library provides several pseudo-random number generators. The quality of a pseudo random number generator crucially depends on both the algorithm and its parameters. This library implements the algorithms as class templates with template value parameters, hidden in namespace `boost::random`. Any particular choice of parameters is represented as the appropriately specializing `typedef` in namespace `boost`.

Pseudo-random number generators should not be constructed (initialized) frequently during program execution, for two reasons. First, initialization requires full initialization of the internal state of the generator. Thus, generators with a lot of internal state (see below) are costly to initialize. Second, initialization always requires some value used as a "seed" for the generated sequence. It is usually difficult to obtain several good seed values. For example, one method to obtain a seed is to determine the current time at the highest resolution available, e.g. microseconds or nanoseconds. When the pseudo-random number generator is initialized again with the then-current time as the seed, it is likely that this is at a near-constant (non-random) distance from the time given as the seed for first initialization. The distance could even be zero if the resolution of the clock is low, thus the generator re-iterates the same sequence of random numbers. For some applications, this is inappropriate.

---

#### Function **SaveDefaultRngState(*FName* As String) As Boolean**

---

The function `SaveDefaultRngState` returns a boolean value: TRUE if the state was successfully save, FALSE otherwise

**Parameter:**

*FName*: A String, containing the full path of the file.

---

#### Function **LoadDefaultRngState(*FName* As String) As Boolean**

---

The function `LoadDefaultRngState` returns a boolean value: TRUE if the state was successfully loaded, FALSE otherwise

#### Parameter:

*FName*: A String, containing the full path of the file.

Note that all pseudo-random number generators described below are `CopyConstructible` and `Assignable`. Copying or assigning a generator will copy all its internal state, so the original and the copy will generate the identical sequence of random numbers. Often, such behavior is not wanted.

The following table gives an overview of some characteristics of the generators. The cycle length is a rough estimate of the quality of the generator; the approximate relative speed is a performance measure, higher numbers mean faster random number generation.

### 30.3.1 Minimal Standard

The specialization `minstd_rand0` was originally suggested in [Lewis et al. \(1969\)](#)

It is examined more closely together with `minstd_rand` in [Park & Miller \(1988\)](#).

The specialization `minstd_rand` was suggested in [Park & Miller \(1988\)](#).

### 30.3.2 rand48

Class `rand48` models a pseudo-random number generator . It uses the linear congruential algorithm with the parameters  $a = 0x5DEECE66D$ ,  $c = 0xB$ ,  $m = 2^{48}$ . It delivers identical results to the `lrand48()` function available on some systems (assuming `lcong48` has not been called). It is only available on systems where `uint64_t` is provided as an integral type, so that for example static in-class constants and/or enum definitions with large `uint64_t` numbers work.

### 30.3.3 Ecuyer 1988

The specialization `ecuyer1988` was suggested in [L'Ecuyer \(1988\)](#)

### 30.3.4 Knuth b

The specialization `knuth_b` is specified by the C++ standard. It is described in [Knuth \(1981\)](#)

### 30.3.5 Kreutzer 1986

the specialization `kreutzer1986` was suggested in [Kreutzer \(1986\)](#)

### 30.3.6 Tauss 88

The specialization `taus88` was suggested in [L'Ecuyer \(1996\)](#)

### 30.3.7 Hellekalek 1995

The specialization `hellekalek1995` was suggested in [Hellekalek \(1995\)](#)

### 30.3.8 Mersenne-Twister 11213b

The specializations `mt11213b` and `mt19937` are from [Matsumoto & Nishimura \(1998\)](#)

### 30.3.9 Mersenne-Twister 19937

The specializations `mt11213b` and `mt19937` are from [Matsumoto & Nishimura \(1998\)](#)

### 30.3.10 Mersenne-Twister 19937 64

The specializations `mt11213b` and `mt19937` are from [Matsumoto & Nishimura \(1998\)](#). adapted for 64 bit. The recursion is similar but different, so the output is totally different from the 32-bit versions.

### 30.3.11 Lagged Fibonacci Generators

The specializations `lagged_fibonacci607` ... `lagged_fibonacci44497` use well tested lags. See [Brent \(1992a\)](#)

The lags used here can be found in [Brent \(1992b\)](#).

### 30.3.12 Ranlux Generators

The `ranlux` family of generators are described in [Luescher \(1994\)](#).

The levels are given in [James \(1994\)](#).

## 30.4 Random number distributions

### 30.4.1 Uniform, small integer

discrete uniform distribution on a small set of integers (much smaller than the range of the underlying generator) .

The distribution function `uniform_smallint` models a random distribution . On each invocation, it returns a random integer value uniformly distributed in the set of integer numbers  $\{min, min + 1, min + 2, \dots, max\}$ . It assumes that the desired range  $(max - min + 1)$  is small compared to the range of the underlying source of random numbers and thus makes no attempt to limit quantization errors.

Let  $r_{out} = (max - min + 1)$  be the desired range of integer numbers, and let  $r_{base}$  be the range of the underlying source of random numbers. Then, for the uniform distribution, the theoretical probability for any number  $i$  in the range  $r_{out}$  will be  $p_{out} = \frac{1}{r_{out}}$ . Likewise, assume a uniform distribution on for the underlying source of random numbers, i.e. . Let denote the random distribution generated by `uniform_smallint`. Then the sum over all  $i$  in of shall not exceed .

The template parameter `IntType` shall denote an integer-like value type.

[Note] Note

The property above is the square sum of the relative differences in probabilities between the desired uniform distribution and the generated distribution . The property can be fulfilled with the calculation , as follows: Let . The base distribution on is folded onto the range . The numbers  $i < r$  have assigned numbers of the base distribution, the rest has only . Therefore, for  $i < r$  and otherwise. Substituting this in the above sum formula leads to the desired result.

Note: The upper bound for is . Regarding the upper bound for the square sum of the relative quantization error of , it seems wise to either choose so that or ensure that is divisible by .

### 30.4.2 Uniform, integer

discrete uniform distribution on a set of integers; the underlying generator may be called several times to gather enough randomness for the output.

The class template `uniform_int_distribution` models a random distribution . On each invocation, it returns a random integer value uniformly distributed in the set of integers  $\{min, min+1, min+2, \dots, max\}$ .

The template parameter `IntType` shall denote an integer-like value type.

### 30.4.3 Uniform, 01

continuous uniform distribution on the range  $[0, 1)$ ; important basis for other distributions.

The distribution function `uniform_01` models a random distribution . On each invocation, it returns a random floating-point value uniformly distributed in the range  $[0..1)$ .

The template parameter `RealType` shall denote a float-like value type with support for binary operators `+`, `-`, and `/`.

Note: The current implementation is buggy, because it may not fill all of the mantissa with random bits. I'm unsure how to fill a (to-be-invented) `boost::bigfloat` class with random bits efficiently. It's probably time for a traits class.

### 30.4.4 Uniform, Real

continuous uniform distribution on some range  $[min, max]$  of real numbers

The class template `uniform_real_distribution` models a random distribution . On each invocation, it returns a random floating-point value uniformly distributed in the range  $[min..max)$ .

### 30.4.5 Discrete

discrete distribution with specific probabilities (rolling an unfair dice).

The class `discrete_distribution` models a random distribution . It produces integers in the range  $[0, n)$  with the probability of producing each value is specified by the parameters of the distribution.

Constructs a `discrete_distribution` from a `std::initializer_list`. If the `initializer_list` is empty, equivalent to the default constructor. Otherwise, the values of the `initializer_list` represent weights for the possible values of the distribution. For example, given the distribution

```
* discrete_distribution<> dist{1, 4, 5};
*
```

The probability of a 0 is  $1/10$ , the probability of a 1 is  $2/5$ , the probability of a 2 is  $1/2$ , and no other values are possible.

### 30.4.6 Piecewise constant

Constructs a `piecewise_constant_distribution` from two iterator ranges containing the interval boundaries and the interval weights. If there are less than two boundaries, then this is equivalent to the default constructor and creates a single interval,  $[0, 1)$ .

The values of the interval boundaries must be strictly increasing, and the number of weights must be one less than the number of interval boundaries. If there are extra weights, they are ignored. For example,

```
* double intervals[] = { 0.0, 1.0, 4.0 };
* double weights[] = { 1.0, 1.0 };
* piecewise_constant_distribution<> dist(
*     &intervals[0], &intervals[0] + 3, &weights[0]);
```

The distribution has a 50% chance of producing a value between 0 and 1 and a 50% chance of producing a value between 1 and 4.

### 30.4.7 Piecewise linear

Constructs a piecewise\_linear\_distribution from two iterator ranges containing the interval boundaries and the weights at the boundaries. If there are fewer than two boundaries, then this is equivalent to the default constructor and creates a distribution that produces values uniformly distributed in the range [0, 1).

The values of the interval boundaries must be strictly increasing, and the number of weights must be equal to the number of interval boundaries. If there are extra weights, they are ignored.

For example,

```
* double intervals[] = { 0.0, 1.0, 2.0 };
* double weights[] = { 0.0, 1.0, 0.0 };
* piecewise_constant_distribution<> dist(
*     &intervals[0], &intervals[0] + 3, &weights[0]);
*
```

produces a triangle distribution.

### 30.4.8 Triangle

Instantiations of triangle\_distribution model a random distribution.

A triangle\_distribution has three parameters,  $a$ ,  $b$ , and  $c$ , which are the smallest, the most probable and the largest values of the distribution respectively.

### 30.4.9 Uniform on Sphere

Instantiations of class template uniform\_on\_sphere model a random distribution. Such a distribution produces random numbers uniformly distributed on the unit sphere of arbitrary dimension dim.

# Chapter 31

## Special Functions (based on Boost)

Boost: [Maddock & Kormanyos \(2013\)](#)

The standard references are [Olver et al. \(2010\)](#), and [Temme \(1996\)](#), and [Press et al. \(2007\)](#)

See also [Gil et al. \(2007\)](#) and [Gil et al. \(2011\)](#)

See also [Cuyt et al. \(2008\)](#)

### 31.1 Gamma and Beta Functions

Detailed review of the gamma function can be found in [Pugh \(2004\)](#) and [Luschny \(2012\)](#).  
The implementation is based on [Bristow et al. \(2013\)](#)

#### 31.1.1 Gamma function $\Gamma(x)$

---

Function **TgammaBoost**(*x* As *mpNum*) As *mpNum*

---

The function **TgammaBoost** returns the gamma function for  $x \neq 0, -1, -2, \dots$

**Parameter:**

*x*: A real number.

The gamma function for  $x \neq 0, -1, -2, \dots$  is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \quad (x > 0), \quad (31.1.1)$$

and by analytic continuation if  $x < 0$ , using the reflection formula

$$\Gamma(x)\Gamma(1-x) = \pi/\sin(\pi x). \quad (31.1.2)$$

#### 31.1.2 Logarithm of $\Gamma(x)$

---

Function **LgammaBoost**(*x* As *mpNum*) As *mpNum*

---

The function **LgammaBoost** returns the logarithm of the gamma function.

**Parameter:**

*x*: A real number.

This function computes  $\ln |\Gamma(x)|$  for  $x \neq 0, -1, -2, \dots$ . If  $x < 0$  the function uses the logarithm of the reflection formula.

### 31.1.3 Auxiliary function $\Gamma(x)/\Gamma(x + \delta)$

---

Function **TgammaDeltaRatioBoost**(*x* As *mpNum*, *δ* As *mpNum*) As *mpNum*

---

The function **TgammaDeltaRatioBoost** returns the ratio of gamma functions.

**Parameters:**

*x*: A real number.

*δ*: A real number.

This functions returns the ratio of gamma functions in the form

$$\frac{\Gamma(a)}{\Gamma(a + \delta)} \quad (31.1.3)$$

Note that the result is calculated accurately even when  $\delta$  is small compared to  $a$ : indeed even if  $a + \delta \approx a$ . The function is typically used when  $a$  is large and  $\delta$  is very small.

### 31.1.4 Digamma function $\psi(x)$

---

Function **DigammaBoost**(*x* As *mpNum*) As *mpNum*

---

The function **DigammaBoost** returns the digamma function for  $x \neq 0, -1, -2, \dots$

**Parameter:**

*x*: A real number.

The digamma or  $\psi$  function is defined as

$$\psi(x) = \frac{d(\ln \Gamma(x))}{dx} = \frac{\Gamma'(x)}{\Gamma(x)}, \quad x \neq 0, -1, -2, \dots \quad (31.1.4)$$

If  $x < 0$  it is transformed to positive values with the reflection formula

$$\psi(1 - x) = \psi(x) + \pi \cot(\pi x) \quad (31.1.5)$$

and for  $0 < x < 12$  the recurrence formula

$$\psi(x + 1) = \psi(x) + \frac{1}{x} \quad (31.1.6)$$

### 31.1.5 Ratio of Gamma Functions

---

Function **TgammaratioBoost**(*a* As *mpNum*, *b* As *mpNum*) As *mpNum*

---

The function **TgammaratioBoost** returns the ratio of gamma functions.

**Parameters:**

*a*: A real number.

*b*: A real number.

This functions returns the ratio of gamma functions in the form

$$\frac{\Gamma(a)}{\Gamma(b)} \quad (31.1.7)$$

### 31.1.6 Normalised incomplete gamma functions

---

#### Function **GammaPBoost**(*a* As mpNum, *x* As mpNum) As mpNum

---

The function **GammaPBoost** returns the normalised incomplete gamma function  $P(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

The normalised incomplete gamma function  $P(a, x)$  is defined as

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt \quad (31.1.8)$$

for  $a \geq 0$  and  $x \geq 0$ .

---

#### Function **GammaQBoost**(*a* As mpNum, *x* As mpNum) As mpNum

---

The function **GammaQBoost** returns the normalised incomplete gamma function  $Q(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

Boost references are [Temme \(1979\)](#) and [Temme \(1994\)](#)

The normalised incomplete gamma function  $Q(a, x)$  is defined as

$$Q(a, x) = \frac{1}{\Gamma(a)} \int_x^\infty t^{a-1} e^{-t} dt \quad (31.1.9)$$

for  $a \geq 0$  and  $x \geq 0$ .

### 31.1.7 Non-Normalised incomplete gamma functions

---

#### Function **NonNormalisedGammaPBoost**(*a* As mpNum, *x* As mpNum) As mpNum

---

The function **NonNormalisedGammaPBoost** returns the non-normalised incomplete gamma function  $\Gamma(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

The non-normalised incomplete gamma function  $\Gamma(a, x)$  is defined as

$$\Gamma(a, x) = \int_0^x t^{a-1} e^{-t} dt \quad (31.1.10)$$

for  $a \geq 0$  and  $x \geq 0$ .

---

#### Function **NonNormalisedGammaQBoost**(*a* As mpNum, *x* As mpNum) As mpNum

---

The function **NonNormalisedGammaQBoost** returns the non-normalised incomplete gamma function  $\gamma(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

The non-normalised incomplete gamma function  $\gamma(a, x)$  is defined as

$$\gamma(a, x) = \int_x^\infty t^{a-1} e^{-t} dt \quad (31.1.11)$$

for  $a \geq 0$  and  $x \geq 0$ .

Note: in Boost, the functions are referred to as TgammaLower and TgammaUpper.

### 31.1.8 Inverse normalised incomplete gamma functions

---

#### Function **GammaPinvBoost(*a* As mpNum, *p* As mpNum) As mpNum**

---

The function GammaPinvBoost returns the inverse of the normalised incomplete gamma function  $P(a, x)$ .

**Parameters:**

- a*: A real number.  
*p*: A real number.

This function returns the inverse normalised incomplete gamma function, i.e. it calculates  $x$  with  $P(a, x) = p$ . The input parameters are  $a > 0$ ,  $p \geq 0$ , and  $p + q = 1$ .

---

#### Function **GammaQinvBoost(*a* As mpNum, *q* As mpNum) As mpNum**

---

The function GammaQinvBoost returns the inverse of the normalised incomplete gamma function  $Q(a, x)$ .

**Parameters:**

- a*: A real number.  
*q*: A real number.

This function returns the inverse normalised incomplete gamma function, i.e. it calculates  $x$  with  $Q(a, x) = q$ . The input parameters are  $a > 0$ ,  $q \geq 0$ , and  $p + q = 1$ .

---

#### Function **GammaPinvaBoost(*x* As mpNum, *p* As mpNum) As mpNum**

---

The function GammaPinvaBoost returns the parameter  $a$  of the normalised incomplete gamma function  $P(a, x)$ , such that  $P(a, x) = p$ .

**Parameters:**

- x*: A real number.  
*p*: A real number.

---

#### Function **GammaQinvaBoost(*x* As mpNum, *q* As mpNum) As mpNum**

---

The function GammaQinvaBoost returns the parameter  $a$  of the normalised incomplete gamma function  $Q(a, x)$ , such that  $Q(a, x) = q$ .

**Parameters:**

- x*: A real number.  
*q*: A real number.

**31.1.9 Derivative of the normalised incomplete gamma function**


---

Function **GammaPDerivativeBoost(*a* As mpNum, *x* As mpNum) As mpNum**

---

The function **GammaPDerivativeBoost** returns the partial derivative with respect to *x* of the incomplete gamma function  $P(a, x)$ .

**Parameters:**

- a*: A real number.  
*x*: A real number.

The partial derivative with respect to *x* of the incomplete gamma function  $P(a, x)$  is defined as:

$$\frac{\partial}{\partial x} P(a, x) = \frac{e^{-x} x^{a-1}}{\Gamma(a)}. \quad (31.1.12)$$

**31.2 Factorials and Binomial Coefficient****31.2.1 Factorial**


---

Function **FactorialBoost(*n* As mpNum) As mpNum**

---

The function **FactorialBoost** returns the factorial  $n! = \Gamma(n + 1) = n \times (n - 1) \times \cdots \times 1$ .

**Parameter:**

- n*: An integer.

**31.2.2 Double Factorial**


---

Function **DoubleFactorialBoost(*n* As mpNum) As mpNum**

---

The function **DoubleFactorialBoost** returns the double factorial  $n!!$ .

**Parameter:**

- n*: An integer.

For even  $n < 0$  the result is  $\infty$ . For positive  $n$  the double factorial is defined as

$$n!! = \begin{cases} 1 \cdot 3 \cdot 5 \cdots n & \text{if } n \text{ is odd.} \\ 2 \cdot 4 \cdot 6 \cdots n & \text{if } n \text{ is even.} \end{cases} \quad (31.2.1)$$

**31.2.3 Rising Factorial**


---

Function **RisingFactorialBoost(*n* As mpNum, *i* As mpNum) As mpNum**

---

The function **RisingFactorialBoost** returns the rising factorial of *x* and *i*.

**Parameters:**

*n*: An integer.

*i*: An integer.

Returns the rising factorial of *x* and *i*:

$$\text{RisingFactorial}(n, i) = n(n+1)(n+2)(n+3)\cdots(n+i-1) \quad (31.2.2)$$

or

$$(n)_i = \frac{\Gamma(n+i)}{\Gamma(n)}. \quad (31.2.3)$$

Note that both *n* and *i* can be negative as well as positive.

### 31.2.4 Falling Factorial

---

Function **FallingFactorialBoost**(*n* As *mpNum*, *i* As *mpNum*) As *mpNum*

---

The function **FallingFactorialBoost** returns the falling factorial of *x* and *i*.

**Parameters:**

*n*: An integer.

*i*: An integer.

The falling factorial of *x* and *i* is defined as:

$$\text{FallingFactorial}(n, i) = n(n-1)(n-2)(n-3)\cdots(n-i+1) \quad (31.2.4)$$

Note that this function is only defined for positive *i*, hence the unsigned second argument. Argument *n* can be either positive or negative however.

### 31.2.5 Binomial coefficient

---

Function **BinomialCoefficientBoost**(*n* As *mpNum*, *k* As *mpNum*) As *mpNum*

---

The function **BinomialCoefficientBoost** returns the binomial coefficient.

**Parameters:**

*n*: An integer.

*k*: An integer.

The binomial coefficient ("*n* choose *k*") is defined as

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots(1)} \quad (31.2.5)$$

for *k*  $\geq 0$ .

## 31.3 Beta Functions

### 31.3.1 Beta function B(*a*, *b*)

---

Function **BetaBoost**(*a* As *mpNum*, *b* As *mpNum*) As *mpNum*

---

The function `BetaBoost` returns the beta function.

**Parameters:**

- a*: A real number.
- b*: A real number.

The beta function is defined as

$$B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a + b)} \quad (31.3.1)$$

where  $\Gamma(\cdot)$  denotes the Gamma function (see section 31.1). The reference is [DiDonato & Morris \(1992\)](#)

## 31.4 Error Function and Related Functions

### 31.4.1 Error Function `erf`

---

Function **ErfBoost**(*x* As *mpNum*) As *mpNum*

---

The function `ErfBoost` returns the value of the error function.

**Parameter:**

- x*: A real number.

The error function is defined by

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt, \quad (31.4.1)$$

### 31.4.2 Complementary Error Function

---

Function **ErfcBoost**(*x* As *mpNum*) As *mpNum*

---

The function `ErfcBoost` returns the value of the complementary error function.

**Parameter:**

- x*: A real number.

### 31.4.3 Inverse Function of `erf`

---

Function **ErfInvBoost**(*x* As *mpNum*) As *mpNum*

---

The function `ErfInvBoost` returns the functional inverse of  $\text{erf}(x)$

**Parameter:**

- x*: A real number.

The functional inverse of  $\text{erf}(x)$  is defined by

$$\text{erf}(\text{erf\_inv}(x)) = x, \quad -1 < x < 1. \quad (31.4.2)$$

### 31.4.4 Inverse Function of erfc

---

Function **ErfcInvBoost**(*x* As *mpNum*) As *mpNum*

---

The function **ErfcInvBoost** returns the functional inverse of  $\text{erfc}(x)$

**Parameter:**

*x*: A real number.

The functional inverse of  $\text{erfc}(x)$  is defined by

$$\text{erfc}(\text{erfc\_inv}(x)) = x, \quad 0 < x < 2. \quad (31.4.3)$$

## 31.5 Polynomials

### 31.5.1 Legendre Polynomials/Functions

---

#### Function **LegendrePBoost**(*l* As mpNum, *x* As mpNum) As mpNum

---

The function LegendrePBoost returns  $P_l(x)$ , the Legendre functions of the first kind.

**Parameters:**

*l*: An Integer.

*x*: A real number.

These functions return  $P_l(x)$ , the Legendre functions of the first kind, also called Legendre polynomials if  $l \geq 0$  and  $|x| \leq 1$ . The Legendre polynomials are orthogonal on the interval  $(-1, 1)$  with  $w(x) = 1$ . If  $l \geq 0$  the function uses the recurrence relation for varying degree from [1, 8.5.3]:

$$\begin{aligned} P_0(x) &= 1 & (31.5.1) \\ P_1(x) &= x \\ (l+1)P_{l+1}(x) &= (2l+1)P_l(x) - lP_{l-1}(x). \end{aligned}$$

and for negative *l* the result is  $P_l(x) = P_{-l-1}(x)$ .

---

#### Function **LegendrePNextBoost**(*l* As mpNum, *x* As mpNum, *Pl* As mpNum, *Plm1* As mpNum) As mpNum

---

The function LegendrePNextBoost returns the Legendre function of the first kind of degree  $l+1$ , using the results for degree *l* and  $l-1$ .

**Parameters:**

*l*: An Integer. The degree of the last polynomial calculated.

*x*: A real number. The abscissa value.

*Pl*: A real number. The value of the polynomial evaluated at degree *l*.

*Plm1*: A real number. The value of the polynomial evaluated at degree  $l-1$ .

This function implements the recursion relation given in equation 31.5.1

### 31.5.2 Associated Legendre Polynomials/Functions

---

#### Function **AssociatedLegendrePlmBoost**(*l* As mpNum, *m* As mpNum, *x* As mpNum) As mpNum

---

The function AssociatedLegendrePlmBoost returns  $L_n^m(x)$ , the associated Legendre polynomials of degree  $l \geq 0$  and order  $m \geq 0$ .

**Parameters:**

*l*: An Integer.

*m*: An Integer.

*x*: A real number.

This function returns  $L_n^m(x)$ , the associated Legendre polynomials of degree  $l \geq 0$  and order  $m \geq 0$ , defined for  $m > 0, |x| < 1$  as

$$P_l^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x). \quad (31.5.2)$$

The following recursion relation holds:

$$(l - m + 1)P_{l+1}^m(x) = (2l + 1)xP_l^m(x) - (l + m + 1)P_{l-1}^m(x). \quad (31.5.3)$$

---

**Function `AssociatedLegendrePlmNextBoost`(*l* As `mpNum`, *m* As `mpNum`, *x* As `mpNum`, *Pl* As `mpNum`, *Plm1* As `mpNum`) As `mpNum`**

---

The function `AssociatedLegendrePlmNextBoost` returns the Legendre function of the first kind of degree  $l + 1$ , using the results for degree  $l$  and  $l - 1$ .

**Parameters:**

*l*: An Integer. The degree of the last polynomial calculated.

*m*: An Integer. The order of the Associated Polynomial.

*x*: A real number. The abscissa value.

*Pl*: A real number. The value of the polynomial evaluated at degree  $l$ .

*Plm1*: A real number. The value of the polynomial evaluated at degree  $l - 1$ .

This function implements the recursion relation given in equation 31.5.3

### 31.5.3 Legendre Functions of the Second Kind

---

**Function `LegendreQBoost`(*l* As `mpNum`, *x* As `mpNum`) As `mpNum`**

---

The function `LegendreQBoost` returns  $Q_l(x)$ , the Legendre functions of the second kind of degree  $l \geq 0$  and  $|x| \neq 1$ .

**Parameters:**

*l*: An Integer.

*x*: A real number.

These functions return  $Q_l(x)$ , the Legendre functions of the second kind of degree  $l \geq 0$  and  $|x| \neq 1$ , defined as

$$\begin{aligned} Q_0(x) &= \frac{1}{2} \ln \frac{1+x}{1-x} \\ Q_1(x) &= \frac{x}{2} \ln \frac{1+x}{1-x} - 1 \\ (k+1)Q_{k+1}(x) &= (2k+1)xQ_k(x) - kQ_{k-1}(x). \end{aligned} \quad (31.5.4)$$

### 31.5.4 Laguerre Polynomials

---

**Function `LaguerreLBoost`(*n* As `mpNum`, *x* As `mpNum`) As `mpNum`**

---

The function `LaguerreLBoost` returns  $L_n(x)$ , the Laguerre polynomials of degree  $n \geq 0$ .

**Parameters:**

*n*: An Integer.

*x*: A real number.

This function returns  $L_n(x)$ , the Laguerre polynomials of degree  $n \geq 0$ . The Laguerre polynomials are just special cases of the generalized Laguerre polynomials

$$L_n(x) = L_n^{(0)}(x). \quad (31.5.5)$$

The standard recurrence formulas are used:

$$\begin{aligned} L_0(x) &= 1 \\ L_1(x) &= -x + 1 \\ nL_n(x) &= (2n - 1 - x)L_{n-1}(x) - (n - 1)L_{n-2}(x). \end{aligned} \tag{31.5.6}$$

---

Function **LaguerreLNextBoost**(*n* As mpNum, *x* As mpNum, *Ln* As mpNum, *Lnm1* As mpNum) As mpNum

---

The function **LaguerreLNextBoost** returns the Laguerre polynomial of the first kind of degree  $n + 1$ , using the results for degree  $n$  and  $n - 1$ .

**Parameters:**

*n*: An Integer. The degree of the last polynomial calculated.

*x*: A real number. The abscissa value.

*Ln*: A real number. The value of the polynomial evaluated at degree  $n$ .

*Lnm1*: A real number. The value of the polynomial evaluated at degree  $n - 1$ .

This function implements the recursion relation given in equation 31.5.6

### 31.5.5 Associated Laguerre Polynomials

---

Function **AssociatedLaguerreBoost**(*n* As mpNum, *m* As mpNum, *x* As mpNum) As mpNum

---

The function **AssociatedLaguerreBoost** returns  $L_n^m(x)$ , the associated Laguerre polynomials of degree  $n \geq 0$  and order  $m \geq 0$ .

**Parameters:**

*n*: An Integer.

*m*: An Integer.

*x*: A real number.

This function returns  $L_n^m(x)$ , the associated Laguerre polynomials of degree  $n \geq 0$  and order  $m \geq 0$ , defined as

$$L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x). \tag{31.5.7}$$

The standard recurrence formulas are used:

$$\begin{aligned} L_0^a(x) &= 1 \\ L_1^a(x) &= -x + 1 + a \\ nL_n^a(x) &= (2n + a - 1 - x)L_{n-1}^a(x) - (n + a - 1)L_{n-2}^a(x). \end{aligned} \tag{31.5.8}$$

---

Function **AssociatedLaguerreLNextBoost**(*n* As mpNum, *m* As mpNum, *x* As mpNum, *Ln* As mpNum, *Lnm1* As mpNum) As mpNum

---

The function **AssociatedLaguerreLNextBoost** returns the associated Laguerre polynomial of the first kind of degree  $n + 1$ , using the results for degree  $n$  and  $n - 1$ .

**Parameters:**

*n*: An Integer. The degree of the last polynomial calculated.

*m*: An Integer. The order of the Associated Polynomial.

*x*: A real number. The abscissa value.

*Ln*: A real number. The value of the polynomial evaluated at degree *n*.

*Lnm1*: A real number. The value of the polynomial evaluated at degree *n* – 1.

This function implements the recursion relation given in equation 31.5.8

### 31.5.6 Hermite Polynomials

---

Function **HermiteHBoost**(*n* As mpNum, *x* As mpNum) As mpNum

---

The function **HermiteHBoost** returns  $H_n(x)$ , the Hermite polynomial of degree  $n \geq 0$ .

**Parameters:**

*n*: An Integer.

*x*: A real number.

These functions return  $H_n(x)$ , the Hermite polynomial of degree  $n \geq 0$ . The  $H_n$  are orthogonal on the interval  $(-\infty, \infty)$ , with respect to the weight function  $w(x) = e^{-x^2}$ . They are computed with the standard recurrence formulas [1, 22.7.13]:

$$H_0(x) = 1 \tag{31.5.9}$$

$$H_1(x) = 2x$$

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x).$$

---

Function **HermiteHNextBoost**(*n* As mpNum, *x* As mpNum, *Hn* As mpNum, *Hnm1* As mpNum) As mpNum

---

The function **HermiteHNextBoost** returns the Hermite polynomial of degree  $n+1$ , using the results for degree  $n$  and  $n-1$ .

**Parameters:**

*n*: An Integer. The degree of the last polynomial calculated.

*x*: A real number. The abscissa value.

*Hn*: A real number. The value of the polynomial evaluated at degree *n*.

*Hnm1*: A real number. The value of the polynomial evaluated at degree *n* – 1.

This function implements the recursion relation given in equation 31.5.9

### 31.5.7 Spherical Harmonic Functions

---

Function **SphericalHarmonicBoost**(*l* As mpNumList[2], *m* As mpNum,  $\theta$  As mpNum,  $\phi$  As mpNum) As mpNum

---

The function **SphericalHarmonicBoost** returns the real and imaginary parts of the spherical harmonic function  $Y_{lm}(\theta, \phi)$ .

**Parameters:**

*l*: An Integer.

$m$ : An Integer.

$\theta$ : A real number.

$\phi$ : A real number.

The procedures return the real and imaginary parts of the spherical harmonic function  $Y_{lm}(\theta, \phi)$ . These functions are closely related to the associated Legendre polynomials:

$$Y_{lm}(\theta, \phi) = \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}} P_l^m(\cos(\theta)) e^{im\phi} \quad (31.5.10)$$

## 31.6 Bessel Functions of Real Order

### 31.6.1 Bessel Function $J_\nu(x)$

---

Function **BesselJBoost**( $x$  As *mpNum*,  $\nu$  As *mpNum*) As *mpNum*

---

The function **BesselJBoost** returns  $J_\nu(z)$ , the Bessel function of the first kind of real order  $\nu$ .

**Parameters:**

$x$ : A real number.

$\nu$ : A real number.

$J_\nu(z)$ , the Bessel function of the first kind of order  $\nu$ , is defined as

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{k=0}^{\infty} (-1)^k \frac{(x^2/4)^k}{k!\Gamma(\nu+k+1)} \quad (31.6.1)$$

### 31.6.2 Bessel Function $Y_\nu(x)$

---

Function **BesselYBoost**( $x$  As *mpNum*,  $\nu$  As *mpNum*) As *mpNum*

---

The function **BesselYBoost** returns  $Y_\nu(z)$ , the Bessel function of the second kind of order  $\nu$ .

**Parameters:**

$x$ : A real number.

$\nu$ : A real number.

$Y_\nu(z)$ , the Bessel function of the second kind of order  $\nu$ , is defined as

$$Y_\nu(x) = \frac{J_\nu(x) \cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)} \quad (31.6.2)$$

## 31.7 Modified Bessel Functions of Real Order

### 31.7.1 Bessel Function $I_\nu(x)$

---

Function **BesselIBoost**( $x$  As *mpNum*,  $\nu$  As *mpNum*) As *mpNum*

---

The function **BesselIBoost** returns the modified Bessel function  $I_\nu(z)$  of the first kind of order  $\nu$ .

**Parameters:**

$x$ : A real number.

$\nu$ : A real number.

This function returns the modified Bessel function  $I_\nu(z)$  of the first kind of order  $\nu$ , defined as

$$I_\nu(z) = \frac{z}{2} \sum_{j=0}^{\infty} \frac{(z^2/4)^j}{j! \Gamma(\nu + j + 1)} \quad (31.7.1)$$

### 31.7.2 Bessel Function $K_\nu(x)$

---

Function **BesselKBoost**( $x$  As *mpNum*,  $\nu$  As *mpNum*) As *mpNum*

---

The function **BesselKBoost** returns  $K_\nu(x)$ , the modified Bessel function of the second kind of order  $\nu$ .

**Parameters:**

$x$ : A real number.

$\nu$ : A real number.

This function returns  $K_\nu(x)$ , the modified Bessel function of the second kind of order  $\nu$ , defined as

$$K_\nu(x) = \frac{\pi}{2} \frac{I_{-\nu}(x) - I_\nu(x)}{\sin(\nu\pi)} \quad (31.7.2)$$

## 31.8 Spherical Bessel Functions

### 31.8.1 Spherical Bessel function $j_n(x)$

---

Function **BesselSphericaljBoost**( $x$  As *mpNum*,  $\nu$  As *mpNum*) As *mpNum*

---

The function **BesselSphericaljBoost** returns  $j_n(x)$ , the spherical Bessel function of the 1st kind, order  $n$ .

**Parameters:**

$x$ : A real number.

$\nu$ : A real number.

The function  $j_n(x)$ , the spherical Bessel function of the 1st kind, order  $n$ , is defined as

$$j_n(x) = \sqrt{\frac{1}{2}\pi/x} J_{n+\frac{1}{2}}(x), \quad (x \leq 0), \quad \text{and } j_n(-x) = (-1)^n j_n(x). \quad (31.8.1)$$

### 31.8.2 Spherical Bessel function $y_n(x)$

---

Function **BesselSphericalyBoost**( $x$  As *mpNum*,  $\nu$  As *mpNum*) As *mpNum*

---

The function **BesselSphericalyBoost** returns  $y_n(x)$ , the spherical Bessel function of the 1st kind, order  $n$ .

**Parameters:**

$x$ : A real number.

$\nu$ : A real number.

The function  $y_n(x)$ , the spherical Bessel function of the second kind, order  $n$ ,  $x \neq 0$ , is defined as

$$y_n(x) = \sqrt{\frac{1}{2}\pi/x}Y_{n+\frac{1}{2}}(x), \quad (x > 0), \quad \text{and } y_n(-x) = (-1)^{n+1}y_n(x). \quad (31.8.2)$$

## 31.9 Hankel Functions

### 31.9.1 Hankel Function of the First Kind

---

Function **cplxHankel1Boost**(*x* As *mpNum*, *ν* As *mpNum*) As *mpNum*

---

The function **cplxHankel1Boost** returns the Hankel function of the first kind  $H_\nu^{(1)}(x)$ .

**Parameters:**

*x*: A real number.

*ν*: A real number.

This routine returns the Hankel function of the first kind  $H_\nu^{(1)}(x)$ , defined as

$$H_\nu^{(1)}(x) = J_\nu(x) + iY_\nu(x). \quad (31.9.1)$$

### 31.9.2 Hankel Function of the Second Kind

---

Function **cplxHankel2Boost**(*x* As *mpNum*, *ν* As *mpNum*) As *mpNum*

---

The function **cplxHankel2Boost** returns the Hankel function of the second kind  $H_\nu^{(2)}(x)$ .

**Parameters:**

*x*: A real number.

*ν*: A real number.

This routine returns the Hankel function of the second kind  $H_\nu^{(2)}(x)$ , defined as

$$H_\nu^{(2)}(x) = J_\nu(x) - iY_\nu(x). \quad (31.9.2)$$

### 31.9.3 Spherical Hankel Function of the First Kind

---

Function **cplxHankelSph1Boost**(*x* As *mpNum*, *ν* As *mpNum*) As *mpNum*

---

The function **cplxHankelSph1Boost** returns the spherical Hankel function of the first kind  $h_\nu^{(1)}(x)$ .

**Parameters:**

*x*: A real number.

*ν*: A real number.

This routine returns the spherical Hankel function of the first kind  $h_\nu^{(1)}(x)$ , defined as

$$h_\nu^{(1)} = \sqrt{\frac{\pi}{2x}}H_{\nu+\frac{1}{2}}^{(1)}(x). \quad (31.9.3)$$

### 31.9.4 Spherical Hankel Function of the Second Kind

---

Function **cplxHankelSph2Boost**(*x* As *mpNum*, *ν* As *mpNum*) As *mpNum*

---

The function **cplxHankelSph2Boost** returns the spherical Hankel function of the second kind  $h_\nu^{(2)}(x)$ .

**Parameters:**

*x*: A real number.

*ν*: A real number.

This routine returns the spherical Hankel function of the second kind  $h_\nu^{(2)}(x)$ , defined as

$$h_\nu^{(2)} = \sqrt{\frac{\pi}{2x}} H_{\nu+\frac{1}{2}}^{(2)}(x). \quad (31.9.4)$$

## 31.10 Airy Functions

In this section let  $z = (2/3)|x|^{3/2}$ . For large negative *x* the Airy functions and the Scorer function  $Gi(x)$  have asymptotic expansions oscillating with  $\cos(z+\pi/4)$  or  $\sin(z+\pi/4)$ , see Abramowitz and Stegun [1, 10.4.60, 10.4.64, 10.4.87]; therefore the phase information becomes totally unreliable for  $x < \sqrt[3]{2/|epsx|^{2/3}}$ , and the relative error increases strongly for *x* less than the square root.

### 31.10.1 Airy Function $\text{Ai}(x)$

---

Function **AiryAiBoost**(*x* As *mpNum*) As *mpNum*

---

The function **AiryAiBoost** returns the Airy function  $\text{Ai}(x)$ .

**Parameter:**

*x*: A real number.

The Airy function  $\text{Ai}(x)$  is defined as

$$\text{Ai}(x) = \frac{1}{\pi} \sqrt{\frac{x}{3}} K_{1/3}(z), \quad (x > 0) \quad (31.10.1)$$

$$\text{Ai}(x) = \frac{1}{3^{2/3} \Gamma(2/3)}, \quad (x = 0) \quad (31.10.2)$$

$$\text{Ai}(x) = \frac{1}{2} \sqrt{-x} \left( J_{1/3}(z) - \frac{1}{\sqrt{3}} Y_{1/3}(z) \right), \quad (x < 0) \quad (31.10.3)$$

### 31.10.2 Airy Function $\text{Ai}'(x)$

---

Function **AiryAiDerivativeBoost**(*x* As *mpNum*) As *mpNum*

---

The function **AiryAiDerivativeBoost** returns the Airy function  $\text{Ai}'(x)$ .

**Parameter:**

*x*: A real number.

This routine returns the Airy function  $\text{Ai}'(x)$ , defined as

$$\text{Ai}'(x) = \frac{x}{\pi \sqrt{3}} K_{2/3}(z), \quad (x > 0) \quad (31.10.4)$$

$$\text{Ai}'(x) = \frac{1}{-(3^{2/3})\Gamma(1/3)}, \quad (x = 0) \quad (31.10.5)$$

$$\text{Ai}'(x) = -\frac{x}{2} \left( J_{2/3}(z) + \frac{1}{\sqrt{3}} Y_{2/3}(z) \right), \quad (x < 0) \quad (31.10.6)$$

### 31.10.3 Airy Function $\text{Bi}(x)$

---

#### Function **AiryBiBoost**(*x* As mpNum) As mpNum

---

The function **AiryBiBoost** returns the Airy function  $\text{Bi}(x)$ .

**Parameter:**

*x*: A real number.

This routine returns the Airy function  $\text{Bi}(x)$ , defined as

$$\text{Bi}(x) = \sqrt{x} \left( \frac{2}{\sqrt{3}} I_{1/3}(z) + \frac{1}{\pi} K_{1/3}(z) \right), \quad (x > 0) \quad (31.10.7)$$

$$\text{Bi}(x) = \frac{1}{3^{1/6}\Gamma(2/3)}, \quad (x = 0) \quad (31.10.8)$$

$$\text{Bi}(x) = -\frac{1}{2}\sqrt{-x} \left( \frac{1}{\sqrt{3}} J_{1/3}(z) + Y_{1/3}(z) \right), \quad (x < 0) \quad (31.10.9)$$

### 31.10.4 Airy Function $\text{Bi}'(x)$

---

#### Function **AiryBiDerivativeBoost**(*x* As mpNum) As mpNum

---

The function **AiryBiDerivativeBoost** returns the Airy function  $\text{Bi}'(x)$ .

**Parameter:**

*x*: A real number.

This routine returns the Airy function  $\text{Bi}'(x)$ , defined as

$$\text{Bi}'(x) = x \left( \frac{2}{\sqrt{3}} I_{2/3}(z) + \frac{1}{\pi} K_{2/3}(z) \right), \quad (x > 0) \quad (31.10.10)$$

$$\text{Bi}(x) = \frac{3^{1/6}}{\Gamma(1/3)}, \quad (x = 0) \quad (31.10.11)$$

$$\text{Bi}(x) = -\frac{x}{2} \left( \frac{1}{\sqrt{3}} J_{2/3}(z) - Y_{2/3}(z) \right), \quad (x < 0) \quad (31.10.12)$$

## 31.11 Carlson-style Elliptic Integrals

The Carlson style elliptic integrals are a complete alternative group to the classical Legendre style integrals. They are symmetric and the numerical calculation is usually performed by duplication as described in [Carlson & Gustafson \(1994\)](#) and [Carlson \(1995\)](#).

### 31.11.1 Degenerate elliptic integral RC

---

Function **CarlsonRCBoost**(*x* As *mpNum*, *y* As *mpNum*) As *mpNum*

---

The function CarlsonRCBoost returns the value of the of Carlson's degenerate elliptic integral  $R_C$ .

**Parameters:**

*x*: A real number.

*y*: A real number.

This function computes the value of the of Carlson's degenerate elliptic integral  $R_C$  for  $x \geq 0$ ,  $y \neq 0$ :

$$R_C(x, y) = R_F(x, y, y) = \frac{1}{2} \int_0^\infty (t + x)^{-1/2} (t + y)^{-1} dt. \quad (31.11.1)$$

### 31.11.2 Integral of the 1st kind RF

---

Function **CarlsonRFBoost**(*x* As *mpNum*, *y* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function CarlsonRFBoost returns the value of the of Carlson's elliptic integral  $R_F$  of the first kind.

**Parameters:**

*x*: A real number.

*y*: A real number.

*z*: A real number.

This function computes the value of the of Carlson's elliptic integral  $R_F$  of the first kind

$$R_F(x, y, z) = \frac{1}{2} \int_0^\infty ((t + x)(t + y)(t + z))^{-1/2} dt. \quad (31.11.2)$$

with  $x, y, z \geq 0$ , at most one may be zero.

### 31.11.3 Integral of the 2nd kind RD

---

Function **CarlsonRDBoost**(*x* As *mpNum*, *y* As *mpNum*, *z* As *mpNum*) As *mpNum*

---

The function CarlsonRDBoost returns the value of the of Carlson's elliptic integral  $R_D$  of the second kind.

**Parameters:**

*x*: A real number.

*y*: A real number.

*z*: A real number.

This function computes the value of the of Carlson's elliptic integral  $R_D$  of the second kind

$$R_D(x, y, z) = R_J(x, y, z, z) = \frac{3}{2} \int_0^\infty ((t + x)(t + y))^{-1/2} (t + z)^{-3/2} dt. \quad (31.11.3)$$

with  $z > 0$ ,  $x, y \geq 0$ , at most one of  $x, y$  may be zero.

### 31.11.4 Integral of the 3rd kind $R_J$

---

Function **CarlsonRJBoost**(*x* As *mpNum*, *y* As *mpNum*, *z* As *mpNum*, *r* As *mpNum*) As *mpNum*

---

The function `CarlsonRJBoost` returns the value of the of Carlson's elliptic integral  $R_J$  of the third kind.

**Parameters:**

*x*: A real number.  
*y*: A real number.  
*z*: A real number.  
*r*: A real number.

This function computes the value of the of Carlson's elliptic integral  $R_J$  of the third kind

$$R_J(x, y, z, r) = \frac{3}{2} \int_0^\infty ((t + x)(t + y)(t + z))^{-1/2} (t + r)^{-1} dt. \quad (31.11.4)$$

with  $x, y, z \geq 0$ , at most one of may be zero, and  $r \neq 0$ .

## 31.12 Legendre-style Elliptic Integrals

### 31.12.1 Complete elliptic integral of the 1st kind

---

Function **CompleteLegendreEllint1Boost**(*k* As *mpNum*) As *mpNum*

---

The function `CompleteLegendreEllint1Boost` returns the value of the complete elliptic integral of the first kind.

**Parameter:**

*k*: A real number.

This function computes the value of the complete elliptic integral of the first kind  $K(k)$  with  $|k| < 1$

$$K(k) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - k^2 \sin^2 t}}. \quad (31.12.1)$$

### 31.12.2 Complete elliptic integral of the 2nd kind

---

Function **CompleteLegendreEllint2Boost**(*k* As *mpNum*) As *mpNum*

---

The function `CompleteLegendreEllint2Boost` returns the value of the complete elliptic integral of the second kind.

**Parameter:**

*k*: A real number.

This function computes the value of the complete elliptic integral of the second kind  $E(k)$  with  $|k| \leq 1$

$$E(k) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2 t}. \quad (31.12.2)$$

### 31.12.3 Complete elliptic integral of the 3rd kind

---

Function **CompleteLegendreEllint3Boost**( $\nu$  As *mpNum*,  $k$  As *mpNum*) As the value of the complete elliptic integral of the third kind.

---

The function **CompleteLegendreEllint3Boost** returns

**Parameters:**

$\nu$ : A real number.  
 $k$ : A real number.

This function computes the value of the complete elliptic integral of the third kind  $\Pi(\nu, k)$  with  $|k| < \nu \neq 1$

$$\Pi(\nu, k) = \int_0^{\pi/2} \frac{dt}{(1 - \nu \sin^2 t) \sqrt{1 - k^2 \sin^2 t}}. \quad (31.12.3)$$

### 31.12.4 Legendre elliptic integral of the 1st kind

---

Function **LegendreEllint1Boost**( $\phi$  As *mpNum*,  $k$  As *mpNum*) As *mpNum*

---

The function **LegendreEllint1Boost** returns the value of the incomplete Legendre elliptic integral of the first kind.

**Parameters:**

$\phi$ : A real number.  
 $k$ : A real number.

This function computes the value of the incomplete Legendre elliptic integral of the first kind

$$F(\phi, k) = \int_0^\phi \frac{dt}{\sqrt{1 - k^2 \sin^2 t}}. \quad (31.12.4)$$

with  $|k \sin \phi| \leq 1$ .

### 31.12.5 Legendre elliptic integral of the 2nd kind

---

Function **LegendreEllint2Boost**( $\phi$  As *mpNum*,  $k$  As *mpNum*) As *mpNum*

---

The function **LegendreEllint2Boost** returns the value of the incomplete Legendre elliptic integral of the second kind.

**Parameters:**

$\phi$ : A real number.  
 $k$ : A real number.

This function computes the value of the incomplete Legendre elliptic integral of the second kind

$$E(\phi, k) = \int_0^\phi \sqrt{1 - k^2 \sin^2 t} dt. \quad (31.12.5)$$

with  $|k \sin \phi| \leq 1$ .

### 31.12.6 Legendre elliptic integral of the 3rd kind

---

Function **LegendreEllint3Boost**( $\phi$  As mpNum,  $\nu$  As mpNum,  $k$  As mpNum) As mpNum

---

The function **LegendreEllint3Boost** returns the value of the incomplete Legendre elliptic integral of the third kind.

**Parameters:**

- $\phi$ : A real number.
- $\nu$ : A real number.
- $k$ : A real number.

This function computes the value of the incomplete Legendre elliptic integral of the third kind

$$\Pi(\phi, \nu, k) = \int_0^\phi \frac{dt}{(1 - \nu \sin^2 t) \sqrt{1 - k^2 \sin^2 t}}. \quad (31.12.6)$$

with  $|k \sin \phi| \leq 1$ .

## 31.13 Jacobi Elliptic Functions

These procedures return the Jacobi elliptic functions sn, cn, dn for argument  $x$  and complementary parameter  $m_c$ . A convenient implicit definition of the functions is

$$x = \int_0^{\text{sn}} \frac{dt}{\sqrt{(1 - t^2)(1 - k^2 t^2)}}, \quad \text{sn}^2 + \text{cn}^2 = 1, \quad k^2 \text{sn}^2 + \text{cn}^2 = 1 \quad (31.13.1)$$

with  $k^2 = 1 - m_c$ . There are a lot of equivalent definition of the Jacobi elliptic functions, e.g. with the Jacobi amplitude function (see e.g. [Olver et al. \(2010\)](#) [30, 22.16.11/12])

$$\begin{aligned} \text{sn}(x, k) &= \sin(\text{am}(x, k)), \\ \text{cn}(x, k) &= \cos(\text{am}(x, k)), \end{aligned}$$

or with Jacobi theta functions (cf. [\[Olver et al. \(2010\), 22.2\]](#)).

### 31.13.1 Jacobi elliptic function sn

---

Function **JacobiSNBoost**( $x$  As mpNum,  $k$  As mpNum) As mpNum

---

The function **JacobiSNBoost** returns the Jacobi elliptic function  $\text{sn}(x, k)$ .

**Parameters:**

- $x$ : A real number.
- $k$ : A real number.

### 31.13.2 Jacobi elliptic function cn

---

Function **JacobiCNBoost**( $x$  As mpNum,  $k$  As mpNum) As mpNum

---

The function **JacobiCNBoost** returns the Jacobi elliptic function  $\text{cn}(x, k)$ .

**Parameters:**

- $x$ : A real number.
- $k$ : A real number.

### 31.13.3 Jacobi elliptic function dn

---

Function **JacobiDNBoost**(*x* As *mpNum*, *k* As *mpNum*) As *mpNum*

---

The function **JacobiDNBoost** returns the Jacobi elliptic function  $\text{dn}(x, k)$ .

**Parameters:**

*x*: A real number.

*k*: A real number.

## 31.14 Zeta Functions

### 31.14.1 Riemann $\zeta(s)$ function

---

Function **RiemannZetaBoost**(*s* As *mpNum*) As *mpNum*

---

The function **RiemannZetaBoost** returns the Riemann zeta function.

**Parameter:**

*s*: A real number.

The Riemann zeta function  $\zeta(s)$  for  $s \neq 1$  is defined as

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}, \quad s > 1. \quad (31.14.1)$$

If  $s < 0$ , the reflection formula is used:

$$\zeta(s) = 2(2\pi)^{s-1} \sin\left(\frac{1}{2}\pi s\right) \Gamma(1-s) \zeta(1-s) \quad (31.14.2)$$

## 31.15 Exponential Integral and Related Integrals

### 31.15.1 Exponential Integral E1

---

Function **ExponentialIntegralE1Boost**(*x* As *mpNum*) As *mpNum*

---

The function **ExponentialIntegralE1Boost** returns the exponential integral  $E_1(x)$ .

**Parameter:**

*x*: A real number.

The exponential integral  $E_1(x)$  for  $x \neq 0$  is defined as

$$E_1(x) = \int_1^{\infty} \frac{e^{-xt}}{t} dt, \quad (31.15.1)$$

For  $x < 0$  the integral is calculated as  $E_1(x) = -\text{Ei}(-x)$ .

### 31.15.2 Exponential Integral Ei

---

Function **ExponentialIntegralEiBoost(x As mpNum) As mpNum**

---

The function **ExponentialIntegralEiBoost** returns the exponential integral  $Ei(x)$ .

**Parameter:**

*x*: A real number.

The exponential integral  $Ei(x)$  for  $x \neq 0$  is defined as

$$Ei(x) = -PV \int_{-x}^{\infty} \frac{e^{-t}}{t} dt = PV \int_{-\infty}^x \frac{e^t}{t} dt, \quad (31.15.2)$$

For  $x < 0$  the integral is calculated as  $Ei(x) = -E_1(-x)$ .

### 31.15.3 Exponential Integrals En

---

Function **ExponentialIntegralEnBoost(x As mpNum, n As mpNum) As mpNum**

---

The function **ExponentialIntegralEnBoost** returns the exponential integral  $E_n(x)$ .

**Parameters:**

*x*: A real number.

*n*: A real number.

The exponential integrals  $E_n(x)$  of integer order is defined as

$$E_n(x) = \int_1^{\infty} \frac{e^{-xt}}{t^n} dt, \quad (n \geq 0). \quad (31.15.3)$$

For  $x < 0$  the integral is calculated as  $Ei(x) = -E_1(-x)$ .

## 31.16 Basic Functions

---

Function **SinPiBoost(x As mpNum) As mpNum**

---

The function **SinPiBoost** returns the value of the sine of  $\pi x$ , with *x* in radians.

**Parameter:**

*x*: A real number.

---

Function **CosPiBoost(x As mpNum) As mpNum**

---

The function **CosPiBoost** returns the value of the cosine of  $\pi x$ , with *x* in radians.

**Parameter:**

*x*: A real number.

### 31.16.1 Auxiliary Function $\ln(1 + x)$

---

Function **Lnp1Boost(x As mpNum) As mpNum**

---

The function **Lnp1Boost** returns the value of the function  $\ln(1 + x)$ .

**Parameter:**

$x$ : A real number.

In Boost, this function is called Log1p.

### 31.16.2 Auxiliary Function $e^x - 1$

---

**Function `Expm1Boost(x As mpNum) As mpNum`**

---

The function Expm1Boost returns the value of the function  $\text{expm1}(x) = e^x - 1$ .

**Parameter:**

$x$ : A real number.

### 31.16.3 Cube Root: $\sqrt[3]{x}$

---

**Function `CbrtBoost(x As mpNum) As mpNum`**

---

The function CbrtBoost returns the absolute value of the cube root of  $x$ ,  $\sqrt[3]{x}$ .

**Parameter:**

$x$ : A real number.

### 31.16.4 Auxiliary Function $\sqrt{x + 1} - 1$

---

**Function `Sqrtp1m1Boost(x As mpNum) As mpNum`**

---

The function Sqrtp1m1Boost returns the value of  $\sqrt{x + 1} - 1$ .

**Parameter:**

$x$ : A real number.

### 31.16.5 Auxiliary Function $x^y - 1$

---

**Function `Powm1Boost(x As mpNum, y As mpNum) As mpNum`**

---

The function Powm1Boost returns the value of  $x^y - 1$ ,  $y \in \mathbb{R}$ .

**Parameters:**

$x$ : A real number.

$y$ : A real number.

### 31.16.6 Auxiliary Function $\sqrt{x^2 + y^2}$

---

**Function `HypotBoost(x As mpNum, y As mpNum) As mpNum`**

---

The function HypotBoost returns the value of  $\sqrt{x^2 + y^2}$ .

**Parameters:**

$x$ : A real number.

$y$ : A real number.

## 31.17 Sinus Cardinal Function and Hyperbolic Sinus Cardinal Functions

---

### Function **SincaBoost**(*x* As mpNum) As mpNum

---

The function SincaBoost returns the sinus cardinal function

**Parameter:**

*x*: A real number.

The sinus cardinal function is defined as

$$\text{sinc}_a(x) = \sin\left(\frac{\pi x}{a}\right) \frac{a}{\pi x} \quad (31.17.1)$$

### 31.17.1 Hyperbolic Sinus Cardinal: $\text{Sinhc}_a(x)$

---

### Function **SinhcaBoost**(*x* As mpNum) As mpNum

---

The function SinhcaBoost returns the hyperbolic sinus cardinal function.

**Parameter:**

*x*: A real number.

The hyperbolic sinus cardinal function is defined as

$$\text{sinhc}_a(x) = \sinh\left(\frac{\pi x}{a}\right) \frac{a}{\pi x} \quad (31.17.2)$$

## 31.18 Inverse Hyperbolic Functions

### 31.18.1 Hyperbolic Arc-cosine: $\text{acosh}(x)$

---

### Function **AcoshBoost**(*x* As mpNum) As mpNum

---

The function AcoshBoost returns the value of the hyperbolic arc-cosine of *x* in radians.

**Parameter:**

*x*: A real number.

### 31.18.2 Hyperbolic Arc-sine: $\text{asinh}(x)$

---

### Function **AsinhBoost**(*x* As mpNum) As mpNum

---

The function AsinhBoost returns the value of the hyperbolic arc-sine of *x* in radians.

**Parameter:**

*x*: A real number.

### 31.18.3 Hyperbolic Arc-tangent: $\operatorname{atanh}(x)$

---

Function **AtanhBoost**(*x* As *mpNum*) As *mpNum*

---

The function **AtanhBoost** returns the value of the hyperbolic arc-tangent of *x* in radians.

**Parameter:**

*x*: A real number.

# Chapter 32

## Distribution Functions

### 32.1 Introduction to Distribution Functions

This is a citation [Walck \(2007\)](#), and some more.

This is a citation [Van Hauwermeiren & Vose \(2009\)](#), and some more.

This is a citation [Rinne \(2008\)](#), and some more.

This is a citation [Johnson \*et al.\* \(1994.\)](#), and some more.

This is a citation [Johnson \*et al.\* \(1995.\)](#), and some more.

See also [Monahan \(2011\)](#)

See also [Lange \(2010\)](#)

See also [Chernick \(2008\)](#)

See also [Cheney & Kincaid \(2008\)](#)

#### 32.1.1 Continuous Distribution Functions

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  $p dx$ . The cumulative distribution function for the lower tail  $P(x)$  gives the probability of a variate taking a value less than  $x$ , and the cumulative distribution function for the upper tail  $Q(x)$  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.

#### 32.1.2 Discrete Distribution Functions

For discrete distributions the probability of sampling the integer value  $k$  is given by  $p(k)$ . The cumulative distribution for the lower tail  $P(k)$  of a discrete distribution is defined as the sum 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 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  $P(1) = p(1)$ ,  $Q(1) = 1 - p(1)$ .

### 32.1.3 Commonly Used Function Types

#### 32.1.3.1 Functions returning pdf, CDF, and related information

These functions have the form `?Dist(x; [Parameters], OutputString)`. Here  
 “?” is a placeholder for the name of the distribution,  
 “*x*” is the value for which we want to calculate the pdf, CDF etc,  
 “[Parameters;]” denote any parameters (like degrees of freedom) of the distribution, and  
 “OutputString” specifies the computed results which will be returned. This can be any of the following:

- **pdf**: the probability density function
- **P**: the cumulative distribution function (CDF)
- **Q**: the complement of cumulative distribution function (CDF)
- **logpdf**: the logarithm of the probability density function
- **logP**: the logarithm of the cumulative distribution function (CDF)
- **logQ**: the logarithm of the complement of cumulative distribution function (CDF)
- **h**: hazard function
- **H**: cumulative hazard function

As an example, for Student’s t-distribution, a “T” is used to specify the name of the distribution, and there is just one distribution parameter,  $\nu$ , the degrees of freedom. Therefore, the function has the form

`TDist(x As nmNum;  $\nu$  As mpNum, OutputString As String) As mpNumList`,

and an actual call to the function, requesting the pdf, CDF, and the complement of the CDF for  $x = 2.3$  and  $\nu = 22$  could be

---

```
Result = TDist(2.3, 22, "pdf + P + Q")
mp.Print Result
```

---

which produces the output

```
pdf: 0.434234342343434
P: 0.943453463453453
Q: 0.054564564564236
```

### 32.1.3.2 Functions returning Quantiles

These functions have the form `?DistInv(Prob; [Parameters;], OutputString)`. Here  
 “?” is a placeholder for the name of the distribution,  
 “Prob” sets the target values for  $P$  and  $Q$ ,  
 “[Parameters;]” denote any parameters (like degrees of freedom) of the distribution, and  
 “OutputString” specifies the computed results which will be returned. This can be any of the following:

- **PInv**: the inverse of the cumulative distribution function (CDF). For discrete distribution, this will be outwardly rounded
- **QInv**: the inverse of the complement of the cumulative distribution function (CDF). For discrete distribution, this will be outwardly rounded
- **P**: the value of the cumulative distribution function (CDF), which has actually been achieved
- **Q**: the value of the complement of the cumulative distribution function (CDF), which has actually been achieved

As an example, for Student’s t-distribution, a “T” is used to specify the name of the distribution, and there is just one distribution parameter,  $\nu$ , the degrees of freedom. Therefore, the function has the form

`TDistInv(Prob As mpNum;  $\nu$  As mpNum, OutputString As String) As mpNumList`,

and an actual call to the function, requesting the inverse of the complement of the CDF for  $Prob = 0.01$  and  $\nu = 22$  could be

---

```
Result = TDistInv(0,01, 22, "QInv")
mp.Print Result
```

---

which produces the output

`QInv: 2.943453463453453`

### 32.1.3.3 Functions returning moments and related information

These functions have the form `?DistInfo([Parameters;], OutputString)`. Here “?” is a placeholder for the name of the distribution, “[Parameters;]” denote any parameters (like degrees of freedom) of the distribution, and “OutputString” specifies the computed results which will be returned. This can be any of the following:

- **range**: Returns the valid range of the random variable over distribution dist.
- **support**:
- **mode**: Returns the mode of the distribution dist. This function may return a `domain_error` if the distribution does not have a defined mode.
- **median**: Returns the median of the distribution dist.
- **mean**: Returns the mean of the distribution dist. This function may return a `domain_error` if the distribution does not have a defined mean (for example the Cauchy distribution).
- **stdev**: Returns the standard deviation of distribution dist. This function may return a `domain_error` if the distribution does not have a defined standard deviation.
- **variance**: Returns the variance of the distribution dist. This function may return a `domain_error` if the distribution does not have a defined variance.
- **skewness**: Returns the skewness of the distribution dist. This function may return a `domain_error` if the distribution does not have a defined skewness.
- **kurtosis**: Returns the ‘proper’ kurtosis (normalized fourth moment) of the distribution dist.
- **kurtosis excess**: Returns the kurtosis excess of the distribution dist.  $\text{kurtosis excess} = \text{kurtosis} - 3$

As an example, for Student’s t-distribution, a “T” is used to specify the name of the distribution, and there is just one distribution parameter,  $\nu$ , the degrees of freedom. Therefore, the function has the form

`TDistInfo( $\nu$  As mpNum, OutputString As String) As mpNumList,`

and an actual call to the function, requesting the mean, variance, skewness and kurtosis with  $\nu = 22$  could be

---

```
Result = TDistInfo(22, "mean + variance + skewness + kurtosis")
mp.Print Result
```

---

which produces the output

```
mean: 0.434234342343434
variance: 0.943453463453453
skewness: 0.054564564564236
kurtosis: 0.6054564564564236
```

### 32.1.3.4 Functions returning Sample Size estimates

These functions have the form `?SampleSize(Alpha; Beta; ModifiedNoncentrality; [Parameters;], OutputString)`. Here

”?” is a placeholder for the name of the distribution,  
”Alpha” specifies the confidence level (or Type I error),

”Beta” specifies the Type I error (or  $1 - \text{Power}$ ),

”ModifiedNoncentrality” specifies the (modified) noncentrality parameter of the distribution in a form which does not depend on sample size (which may require a modification compared to the conventional form for stating the noncentrality parameter),

”[Parameters;]” denote any additional parameters of the distribution (if any) which are not a function of the sample size, and

”OutputString” specifies the computed results which will be returned. This can be any of the following:

- **ExactN**: returns an ”exact”, i.e. typically non-integer sample size estimate
- **UpperN**: upper integer sample size estimate
- **LowerN**: lower integer sample size estimate
- **UpperNPower**: actual power when using UpperN
- **LowerNPower**: actual power when using LowerN

As an example, for the noncentral t-distribution, the prefix ”NoncentralT” is used to specify the name of the distribution. The distribution parameter  $\nu$ , the degrees of freedom, which depends on the sample size, and is therefore not included in the parameter list of this function. The modified noncentrality parameter is called  $\tilde{\rho} = \Delta/\sigma$ . Therefore, the function has the form

`NoncentralTSampleSize(α As mpNum, β As mpNum, ρ̃ As mpNum, OutputString As String) As mpNumList`

and an actual call to the function, requesting an upper sample size estimate (and actual power) for  $\alpha = 0.95$ ,  $\beta = 0.1$ , and  $\tilde{\rho} = \Delta/\sigma = 0.6$  would be

---

```
Result = NoncentralTSampleSize(0.95, 0.1, 0.6, "UpperN + UpperNPower")
mp.Print Result
```

---

which produces the output

```
UpperN: 26
UpperNPower: 0.92435435
```

### 32.1.3.5 Functions related to noncentrality parameters

These functions have the form `?Noncentrality(Alpha; Noncentrality; [Parameters;], OutputString)`. Here

”?” is a placeholder for the name of the distribution,  
 ”Alpha” specifies the confidence level (or Type I error),  
 ”Noncentrality” specifies the noncentrality parameter of the distribution,  
 ”[Parameters;]” denote any additional parameters of the distribution, and  
 ”OutputString” specifies the computed results which will be returned. This can be any of the following:

- **UpperCI**: upper confidence interval
- **LowerCI**: lower confidence interval
- **TwoSidedCI**: two-sided confidence interval

As an example, for the noncentral t-distribution, the prefix ”NoncentralT” is used to specify the name of the distribution. The noncentrality parameter is  $\delta$ , and the other distribution parameter is  $\nu$ , the degrees of freedom. Therefore, the function has the form

`NoncentralTNoncentrality(α As mpNum, δ As mpNum, ν As mpNum, OutputString As String) As mpNumList`

and an actual call to the function, requesting an upper confidence interval for  $\delta$  with  $\alpha = 0.95$ ,  $\delta = 0.6$  and  $\nu = 22$  would be

---

```
Result = NoncentralTNoncentrality(0.95, 0.6, 22, "UpperCI")
mp.Print Result
```

---

which produces the output

`UpperCI: 0.7546534`

### 32.1.3.6 Functions returning Random numbers

These functions have the form `?DistRan(Size; [Parameters;], Generator, OutputString)`. Here “?” is a placeholder for the name of the distribution, “Size” specifies the size of the random sample, “[Parameters;]” denote any parameters (like degrees of freedom) of the distribution, and “Generator” specifies the pseudo random generator which will be used to produce the random sample, “OutputString” specifies the computed results which will be returned. This can be any of the following:

- **Unsorted**: produces unsorted output
- **Ascending**: output sorted in ascending order
- **Descending**: output sorted in descending order
- **Histogram**( $k$ ): output grouped in histogram format, with  $k$  buckets
- **HistogramCDF**( $k$ ): cumulated output grouped in histogram format, with  $k$  buckets

As an example, for Student’s t-distribution, a “T” is used to specify the name of the distribution, and there is just one distribution parameter,  $\nu$ , the degrees of freedom. Therefore, the function has the form

`TDistRan(Size As Integer;  $\nu$  As mpNum, Generator As String, OutputString As String) As mpNumList,`

and an actual call to the function, requesting a random sample of  $Size = 10000$  of a t-distribution with  $\nu = 22$ , using the default pseudo-random number generator, sorting output in ascending order could be

---

```
Result = TDistRan(10000, 22, "Default", "Ascending")
mp.Plot Result
```

---

which produces the output

`QInv: 2.943453463453453`

## 32.2 Beta-Distribution

### 32.2.1 Definition

If  $X_1$  and  $X_2$  are independent random variables following  $\chi^2$ -distribution with  $2a$  and  $2b$  degrees of freedom respectively, then the distribution of the ratio  $\frac{X_1}{X_1+X_2}$  is said to follow a Beta-distribution with  $a$  and  $b$  degrees of freedom.

See [Tretter & Walster \(1979\)](#)

### 32.2.2 Density and CDF

---

Function **BetaDist**(*x* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function BetaDist returns returns pdf, CDF and related information for the central Beta-distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.2.2.1 and 32.2.2.2.

#### 32.2.2.1 Density

The pdf of a variable following a central Beta-distribution with *a* and *b* degrees of freedom is given by

$$f_{\text{Beta}}(a, b, x) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1} \quad (32.2.1)$$

where  $B(a, b)$  denotes the beta function (see section ??).

#### 32.2.2.2 CDF: General formulas

The cdf of a variable following a central Beta-distribution with *a* and *b* degrees of freedom is given by

$$\Pr [X \leq x] = F_{\text{Beta}}(a, b, x) = \int_0^x f_{\text{Beta}}(a, b, t) dt \quad (32.2.2)$$

#### 32.2.2.3 Exact cdf as continued fraction

The following representation as continued fraction is used (Peizer 1968, .1428 and 1452):

$$I(a, b, x) = \binom{n}{a} p^{b-1} q^a \frac{1}{(1 + u_1/(v_1 + u_2/(v_2 + u_3/(v_3 + \dots)))), \quad \text{where}} \quad (32.2.3)$$

$$\begin{aligned} p &= (1-x), & q &= x, & n &= a+b-1, & u_1 &= \frac{-(b-1)q}{p}, & u_{2j} &= \frac{j(n+j)q}{p}, \\ u_{2j+1} &= \frac{-(a+j)(b-j-1)q}{p}, & v_j &= a+j, & j &= 1, 2, \dots \end{aligned}$$

### 32.2.3 Quantiles

---

Function **BetaDistInv**(*Prob* As mpNum, *m* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function BetaDistInv returns returns quantiles and related information for the the central Beta-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

### 32.2.4 Properties

---

#### Function **BetaDistInfo(a As mpNum, b As mpNum, Output As String) As mpNumList**

---

The function BetaDistInfo returns returns moments and related information for the central Beta-distribution

**Parameters:**

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

#### 32.2.4.1 Moments: algorithms and formulas

The raw moments are given by:

$$E^h(W) = \frac{\Gamma(a + h)\Gamma(a + b)}{\Gamma(a)\Gamma(a + b + h)} \quad (32.2.4)$$

The raw moments of the power of a beta variable are given by:

$$E^h(W^s) = \frac{\Gamma(a + hs)\Gamma(a + b)}{\Gamma(a)\Gamma(a + b + hs)} \quad (32.2.5)$$

#### 32.2.4.2 Recurrences

$$I(a, b; x) = 1 - I(b, a; 1 - x) \quad (32.2.6)$$

$$I(a, b; x) = \binom{n}{a} x^a (1 - x)^{b-1} + I(a + 1, b - 1; x) \quad (32.2.7)$$

$$I(a, b; x) = \binom{n}{a} x^a (1 - x)^b + I(a + 1, b; x) \quad (32.2.8)$$

$$I(a, b + 1; x) = \binom{n}{a} x^a (1 - x)^b + I(a, b; x) \quad (32.2.9)$$

$$I(a, b; x) = \binom{n}{a + b} x^a (1 - x)^b \frac{a}{a + b - x} + I(a + 1, b + 1; x) \quad (32.2.10)$$

$$I(a, b; x) = F\left(2a, 2b, \frac{nx}{m - mx}\right) \quad (32.2.11)$$

### 32.2.5 Random Numbers

---

Function **BetaDistRandom**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

---

The function BetaDistRandom returns random numbers following a central Beta-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

#### 32.2.5.1 Random Numbers: algorithms and formulas

In order to obtain random numbers from a Beta distribution we first single out a few special cases. For  $p = 1$  and/or  $q = 1$  we may easily solve the equation  $F(x) = \xi$  where  $F(x)$  is the cumulative function and  $\xi$  a uniform random number between zero and one. In these cases

$$\begin{aligned} p = 1 \Rightarrow x &= 1 - \xi^{1/q} \\ q = 1 \Rightarrow x &= \xi^{1/q} \end{aligned}$$

For  $p$  and  $q$  half-integers we may use the relation to the chi-square distribution by forming the ratio  $\frac{y_m}{y_m + y_n}$  with  $y_m$  and  $y_n$  two independent random numbers from chi-square distributions with  $m = 2p$  and  $n = 2q$  degrees of freedom, respectively.

Yet another way of obtaining random numbers from a Beta distribution valid when  $p$  and  $q$  are both integers is to take the  $l^{th}$  out of  $k$  ( $1 \leq l \leq k$ ) independent uniform random numbers between zero and one (sorted in ascending order). Doing this we obtain a Beta distribution with parameters  $p = l$  and  $q = k + 1 - l$ . Conversely, if we want to generate random numbers from a Beta distribution with integer parameters  $p$  and  $q$  we could use this technique with  $l = p$  and  $k = p + q - 1$ . This last technique implies that for low integer values of  $p$  and  $q$  simple code may be used, e.g. for  $p = 2$  and  $q = 1$  we may simply take  $\max(\xi_1, \xi_2)$  i.e. the maximum of two uniform random numbers (Walck, 2007).

## 32.3 Binomial Distribution

These functions return PMF and CDF of the (discrete) binomial distribution with number of trials  $n \geq 0$  and success probability  $0 \leq p \leq 1$ .

### 32.3.1 Density and CDF

---

Function **BinomialDist**(*x* As *mpNum*, *n* As *mpNum*, *p* As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function BinomialDist returns random numbers following a central Binomial-distribution

**Parameters:**

*x*: The number of successes in trials.  
*n*: The number of independent trials.  
*p*: The probability of success on each trial  
*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.3.1.1 and 32.3.1.2.

**32.3.1.1 Density**

$$f_{\text{Bin}}(n, k; p) = \binom{n}{k} p^k (1-p)^{n-k} = f_{\text{Beta}}(k+1, n-k+1, p)/(n+1) \quad (32.3.1)$$

**32.3.1.2 CDF**

$$F_{\text{Bin}}(n, k; p) = I_{1-p}(n-k, k+1) = \text{ibeta}(n-k, k+1, 1-p) \quad (32.3.2)$$

**32.3.2 Quantiles**


---

Function **BinomialDistInv**(*Prob* As mpNum, *n* As mpNum, *p* As mpNum, *Output* As String) As mpNumList

---

The function BinomialDistInv returns returns quantiles and related information for the the central binomial-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.  
*n*: The number of Bernoulli trials.  
*p*: The probability of a success on each trial.  
*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

**32.3.3 Properties**


---

Function **BinomialDistInfo**(*n* As mpNum, *p* As mpNum, *Output* As String) As mpNumList

---

The function BinomialDistInfo returns returns moments and related information for the central Binomial-distribution

**Parameters:**

*n*: The number of Bernoulli trials.  
*p*: The probability of a success on each trial.  
*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

### 32.3.3.1 Moments: algorithms and formulas

$$\mu'_r = \sum_{i=0}^r \binom{n}{i} \left( \sum_{j=0}^i \binom{i}{j} (-1)^j (i-j)^r \right) \quad (32.3.3)$$

$$\mu_1 = np \quad (32.3.4)$$

$$\mu_2 = np(1-p) = npq \quad (32.3.5)$$

$$\mu_3 = npq(q-p) \quad (32.3.6)$$

$$\mu_4 = 3(npq)^3 + npq(1-6pq) \quad (32.3.7)$$

### 32.3.4 Random Numbers

---

Function **BinomialDistRandom**(*Size* As *mpNum*, *n* As *mpNum*, *p* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **BinomialDistRandom** returns random numbers following a central Binomial-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: The number of Bernoulli trials.

*p*: The probability of a success on each trial.

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

### 32.3.4.1 Random Numbers: algorithms and formulas

In order to obtain random numbers from a Binomial distribution we first single out a few special cases. For  $p = 1$  and/or  $q = 1$  we may easily solve the equation  $F(x) = \xi$  where  $F(x)$  is the cumulative function and  $\xi$  a uniform random number between zero and one. In these cases

$$\begin{aligned} p = 1 \Rightarrow x &= 1 - \xi^{1/q} \\ q = 1 \Rightarrow x &= \xi^{1/q} \end{aligned}$$

For  $p$  and  $q$  half-integers we may use the relation to the chi-square distribution by forming the ratio  $\frac{y_m}{y_m + y_n}$  with  $y_m$  and  $y_n$  two independent random numbers from chi-square distributions with  $m = 2p$  and  $n = 2q$  degrees of freedom, respectively.

Yet another way of obtaining random numbers from a Beta distribution valid when  $p$  and  $q$  are both integers is to take the  $l^{th}$  out of  $k$  ( $1 \leq l \leq k$ ) independent uniform random numbers between zero and one (sorted in ascending order). Doing this we obtain a Beta distribution with parameters  $p = l$  and  $q = k + 1 - l$ . Conversely, if we want to generate random numbers from a Beta distribution with integer parameters  $p$  and  $q$  we could use this technique with  $l = p$  and  $k = p + q - 1$ . This last technique implies that for low integer values of  $p$  and  $q$  simple code may be used, e.g. for  $p = 2$  and  $q = 1$  we may simply take  $\max(\xi_1, \xi_2)$  i.e. the maximum of two uniform random numbers (Walck, 2007).

## 32.4 Chi-Square Distribution

### 32.4.1 Definition

Let  $X_1, X_2, \dots, X_n$  be independent and identically distributed random variables each following a normal distribution with mean zero and unit variance. Then  $\chi^2 = \sum_{j=1}^n X_j$  is said to follow a  $\chi^2$ -distribution with  $n$  degrees of freedom.

### 32.4.2 Density and CDF

---

Function **CDist**(*x* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function **CDist** returns pdf, CDF and related information for the central  $\chi^2$ -distribution

**Parameters:**

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.4.2.1 and 32.4.2.2.

#### 32.4.2.1 Density

The density of a central chi-square variable with  $n$  degrees of freedom is given by

$$f_{\chi^2}(n, x) = \frac{1}{2^{n/2}\Gamma(n/2)} x^{(n-2)/2} e^{-x/2}. \quad (32.4.1)$$

#### 32.4.2.2 CDF: General formulas

The cdf of a central chi-square variable with  $n$  degrees of freedom is given by

$$\Pr [\chi^2 \leq x] = F_{\chi^2}(n, x) = \int_0^x f_{\chi^2}(n, t) dt \quad (32.4.2)$$

#### 32.4.2.3 CDF: Continued fraction

For real  $n > 0$ , the CDF can be calculated using continued fraction (Peizer & Pratt, 1968).

If  $(n-1) \leq x$  let  $1 - F_{\chi^2}(n, x)$  be a right tail chi square probability. Then

$$1 - F_{\chi^2}(n, x) = f_{\chi^2}(n, x) \frac{1}{(1 + u_1/(v_1 + u_2/(v_2 + u_3/(v_3 + \dots))))} \quad (32.4.3)$$

where  $M = \frac{1}{2}x$ ,  $b = \frac{1}{2}n$ ,  $u_{2j-1} = j - b$ ,  $v_{2j-1} = M$ ,  $u_{2j} = j$ ,  $v_{2j} = 1$ ,  $j = 1, 2, \dots$

If  $(n-1) > x$  let  $F_{\chi^2}(n, x)$  be a left tail chi square probability. Then

$$F_{\chi^2}(n, x) = f_{\chi^2}(n, x) \frac{m}{b} \frac{1}{(1 + u_1/(v_1 + u_2/(v_2 + u_3/(v_3 + \dots))))} \quad (32.4.4)$$

where  $M = \frac{1}{2}x$ ,  $b = \frac{1}{2}n$ ,  $u_1 = -M$ ,  $u_{2j} = jM$ ,  $u_{2j+1} = -(b+j)M$ ,  $v_j = b+j$ ,  $j = 1, 2, \dots$

### 32.4.3 Quantiles

---

Function **CDistInv**(*Prob* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function **CDistInv** returns quantiles and related information for the the central  $\chi^2$ -distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section [32.1.3.2](#) for the options for *Prob* and *Output*).

### 32.4.4 Properties

---

Function **CDistInfo**(*n* As mpNum, *Output* As String) As mpNumList

---

The function **CDistInfo** returns moments and related information for the central  $\chi^2$ -distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section [32.1.3.3](#) for the options for *Output*. Algorithms and formulas are given in section [32.4.4](#).

### 32.4.5 Random Numbers

---

Function **CDistRan**(*Size* As mpNum, *n* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function **CDistRan** returns random numbers following a central  $\chi^2$ -distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section [32.4.5](#).

As we saw above the sum of *n* independent standard normal random variables gave a chi-square distribution with *n* degrees of freedom. This may be used as a technique to produce pseudorandom numbers from a chi-square distribution. This required a generator for standard normal random numbers and may be quite slow. However, if we make use of the Box-Muller transformation in order to obtain the standard normal random numbers we may simplify the calculations. Adding *n* such squared random numbers implies that

$$y_{2k} = -2 \ln(\xi_1 \cdot \xi_2 \cdot \dots \cdot \xi_k)$$

$$y_{2k+1} = -2 \ln(\xi_1 \cdot \xi_2 \cdot \dots \cdot \xi_k) - 2 \ln(\xi_{k+1})[\cos(2\pi\xi_{k+2})]^2$$

for  $k$  a positive integer will be distributed as chi-square variable with even or odd number of degrees of freedom. In this manner a lot of unnecessary operations are avoided. Since the chi-square distribution is a special case of the Gamma distribution we may also use a generator for this distribution.

### 32.4.6 Wishart Matrix

See [Gleser \(1976\)](#)

## 32.5 Exponential Distribution

These functions return PDF, CDF, and ICDF of the exponential distribution with location  $a$ , rate  $\alpha > 0$ , and the support interval  $(a, +\infty)$  :

### 32.5.1 Density and CDF

---

Function **ExponentialDist**(*x* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **ExponentialDist** returns returns pdf, CDF and related information for the central Exponential distribution

**Parameters:**

*x*: The value of the distribution.

*lambda*: The parameter of the distribution.

*Output*: A string describing the output choices

See section [32.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [32.2.2.1](#) and [32.2.2.2](#).

#### 32.5.1.1 Density

$$f(x) = \alpha \exp(-\alpha(x - a)) \quad (32.5.1)$$

#### 32.5.1.2 CDF

$$F(x) = 1 - \exp(-\alpha(x - a)) = \text{expm1}(-\alpha(x - a)) \quad (32.5.2)$$

### 32.5.2 Quantiles

---

Function **ExponentialDistInv**(*Prob* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **ExponentialDistInv** returns returns quantiles and related information for the the central Exponential distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*lambda*: The number of Bernoulli trials.

*Output*: A string describing the output choices

See section [32.1.3.2](#) for the options for *Prob* and *Output*).

$$F^{-1}(y) = a - \ln 1p(-y)/\alpha \quad (32.5.3)$$

### 32.5.3 Properties

---

#### Function **ExponentialDistInfo**(*lambda* As mpNum, *Output* As String) As mpNumList

---

The function **ExponentialDistInfo** returns returns moments and related information for the central *t*-distribution

**Parameters:**

*lambda*: A real number greater 0, representing the parameter of the distribution

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

#### 32.5.3.1 Moments and cumulants

The mean or expected value of an exponentially distributed random variable *X* with rate parameter  $\lambda$  is given by

$$E[X] = \frac{1}{\lambda} \quad (32.5.4)$$

The variance of *X* is given by

$$E[X^2] = \frac{1}{\lambda^2} \quad (32.5.5)$$

so the standard deviation is equal to the mean.

The moments of *X*, for  $n = 1, 2, \dots$ , are given by

$$E[X^n] = \frac{n!}{\lambda^n} \quad (32.5.6)$$

### 32.5.4 Random Numbers

---

#### Function **ExponentialDistRandom**(*Size* As mpNum, *lambda* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function **ExponentialDistRandom** returns returns random numbers following a central Beta-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*lambda*: A real number greater 0, representing the numerator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 32.6.5.

#### 32.5.4.1 Random Numbers: algorithms and formulas

Random numbers can be generated using the inversion formula.

## 32.6 Fisher's F-Distribution

### 32.6.1 Definition

If  $X_1$  and  $X_2$  are independent random variables following  $\chi^2$ -distribution with  $m$  and  $n$  degrees of freedom respectively, then the distribution of the ratio  $F = \frac{X_1/m}{X_2/n}$  is said to follow a F-distribution with  $m$  and  $n$  degrees of freedom.

### 32.6.2 Density and CDF

---

Function **FDist**(*x* As mpNum, *m* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function FDist returns returns pdf, CDF and related information for the central  $F$ -distribution

**Parameters:**

*x*: A real number

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.6.2.1 and 32.6.2.2.

#### 32.6.2.1 Density

The density of a variable following a central F-distribution with  $m$  and  $n$  degrees of freedom is given by

$$f_F(m, n, x) = \frac{m^{m/2} n^{n/2}}{B(m/2, n/2)} x^{(m-2)/2} (n + mx)^{-(m+n)/2} \quad (32.6.1)$$

#### 32.6.2.2 CDF: General formulas

The cdf of a variable following a central F-distribution with  $m$  and  $n$  degrees of freedom is given by

$$\Pr[X \leq x] = F_F(m, n, x) = \int_0^x f(m, n, t) dt \quad (32.6.2)$$

### 32.6.3 Quantiles

---

Function **FDistInv**(*Prob* As mpNum, *m* As mpNum, *n* As mpNum) As mpNumList

---

The function FDistInv returns returns quantiles and related information for the the central  $t$ -distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Output*? String? A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

### 32.6.4 Properties

---

Function **FDistInfo**(*m* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function FDistInfo returns returns moments and related information for the central *t*-distribution

**Parameters:**

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.6.4.

### 32.6.5 Random Numbers

---

Function **FDistRan**(*Size* As mpNum, *m* As mpNum, *n* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function FDistRan returns returns random numbers following a central *F*-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 32.6.5.

#### 32.6.5.1 Random Numbers: algorithms and formulas

Following the definition the quantity  $F = \frac{y_m/m}{y_n/n}$  where  $y_n$  and  $y_m$  are two variables distributed according to the chi-square distribution with *n* and *m* degrees of freedom respectively follows the *F*-distribution. We may thus use this relation inserting random numbers from chi-square distributions (see section ...).

## 32.7 Gamma (and Erlang) Distribution

These functions return PDF, CDF, and ICDF of the gamma distribution with shape  $a > 0$ , scale  $b > 0$ , and the support interval  $(0, +\infty)$ .

A gamma distribution with shape  $a \in \mathbb{N}$  is called Erlang distribution.

### 32.7.1 Density and CDF

---

Function **GammaDist**(*x* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function GammaDist returns returns pdf, CDF and related information for the central Gamma-distribution

**Parameters:***x*: A real number*a*: A real number greater 0, a parameter to the distribution*b*: A real number greater 0, a parameter to the distribution*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.2.2.1 and 32.2.2.2.

**32.7.1.1 Density**

$$f(x; a, b) = \frac{x^{a-1} e^{-x/b}}{\Gamma(a) b^a} \quad (32.7.1)$$

**32.7.1.2 CDF: General formulas**

$$F(x; a, b) = P(a, x/b) = igammap(a, x/b) \quad (32.7.2)$$

**32.7.2 Quantiles**

---

**Function `GammaDistInv(Prob As mpNum, m As mpNum, n As mpNum) As mpNumList`**

---

The function `GammaDistInv` returns returns quantiles and related information for the the central Gamma-distribution

**Parameters:***Prob*: A real number between 0 and 1.*m*: A real number greater 0, a parameter to the distribution*n*: A real number greater 0, a parameter to the distribution  
*Output*? String? A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

$$F^{-1}(y) = b \cdot igammapInv(a, y) \quad (32.7.3)$$

**32.7.3 Properties**

---

**Function `GammaDistInfo(a As mpNum, b As mpNum) As mpNumList`**

---

The function `GammaDistInfo` returns returns moments and related information for the central Gamma-distribution

**Parameters:***a*: A real number greater 0, representing the degrees of freedom*b*: A real number greater 0, representing the degrees of freedom  
*Output*? String? A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

### 32.7.3.1 Moments

The algebraic moments are given by (Wolfram)

$$\mu'_r = \frac{b^r \Gamma(a + r)}{\Gamma(a)} \quad (32.7.4)$$

## 32.7.4 Random Numbers

---

Function **GammaDistRandom**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

---

The function **GammaDistRandom** returns random numbers following a central Beta-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, a parameter to the distribution

*b*: A real number greater 0, a parameter to the distribution

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

### 32.7.4.1 Random Numbers: algorithms and formulas

In the case of an Erlangian distribution (*b* a positive integer) we obtain a random number by adding *b* independent random numbers from an exponential distribution i.e.

$$x = -\ln(\xi_1 \cdot \xi_2 \cdot \dots \cdot \xi_b)/a$$

where all the  $\xi_i$  are uniform random numbers in the interval from zero to one. Note that care must be taken if *b* is large in which case the product of uniform random numbers may become zero due to machine precision. In such cases simply divide the product in pieces and add the logarithms afterwards.

### 32.7.4.2 General case

In a more general case we use the so called Johnk's algorithm

1. Denote the integer part of *b* with *i* and the fractional part with *f* and put *r* = 0. Let  $\xi$  denote uniform random numbers in the interval from zero to one.
2. If *i* > 0 then put  $r = -\ln(\xi_1 \cdot \xi_2 \cdot \dots \cdot \xi_i)$ .
3. If *f* = 0 then go to 7.
4. Calculate  $w_1 = \xi_{i+1}^{1/f}$  and  $w_2 = \xi_{i+2}^{1/(1-f)}$ .
5. If  $w_1 + w_2 > 1$  then go back to iv.
6. Put  $r = r - \ln(\xi_{i+3}) \cdot \frac{w_1}{w_1 + w_2}$ .
7. Quit with  $r = r/a$ .

## 32.8 Hypergeometric Distribution

See [Upton \(1982\)](#), [Harkness & Katz \(1964\)](#)

See [Ling & Pratt \(1984\)](#)

See [Knüsel & Michalk \(1987\)](#)

See also [Conlon & Thomas \(1993\)](#)

See also [Casagrande \*et al.\* \(1978\)](#)

### 32.8.1 Definition

These functions return PMF and CDF of the (discrete) hypergeometric distribution; the PMF gives the probability that among  $n$  randomly chosen samples from a container with  $n_1$  type1 objects and  $n_2$  type2 objects there are exactly  $k$  type1 objects.

### 32.8.2 Density and CDF

---

Function **HypergeometricDist**(*x* As mpNum, *n* As mpNum, *M* As mpNum, *N* As mpNum, *Output* As String) As mpNumList

---

The function **HypergeometricDist** returns returns pdf, CDF and related information for the central hypergeometric distribution

#### Parameters:

*x*: The number of successes in the sample.

*n*: The size of the sample.

*M*: The number of successes in the population

*N*: The population size

*Output*: A string describing the output choices

See section [32.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [32.2.2.1](#) and [32.2.2.2](#).

#### 32.8.2.1 Density

$$f(k) = \frac{\binom{n_1}{k} \binom{n_2}{n-k}}{\binom{n_1+n_2}{n}}, \quad (n, n_1, n_2 \geq 0; n \leq n_1 + n_2). \quad (32.8.1)$$

$f(k)$  is computed with the R trick [39], which replaces the binomial coefficients by binomial PMFs with  $p = n/(n_1+n_2)$ .

#### 32.8.2.2 CDF

There is no explicit formula for the CDF, it is calculated as  $\sum f(i)$ , using the lower tail if  $k < nn_1/(n_1+n_2)$  and the upper tail otherwise with one value of the PMF and the recurrence formulas:

$$f(k+1) = \frac{(n_1 - k)(n - k)}{(k + 1)(n_2 - n + k + 1)} f(k) \quad (32.8.2)$$

$$f(k-1) = \frac{k(n_2 - n + k)}{(n_1 - k + 1)(n - k + 1)} f(k) \quad (32.8.3)$$

### 32.8.3 Quantiles

---

Function **HypergeometricDistInv**(*Prob* As mpNum, *n* As mpNum, *M* As mpNum, *N* As mpNum, *Output* As String) As mpNumList

---

The function **HypergeometricDistInv** returns quantiles and related information for the the central hypergeometric distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: The size of the sample.

*M*: The number of successes in the population

*N*: The population size

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

### 32.8.4 Sample Size

See [Guenther \(1974\)](#)

### 32.8.5 Properties

---

Function **HypergeometricDistInfo**(*n* As mpNum, *M* As mpNum, *N* As mpNum, *Output* As String) As mpNumList

---

The function **HypergeometricDistInfo** returns moments and related information for the central hypergeometric distribution

**Parameters:**

*n*: The size of the sample.

*M*: The number of successes in the population

*N*: The population size

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

#### 32.8.5.1 Moments

$$\mu_1 = nP \tag{32.8.4}$$

$$\mu_2 = nPQ \frac{N-n}{N-1} \tag{32.8.5}$$

$$\mu_3 = nPQ(Q-P) \frac{(N-n)(N-2n)}{(N-1)(N-2)} \tag{32.8.6}$$

$$\kappa_4 = \frac{6nP^2Q^2(N-n)}{N-1} \frac{n(N-n)(5N-6) - N(N-1)}{(N-2)(N-3)} \tag{32.8.7}$$

### 32.8.6 Random Numbers

---

Function **HypergeometricDistRandom**(*Size* As *mpNum*, *n* As *mpNum*, *M* As *mpNum*, *N* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **HypergeometricDistRandom** returns random numbers following a central hypergeometric distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: The size of the sample.

*M*: The number of successes in the population

*N*: The population size

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 32.9 Lognormal Distribution

### 32.9.1 Definition

These functions return PDF, CDF, and ICDF of the lognormal distribution with location  $a$ , scale  $b > 0$ , and the support interval  $(0, +\infty)$  :

A log-normal (or lognormal) distribution is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable is log-normally distributed, then has a normal distribution. Likewise, if has a normal distribution, then has a log-normal distribution. A random variable which is log-normally distributed takes only positive real values.

In a log-normal distribution  $X$ , the parameters denoted  $\mu$  and  $\sigma$  are, respectively, the mean and standard deviation of the variable's natural logarithm (by definition, the variable's logarithm is normally distributed), which means

$$X = e^{\mu + \sigma Z} \tag{32.9.1}$$

with  $Z$  a standard normal variable.

This relationship is true regardless of the base of the logarithmic or exponential function. If  $\log_a(Y)$  is normally distributed, then so is  $\log_b(Y)$ , for any two positive numbers  $a, b \neq 1$ . Likewise, if  $e^X$  is log-normally distributed, then so is  $a^X$ , where  $a$  is a positive number  $\neq 1$ .

On a logarithmic scale,  $\mu$  and  $\sigma$  can be called the location parameter and the scale parameter, respectively.

In contrast, the mean, standard deviation, and variance of the non-logarithmized sample values are respectively denoted  $m$ , *s.d.*, and  $v$  in this article. The two sets of parameters can be related as

$$\mu = \ln \left( \frac{m^2}{\sqrt{v + m^2}} \right), \quad \sigma = \sqrt{\ln \left( 1 + \frac{v}{m^2} \right)} \tag{32.9.2}$$

### 32.9.2 Density and CDF

---

Function **LogNormalDist**(*x* As *mpNum*, **mean** As *mpNum*, **stdev** As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function **LogNormalDist** returns returns pdf, CDF and related information for the Lognormal-distribution

**Parameters:**

*x*: A real number

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.2.2.1 and 32.2.2.2.

#### 32.9.2.1 Density

$$f(x) = \frac{1}{bx\sqrt{2\pi}} \exp\left(-\frac{(\ln(x) - a)^2}{2b^2}\right) \quad (32.9.3)$$

#### 32.9.2.2 CDF

$$F(x) = \frac{1}{2} \left( 1 + \operatorname{erf}\left(\frac{\ln(x) - a}{b\sqrt{2}}\right) \right) \quad (32.9.4)$$

### 32.9.3 Quantiles

---

Function **LognormalDistInv**(*Prob* As *mpNum*, **mean** As *mpNum*, **stdev** As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function **LognormalDistInv** returns returns quantiles and related information for the the Lognormal-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

#### 32.9.3.1 Quantiles: algorithms and formulas

$$F^{-1}(y) = \exp(a + b \cdot \operatorname{normstdinv}(y)) \quad (32.9.5)$$

### 32.9.4 Properties

---

Function **LognormalDistInfo**(**mean** As *mpNum*, **stdev** As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function **LognormalDistInfo** returns returns moments and related information for the central Lognormal-distribution

**Parameters:**

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

#### 32.9.4.1 Moments: algorithms and formulas

Algebraic moments of the log-normal distribution are given by

$$\mu'_k = e^{k\mu + k^2\sigma^2/2} \quad (32.9.6)$$

#### 32.9.5 Random Numbers

---

Function **LognormalRandom**(*Size* As *mpNum*, *mean* As *mpNum*, *stdev* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **LognormalRandom** returns returns random numbers following a central Beta-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 32.6.5.

#### 32.9.5.1 Random Numbers: algorithms and formulas

The most straightforward way of achieving random numbers from a log-normal distribution is to generate a random number  $u$  from a normal distribution with mean  $\mu$  and standard deviation  $\sigma$  and construct  $r = e^u$ .

## 32.10 Negative Binomial Distribution

These functions return PMF and CDF of the (discrete) negative binomial distribution with target for number of successful trials  $r > 0$  and success probability  $0 \leq p \leq 1$ .

If  $r = n$  is a positive integer the name Pascal distribution is used, and for  $r = 1$  it is called geometric distribution.

See [Ong & Lee \(1979\)](#) for information on the noncentral negative binomial distribution

### 32.10.1 Density and CDF

---

Function **NegativeBinomialDist**(*x* As mpNum, *r* As mpNum, *p* As mpNum, **Output** As String) As mpNumList

---

The function NegativeBinomialDist returns returns pdf, CDF and related information for the central negative binomial distribution

**Parameters:**

*x*: The number of failures in trials.

*r*: The threshold number of successes.

*p*: The probability of a success

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.3.1.1 and 32.3.1.2.

#### 32.10.1.1 Density

$$f_{\text{NegBin}}(r, k; p) = \frac{\Gamma(k + r)}{k! \Gamma(r)} p^r (1 - p)^k = \frac{p}{r + k} f_{\text{Beta}}(r, k + 1, p) \quad (32.10.1)$$

#### 32.10.1.2 CDF

$$F_{\text{NegBin}}(r, k; p) = I_{1-p}(r, k + 1) = ibeta(r, k + 1, 1 - p) \quad (32.10.2)$$

### 32.10.2 Quantiles

---

Function **NegativeBinomialDistInv**(**Prob** As mpNum, *r* As mpNum, *p* As mpNum, **Output** As String) As mpNumList

---

The function NegativeBinomialDistInv returns returns quantiles and related information for the the central binomial-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*r*: The threshold number of successes.

*p*: The probability of a success

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

### 32.10.3 Properties

---

Function **NegativeBinomialDistInfo**(*r* As mpNum, *p* As mpNum, **Output** As String) As mp-NumList

---

The function NegativeBinomialDistInfo returns returns moments and related information for the central Binomial-distribution

**Parameters:**

*r*: The threshold number of successes.

*p*: The probability of a success

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

### 32.10.3.1 Moments: algorithms and formulas

$$\mu_1 = np \quad (32.10.3)$$

$$\mu_2 = np(1 - p) = npq \quad (32.10.4)$$

$$\mu_3 = npq(q + p) \quad (32.10.5)$$

$$\mu_4 = npq(3npq + 6pq + 1) \quad (32.10.6)$$

### 32.10.3.2 Recurrence relations

The following recurrence relations hold:

$$f_{\text{NegBin}}(r, k + 1; p) = \frac{(r + k)(1 - p)}{k + 1} f_{\text{NegBin}}(r, k; p) \quad (32.10.7)$$

$$f_{\text{NegBin}}(r, k - 1; p) = \frac{k}{(r + k - 1)(1 - p)} f_{\text{NegBin}}(r, k; p) \quad (32.10.8)$$

## 32.10.4 Random Numbers

---

Function **NegativeBinomialDistRandom**(*Size* As *mpNum*, *r* As *mpNum*, *p* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **NegativeBinomialDistRandom** returns random numbers following a central Binomial-distribution

#### Parameters:

*Size*: A positive integer up to  $10^7$

*r*: The threshold number of successes.

*p*: The probability of a success

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

### 32.10.4.1 Random Numbers: algorithms and formulas

Random numbers from a negative binomial distribution can be obtained using the algorithms outline for the beta distribution.

## 32.11 Normal Distribution

### 32.11.1 Definition

A random variable is said to follow a normal distribution with mean  $\mu$  and variance  $\sigma^2$ , if its pdf is given by 32.11.1. It is said to follow a standardized normal distribution if its pdf is given by 32.11.2.

### 32.11.2 Density and CDF

---

Function **NDist**(*x* As *mpNum*, **mean** As *mpNum*, **stdev** As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function **NDist** returns returns pdf, CDF and related information for the normal-distribution

**Parameters:**

*x*: A real number

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.2.2.1 and 32.2.2.2.

#### 32.11.2.1 Density

This functions returns the pdf of the normal distribution with mean  $\mu$  and variance  $\sigma^2$ , which is given by

$$f_N(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \quad (32.11.1)$$

The pdf of the standardized normal distribution with mean 0 and variance 1 is given by

$$\phi(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2}, \quad (32.11.2)$$

These two functions are related by

$$f_N(x; \mu, \sigma^2) = \frac{1}{\sigma} \phi\left(\frac{x-\mu}{\sigma}\right), \text{ and } \phi(u) = \sigma f_N(\mu + \sigma u) \quad (32.11.3)$$

#### 32.11.2.2 CDF

This functions returns the cdf of the normal distribution with mean  $\mu$  and variance  $\sigma^2$ , which is given by

$$F_N(x; \mu, \sigma^2) = \int_{-\infty}^x f_N(v) dv \quad (32.11.4)$$

The cdf of the standardized normal distribution with mean 0 and variance 1 is given by

$$\Phi(u) = \int_{-\infty}^u \phi(w) dw \quad (32.11.5)$$

These two functions are related by

$$F_N(x; \mu, \sigma^2) = \Phi\left(\frac{x - \mu}{\sigma}\right), \text{ and } \Phi(u) = F_N(\mu + \sigma u) \quad (32.11.6)$$

### 32.11.3 Quantiles

These functions return the quantile of the normal distribution with mean  $\mu$  and variance  $\sigma^2$ ,  $F_N^{-1}(\alpha; \mu, \sigma^2)$ , or the standardized normal distribution with mean 0 and variance 1,  $\Phi^{-1}(\alpha)$ .

---

**Function `NDistInv`**(*Prob* As mpNum, *mean* As mpNum, *stdev* As mpNum, *Output* As String)  
As mpNumList

---

The function `NDistInv` returns quantiles and related information for the Lognormal-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Output*: A string describing the output choices

See section [32.1.3.2](#) for the options for *Prob* and *Output*).

#### 32.11.3.1 Quantiles: algorithms and formulas

$$F^{-1}(y) = \exp(a + b \cdot \text{normstdinv}(y)) \quad (32.11.7)$$

### 32.11.4 Properties

---

**Function `NormalDistInfo`**(*mean* As mpNum, *stdev* As mpNum, *Output* As String) As mpNumList

---

The function `NormalDistInfo` returns moments and related information for the central Lognormal-distribution

**Parameters:**

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Output*: A string describing the output choices

See section [32.1.3.3](#) for the options for *Output*. Algorithms and formulas are given in section [32.13.4](#).

#### 32.11.4.1 Moments: algorithms and formulas

$$\kappa_1 = \mu$$

$$\kappa_2 = \sigma^2$$

$$\kappa_r = 0 \text{ for } r \geq 3.$$

### 32.11.4.2 Differential Equation

Let  $Z^{(m)}$  denote the  $m^{\text{th}}$  derivative of  $Z(x)$ . Then (Abramowitz & Stegun., 1970)

$$Z^{(1)} = -xZ(x) \quad (32.11.8)$$

$$Z^{(m+2)} + xZ^{(m+1)} + (m+1)Z^{(m)} = 0 \quad (32.11.9)$$

## 32.11.5 Random Numbers

---

Function **NormalRandom**(*Size* As *mpNum*, *mean* As *mpNum*, *stdev* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function NormalRandom returns random numbers following a central Beta-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*mean*: A real number greater 0, representing the mean of the distribution

*stdev*: A real number greater 0, representing the standard deviation of the distribution

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 32.6.5.

### 32.11.5.1 Random Numbers: algorithms and formulas

Let  $Z_1 \sim Re(0; 1)$ ,  $Z_2 \sim Re(0, 1)$  be independent random variables. Then

$X_1 = \sqrt{-2 \ln Z_1} \cos(2\pi Z_2)$  and  $X_2 = \sqrt{-2 \ln Z_1} \sin(2\pi Z_2)$  are  $\sim No(0; 1)$ .

It is also possible to directly use  $\Phi^{-1}(\alpha)$ .

## 32.12 Poisson Distribution

### 32.12.1 Definition

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event. The following functions return PMF and CDF of the Poisson distribution with mean  $\mu \geq 0$ .

### 32.12.2 Density and CDF

---

Function **PoissonDist**(*x* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function PoissonDist returns pdf, CDF and related information for the Poisson distribution

**Parameters:**

*x*: A real number

*lambda*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.4.2.1 and 32.4.2.2.

### 32.12.2.1 Density

$$f(k) = \frac{\mu^k}{k!} e^{-\mu} = \text{fcIgprefix}(1 + k, \mu) \quad (32.12.1)$$

### 32.12.2.2 CDF

$$F(k) = e^{-\mu} \sum_{i=0}^k \frac{\mu^i}{i!} = \text{igammaq}(1 + k, \mu) \quad (32.12.2)$$

## 32.12.3 Quantiles

---

Function **PoissonDistInv**(*Prob* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function PoissonDistInv returns quantiles and related information for the the Poisson distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*lambda*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given in section 32.12.3.1.

### 32.12.3.1 Quantiles: algorithms and formulas

The algorithms follow the one for the chisquare distribution.

## 32.12.4 Properties

---

Function **PoissonDistInfo**(*lambda* As mpNum, *Output* As String) As mpNumList

---

The function PoissonDistInfo returns moments and related information for the Poisson distribution

**Parameters:**

*lambda*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.4.4.

### 32.12.4.1 Moments and Cumulants

The momemnts and cumulants are given by

$$\kappa_r = \lambda \quad (32.12.3)$$

$$\mu_1 = \mu_2 = \mu_3 = \lambda \quad (32.12.4)$$

$$\mu_4 = 3\lambda^2 + \lambda \quad (32.12.5)$$

### 32.12.5 Random Numbers

---

Function **PoissonDistRan**(*Size* As *mpNum*, *lambda* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function PoissonDistRan returns random numbers following a Poisson distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*lambda*: A real number greater 0, representing the degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 32.4.5.

## 32.13 Student's t-Distribution

### 32.13.1 Definition

If  $X$  is a random variable following a normal distribution with mean zero and variance unity and  $\chi^2$  is a random variable following an independent  $\chi^2$ -distribution with  $n$  degrees of freedom, then the distribution of the ratio  $\frac{X}{\sqrt{\chi^2/n}}$  is called Student's t-distribution with  $n$  degrees of freedom

### 32.13.2 Density and CDF

---

Function **TDist**(*x* As *mpNum*, *n* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function TDist returns returns pdf, CDF and related information for the central  $t$ -distribution

**Parameters:**

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.13.2.1 and 32.13.2.2.

#### 32.13.2.1 Density

The density of a variable following a central Student's t-distribution with  $n$  degrees of freedom is given by

$$f_t(n, x) = \frac{\Gamma((n+1)/2)}{\sqrt{n\pi}\Gamma(n/2)} \left(\frac{n}{n+x^2}\right)^{(n+1)/2} \quad (32.13.1)$$

where  $\Gamma(\cdot)$  denotes the Gamma function (see section ??.)

### 32.13.2.2 CDF: General formulas

The cdf of a variable following a central t-distribution with  $n$  degrees of freedom is defined as

$$\Pr [X \leq x] = F_t(n, x) = \int_0^x f_t(n, t) dt \quad (32.13.2)$$

The cdf of the central t-distribution is calculated for any positive degrees of freedom  $n$  using the relationships

$$2F_t(n, x) = F_F(1, n; x^2), \quad x \leq 0 \quad (32.13.3)$$

$$F_t(n, x) - F_t(n, -x) = F_F(1, n; x^2), \quad x \geq 0 \quad (32.13.4)$$

$$F_t(n, x) = 1 - F_t(n, -x) \quad (32.13.5)$$

where  $F_F(1, n, x^2)$  denotes the cdf of the central  $F$ -distribution with 1 and  $n$  of freedom (see section 32.6.2.2).

### 32.13.3 Quantiles

---

Function **TDistInv**(*Prob* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function **TDistInv** returns returns quantiles and related information for the the central  $t$ -distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

### 32.13.4 Properties

---

Function **TDistInfo**(*n* As mpNum, *Output* As String) As mpNumList

---

The function **TDistInfo** returns returns moments and related information for the central  $t$ -distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

#### 32.13.4.1 Moments: algorithms and formulas

The algebraic moments (defined for  $n > r$ ) are given by

$$\mu'_r = \left(\frac{1}{2}n\right)^{r/2} \frac{\Gamma\left(\frac{1}{2}(n-r)\right)}{\Gamma\left(\frac{1}{2}n\right)}. \quad (32.13.6)$$

### 32.13.5 Random Numbers

---

Function **TDistRan**(*Size* As *mpNum*, *n* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **TDistRan** returns random numbers following a central *t*-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 32.13.5.

#### 32.13.5.1 Random Numbers: algorithms and formulas

Following the definition we may define a random number *t* from a *t*-distribution, using random numbers from a normal and a chi-square distribution, as  $t = \frac{z}{\sqrt{y_n/n}}$ , where *z* is a standard normal and *y<sub>n</sub>* a chi-squared variable with *n* degrees of freedom. To obtain random numbers from these distributions see the appropriate sections.

### 32.13.6 Behrens-Fisher Problem

See [Golhar \(1972\)](#)

## 32.14 Weibull Distribution

These functions return PDF, CDF, and ICDF of the Weibull distribution with shape parameter *a* and scale *b* > 0 and the support interval  $(0, +\infty)$  :

#### 32.14.1 Density and CDF

---

Function **WeibullDist**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **WeibullDist** returns pdf, CDF and related information for the Weibull distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.2.2.1 and 32.2.2.2.

### 32.14.1.1 Density

$$f(x) = \frac{x}{b^2} \exp\left(-\frac{x^2}{2b^2}\right) \exp(-(x/b)^a) \quad (32.14.1)$$

### 32.14.1.2 CDF

$$F(x) = 1 - \exp(-(x/b)^a) = -\text{expm1}(-(x/b)^a) \quad (32.14.2)$$

## 32.14.2 Quantiles

---

Function **WeibullDistInv**(*Prob* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function WeibullDistInv returns returns quantiles and related information for the the central Beta-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

$$F^{-1}(y) = b(-\text{ln1p}(-y))^{1/a} \quad (32.14.3)$$

## 32.14.3 Properties

---

Function **WeibullDistInfo**(*a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function WeibullDistInfo returns returns moments and related information for the central Beta-distribution

**Parameters:**

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

### 32.14.3.1 Moments: algorithms and formulas

$$\mu'_r = \sum_{j=0}^r \binom{r}{j} \Gamma\left(\frac{r-j}{c} + 1\right) b^{r-j} \quad (32.14.4)$$

$$\mu_1 = b \Gamma\left(\frac{1}{c} + 1\right) \quad (32.14.5)$$

$$\mu_2 = b^2 \left[ \Gamma\left(\frac{1}{c} + 1\right) \Gamma^2\left(\frac{1}{c} + 1\right) \right] \quad (32.14.6)$$

See Rinne (2008) for further details.

### 32.14.4 Random Numbers

---

Function **WeibullDistRandom**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

---

The function **WeibullDistRandom** returns random numbers following a central Beta-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 32.6.5.

## 32.15 Bernoulli Distribution

The Bernoulli distribution is a discrete distribution of the outcome of a single trial with only two results, 0 (failure) or 1 (success), with a probability of success  $p$ . The Bernoulli distribution is the simplest building block on which other discrete distributions of sequences of independent Bernoulli trials can be based. The Bernoulli is the binomial distribution (( $k = 1$ ,  $p$ )) with only one trial.

### 32.15.1 Density and CDF

---

Function **BernoulliDistBoost**(*k* As *mpNum*, *p* As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function **BernoulliDistBoost** returns pdf, CDF and related information for the central  $t$ -distribution

**Parameters:**

*k*: A real number, 0 or 1

*p*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.15.1.1 and 32.15.1.2.

#### 32.15.1.1 Density

$$f(x) = \begin{cases} q = 1 - p & \text{for } k = 0 \\ p & \text{for } k = 1. \end{cases} \quad (32.15.1)$$

#### 32.15.1.2 CDF

$$F(x) = \begin{cases} 0 & \text{for } k = 0 \\ q & \text{for } k = 0 \\ 1 & \text{for } k = 1. \end{cases} \quad (32.15.2)$$

## 32.15.2 Quantiles

---

Function **BernoulliDistInvBoost**(*Prob* As mpNum, *p* As mpNum, *Output* As String) As mp-NumList

---

The function BernoulliDistInvBoost returns returns quantiles and related information for the the central *t*-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*p*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

### 32.15.2.1 Quantiles: Algorithm

Using the relation:  $\text{cdf} = 1 - p$  for  $k = 0$ , else 1.

## 32.15.3 Properties

---

Function **BernoulliDistInfoBoost**(*p* As mpNum, *Output* As String) As mpNumList

---

The function BernoulliDistInfoBoost returns returns moments and related information for the central *t*-distribution

**Parameters:**

*p*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.15.3.

### 32.15.3.1 Moments: algorithms and formulas

$$\mu'_r = \sum_{i=0}^{r-1} \binom{r}{i} (-1)^i p^{i+1} + (-p)^r \quad (32.15.3)$$

$$\mu_1 = p \quad (32.15.4)$$

$$\mu_2 = pq \quad (32.15.5)$$

$$\mu_3 = pq(1 - 2p) \quad (32.15.6)$$

$$\mu_4 = pq(1 - 3pq) \quad (32.15.7)$$

### 32.15.4 Random Numbers

---

Function **BernoulliDistRandomBoost**(*Size* As *mpNum*, *p* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

---

The function BernoulliDistRandomBoost returns returns random numbers following a central Binomial-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*p*: The probability of a success on each trial.

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 32.16 Cauchy Distribution

### 32.16.1 Density and CDF

---

Function **CauchyDistBoost**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function CauchyDistBoost returns returns pdf, CDF and related information for the Cauchy distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*.

#### 32.16.1.1 Density

$$f(x) = \frac{1}{\pi(1 + ((x - a)/b)^2)} \quad (32.16.1)$$

#### 32.16.1.2 CDF

$$F(x) = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{x - a}{b}\right) \quad (32.16.2)$$

### 32.16.2 Quantiles

---

Function **CauchyDistInvBoost**(*Prob* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function CauchyDistInvBoost returns returns quantiles and related information for the Cauchy distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(y) = \begin{cases} a - b / \tan(\pi y), & y < 0.5, \\ a, & y = 0.5, \\ a - b / \tan(\pi(1 - y)) & y > 0.5. \end{cases} \quad (32.16.3)$$

### 32.16.3 Properties

---

Function **CauchyDistInfoBoost**(*a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **CauchyDistInfoBoost** returns returns moments and related information for the Cauchy distribution

**Parameters:**

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*.

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis\_excess, range and support. Note however that the Cauchy distribution does not have a mean, standard deviation, etc. See mathematically undefined function to control whether these should fail to compile with a BOOST\_STATIC\_ASSERTION\_FAILURE, which is the default. Alternately, the functions mean, standard deviation, variance, skewness, kurtosis and kurtosis\_excess will all return a domain\_error if called.

### 32.16.4 Random Numbers

---

Function **CauchyDistRandomBoost**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **CauchyDistRandomBoost** returns returns random numbers following a Cauchy distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in below.

## 32.17 Extreme Value (or Gumbel) Distribution

These functions return PDF, CDF, and ICDF of the Extreme Value Type I distribution with location  $a$ , scale  $b > 0$ , and the support interval  $(-\infty, +\infty)$ :

### 32.17.1 Density and CDF

---

Function **ExtremevalueDistBoost**(*x* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String)  
As mpNumList

---

The function **ExtremevalueDistBoost** returns returns pdf, CDF and related information for the Extreme Value distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.17.1.1 and 32.17.1.2.

#### 32.17.1.1 Density

$$f(x) = \frac{e^{-(x-a)/b}}{b} e^{e^{-(x-a)/b}} \quad (32.17.1)$$

#### 32.17.1.2 CDF

$$F(x) = e^{e^{-(x-a)/b}} \quad (32.17.2)$$

### 32.17.2 Quantiles

---

Function **ExtremevalueDistInvBoost**(*Prob* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function **ExtremevalueDistInvBoost** returns returns quantiles and related information for the the Extreme Value distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(y) = a - \ln(-\ln(y)) \quad (32.17.3)$$

### 32.17.3 Properties

---

Function **ExtremevalueDistInfoBoost**(*a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function ExtremevalueDistInfoBoost returns returns moments and related information for the Extreme Value distribution

**Parameters:**

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section [32.1.3.3](#) for the options for *Output*.

### 32.17.4 Random Numbers

---

Function **ExtremevalueDistRandomBoost**(*Size* As mpNum, *a* As mpNum, *b* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function ExtremevalueDistRandomBoost returns returns random numbers following a Extreme Value distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in below.

## 32.18 Geometric Distribution

Geometric distribution: it is used when there are exactly two mutually exclusive outcomes of a Bernoulli trial: these outcomes are labelled "success" and "failure". For Bernoulli trials each with success fraction *p*, the geometric distribution gives the probability of observing *k* trials (failures, events, occurrences, or arrivals) before the first success.

### 32.18.1 Density and CDF

---

Function **GeometricDistBoost**(*k* As mpNum, *p* As mpNum, *Output* As String) As mpNumList

---

The function GeometricDistBoost returns returns pdf, CDF and related information for the Geometric distribution

**Parameters:**

*k*: A real number

*p*: A real number greater 0, representing the numerator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.18.1.1 and 32.18.1.2.

### 32.18.1.1 Density

$$f(k; p) = p(1 - p)^k \quad (32.18.1)$$

### 32.18.1.2 CDF

$$F(k; p) = 1 - (1 - p)^{k+1} \quad (32.18.2)$$

## 32.18.2 Quantiles

---

Function **GeometricDistInvBoost**(*Prob* As *mpNum*, *p* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **GeometricDistInvBoost** returns returns quantiles and related information for the Geometric distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*p*: A real number greater 0, representing the numerator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(x; p) = \frac{\log1p(-x)}{\log1p(-p)} - 1 \quad (32.18.3)$$

## 32.18.3 Properties

---

Function **GeometricDistInfoBoost**(*p* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **GeometricDistInfoBoost** returns returns moments and related information for the Geometric distribution

**Parameters:**

*p*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*.

## 32.18.4 Random Numbers

---

Function **GeometricDistRandomBoost**(*Size* As *mpNum*, *p* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function `GeometricDistRandomBoost` returns random numbers following a Geometric distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*p*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in below.

## 32.19 Inverse Chi Squared Distribution

### 32.19.1 Definition

The inverse-chi-squared distribution (or inverted-chi-square distribution[1] ) is the probability distribution of a random variable whose multiplicative inverse (reciprocal) has a chi-squared distribution. It is also often defined as the distribution of a random variable whose reciprocal divided by its degrees of freedom is a chi-squared distribution. That is, if  $X$  has the chi-squared distribution with  $\nu$  degrees of freedom, then according to the first definition,  $1/X$  has the inverse-chi-squared distribution with  $\nu$  degrees of freedom; while according to the second definition,  $\nu/X$  has the inverse-chi-squared distribution with  $\nu$  degrees of freedom.

The inverse-chi-squared distribution is a special case of a inverse-gamma distribution with  $\nu$  (degrees of freedom), shape ( $\alpha$ ) and scale ( $\beta$ ), where  $\alpha = \nu/2$  and  $\beta = 1/2$ .

### 32.19.2 Density and CDF

---

Function **InverseChiSquaredDistBoost**(*x* As `mpNum`, *n* As `mpNum`, *Output* As `String`) As `mpNumList`

---

The function `InverseChiSquaredDistBoost` returns pdf, CDF and related information for the inverse-chi-squared -distribution

**Parameters:**

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*.

#### 32.19.2.1 Density

The first definition yields a probability density function given by

$$f(x; \nu) = \frac{2^{-\nu/2}}{\Gamma(\nu/2)} x^{-\nu/2-1} e^{-1/(2x)} \quad (32.19.1)$$

while the second definition yields the density function

$$f(x; \nu) = \frac{(\nu/2)^{\nu/2}}{\Gamma(\nu/2)} x^{-\nu/2-1} e^{-\nu/(2x)} \quad (32.19.2)$$

In both cases,  $x > 0$  and  $\nu$  is the degrees of freedom parameter. Further,  $\Gamma$  is the gamma function. Both definitions are special cases of the scaled-inverse-chi-squared distribution. For the first definition the variance of the distribution is  $\sigma = 1/\nu$ , while for the second definition  $\sigma = 1$ .

### 32.19.2.2 CDF

$$F(x; \nu) = \frac{1}{\Gamma(\nu/2)} \Gamma\left(\frac{\nu}{2}, \frac{1}{2x}\right) \quad (32.19.3)$$

## 32.19.3 Quantiles

---

Function **InverseChiSquaredDistInvBoost**(*Prob* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function **InverseChiSquaredDistInvBoost** returns quantiles and related information for the inverse-chi-squared distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*).

$$F^{-1}(prob; \nu) = \beta // gamma - q - inv(\alpha, p) \quad (32.19.4)$$

## 32.19.4 Properties

---

Function **InverseChiSquaredDistInfoBoost**(*n* As mpNum, *Output* As String) As mpNumList

---

The function **InverseChiSquaredDistInfoBoost** returns moments and related information for the inverse-chi-squared distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

### 32.19.4.1 Moments and Cumulants

$$\mu_1 = \frac{\nu}{\nu - 2} \text{ for } \nu > 2. \quad (32.19.5)$$

## 32.19.5 Random Numbers

---

Function **InverseChiSquaredDistRanBoost**(*Size* As mpNum, *n* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function **InverseChiSquaredDistRanBoost** returns random numbers following a inverse-chi-squared distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 32.20 Inverse Gamma Distribution

### 32.20.1 Definition

In probability theory and statistics, the inverse gamma distribution is a two-parameter family of continuous probability distributions on the positive real line, which is the distribution of the reciprocal of a variable distributed according to the gamma distribution. Perhaps the chief use of the inverse gamma distribution is in Bayesian statistics, where the distribution arises as the marginal posterior distribution for the unknown variance of a normal distribution if an uninformative prior is used; and as an analytically tractable conjugate prior if an informative prior is required.

However, it is common among Bayesians to consider an alternative parametrization of the normal distribution in terms of the precision, defined as the reciprocal of the variance, which allows the gamma distribution to be used directly as a conjugate prior. Other Bayesians prefer to parametrize the inverse gamma distribution differently, as a scaled inverse chi-squared distribution

### 32.20.2 Density and CDF

---

Function **InverseGammaDistBoost**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **InverseGammaDistBoost** returns returns pdf, CDF and related information for the inverse gamma distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.20.2.1 and 32.20.2.2.

#### 32.20.2.1 Density

The inverse gamma distribution's probability density function is defined over the support  $x > 0$

$$f(x; \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} \exp\left(-\frac{\beta}{x}\right) \quad (32.20.1)$$

with shape parameter  $\alpha$  and scale parameter  $\beta$ .

### 32.20.2.2 CDF

The cumulative distribution function is the regularized gamma function

$$F(x; \alpha, \beta) = \frac{\Gamma(\alpha), \beta/x}{\Gamma(\alpha)} = Q\left(\alpha, -\frac{\beta}{x}\right) \quad (32.20.2)$$

where the numerator is the upper incomplete gamma function and the denominator is the gamma function. Many math packages allow you to compute  $Q$ , the regularized gamma function, directly.

### 32.20.3 Quantiles

---

Function **InverseGammaDistInvBoost**(*Prob* As mpNum, *m* As mpNum, *n* As mpNum, **Output** As String) As mpNumList

---

The function **InverseGammaDistInvBoost** returns returns quantiles and related information for the the inverse gamma distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(prob; \nu) = \beta // gamma - q - inv(\alpha, p) \quad (32.20.3)$$

### 32.20.4 Properties

---

Function **InverseGammaDistInfoBoost**(*a* As mpNum, *b* As mpNum, **Output** As String) As mpNumList

---

The function **InverseGammaDistInfoBoost** returns returns moments and related information for the inverse gamma distribution

**Parameters:**

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 32.20.4.1 Moments and Cumulants

$$\mu_1 = \frac{\nu}{\nu - 2} \text{ for } \nu > 2. \quad (32.20.4)$$

### 32.20.5 Random Numbers

---

Function **InverseGammaDistRanBoost**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function `InverseGammaDistRanBoost` returns random numbers following a inverse gamma distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 32.21 Inverse Gaussian (or Wald) Distribution

### 32.21.1 Definition

In probability theory, the inverse Gaussian distribution (also known as the Wald distribution) is a two-parameter family of continuous probability distributions with mean  $\mu$  and shape parameter  $\lambda$  and support on  $(0, \infty)$ . As  $\lambda$  tends to infinity, the inverse Gaussian distribution becomes more like a normal (Gaussian) distribution. The inverse Gaussian distribution has several properties analogous to a Gaussian distribution. The name can be misleading: it is an "inverse" only in that its cumulant generating function (logarithm of the characteristic function) is the inverse of the cumulant generating function of a Gaussian random variable.

While the Gaussian describes a Brownian Motion's level at a fixed time, the inverse Gaussian describes the distribution of the time a Brownian Motion with positive drift takes to reach a fixed positive level.

See also [http://en.wikipedia.org/wiki/Inverse\\_Gaussian\\_distribution](http://en.wikipedia.org/wiki/Inverse_Gaussian_distribution).

### 32.21.2 Density and CDF

---

Function **InverseGaussianDistBoost**(*x* As *mpNum*, *mu* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function `InverseGaussianDistBoost` returns pdf, CDF and related information for the inverse Gaussian distribution

**Parameters:**

*x*: A real number

*mu*: A real number greater 0, representing the numerator degrees of freedom

*lambda*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.21.2.1 and 32.21.2.2.

### 32.21.2.1 Density

$$f(x; \mu, \lambda) = \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left(\frac{-\lambda(x-\mu)^2}{2\mu^2 x}\right) \quad (32.21.1)$$

### 32.21.2.2 CDF

$$F(x; \mu, \lambda) = \Phi\left(\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} - 1\right)\right) + \exp\left(\frac{2\lambda}{\mu}\right)\Phi\left(-\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} + 1\right)\right) \quad (32.21.2)$$

## 32.21.3 Quantiles

---

Function **InverseGaussianDistInvBoost**(*Prob* As mpNum, *mu* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function **InverseGaussianDistInvBoost** returns quantiles and related information for the inverse Gaussian distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*mu*: A real number greater 0, representing the numerator degrees of freedom

*lambda*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(prob; \nu) = \beta // gamma - q - inv(\alpha, p) \quad (32.21.3)$$

## 32.21.4 Properties

---

Function **InverseGaussianDistInfoBoost**(*mu* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function **InverseGaussianDistInfoBoost** returns moments and related information for the inverse Gaussian distribution

**Parameters:**

*mu*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

### 32.21.4.1 Moments and Cumulants

$$\mu_1 = \frac{\nu}{\nu - 2} \text{ for } \nu > 2. \quad (32.21.4)$$

### 32.21.5 Random Numbers

---

Function **InverseGaussianDistRanBoost**(*Size* As *mpNum*, *mu* As *mpNum*, *lambda* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **InverseGaussianDistRanBoost** returns random numbers following a inverse Gaussian distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*mu*: A real number greater 0, representing the numerator degrees of freedom

*lambda*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 32.22 Laplace Distribution

These functions return PDF, CDF, and ICDF of the Laplace distribution with location  $a$ , scale  $b > 0$ , and the support interval  $(-\infty, +\infty)$ :

### 32.22.1 Density and CDF

---

Function **LaplaceDistBoost**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **LaplaceDistBoost** returns pdf, CDF and related information for the Laplace distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.22.1.1 and 32.22.1.2.

#### 32.22.1.1 Density

$$f(x) = \exp(-|x - a|/b)/(2b) \quad (32.22.1)$$

#### 32.22.1.2 CDF

$$F(x) = \begin{cases} \frac{1}{2} - \frac{1}{2}\exp(-\frac{|x-a|}{b}) & x \geq a \\ \frac{1}{2}\exp(-\frac{|x-a|}{b}) & x < a. \end{cases} \quad (32.22.2)$$

### 32.22.2 Quantiles

---

Function **LaplaceDistInvBoost**(*Prob* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function **LaplaceDistInvBoost** returns returns quantiles and related information for the the Laplace distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section [32.1.3.2](#) for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(y) = \begin{cases} a + b \ln(2y), & y < 0.5, \\ a - b \ln(2(1 - y)) & y > 0.5. \end{cases} \quad (32.22.3)$$

### 32.22.3 Properties

---

Function **LaplaceDistInfoBoost**(*a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function **LaplaceDistInfoBoost** returns returns moments and related information for the Laplace distribution

**Parameters:**

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section [32.1.3.3](#) for the options for *Output*.

### 32.22.4 Random Numbers

---

Function **LaplaceDistRanBoost**(*Size* As mpNum, *a* As mpNum, *b* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function **LaplaceDistRanBoost** returns returns random numbers following a Laplace distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in below.

## 32.23 Logistic Distribution

### 32.23.1 Definition

These functions return PDF, CDF, and ICDF of the logistic distribution with location  $a$ , scale  $b > 0$ , and the support interval  $(-\infty, +\infty)$ :

### 32.23.2 Density and CDF

---

Function **LogisticDistBoost**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function `LogisticDistBoost` returns returns pdf, CDF and related information for the Logistic distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.23.2.1 and 32.23.2.2.

#### 32.23.2.1 Density

$$f(x) = \frac{1}{b} \frac{\exp\left(-\frac{x-a}{b}\right)}{\left(1 + \exp\left(-\frac{x-a}{b}\right)\right)^2} \quad (32.23.1)$$

#### 32.23.2.2 CDF

$$F(x) = \frac{1}{1 + \exp\left(-\frac{x-a}{b}\right)} \quad (32.23.2)$$

### 32.23.3 Quantiles

---

Function **LogisticDistInvBoost**(*Prob* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function `LogisticDistInvBoost` returns returns quantiles and related information for the the Logistic distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(y) = a - b \ln((1 - y)/y) \quad (32.23.3)$$

### 32.23.4 Properties

---

Function **LogisticDistInfoBoost**(*a* As mpNum, *b* As mpNum, **Output** As String) As mpNumList

---

The function LogisticDistInfoBoost returns returns moments and related information for the Logistic distribution

**Parameters:**

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section [32.1.3.3](#) for the options for *Output*.

### 32.23.5 Random Numbers

---

Function **LogisticDistRanBoost**(*Size* As mpNum, *a* As mpNum, *b* As mpNum, **Generator** As String, **Output** As String) As mpNumList

---

The function LogisticDistRanBoost returns returns random numbers following a Logistic distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in below.

## 32.24 Pareto Distribution

### 32.24.1 Definition

These functions return PDF, CDF, and ICDF of the Pareto distribution with minimum (real) value  $k > 0$ , shape  $a > 0$ , and the support interval  $(k, +\infty)$  : This is a reference: [Wikipedia contributors \(2013\)](#)

### 32.24.2 Density and CDF

---

Function **ParetoDistBoost**(*x* As mpNum, *a* As mpNum, *b* As mpNum, **Output** As String) As mpNumList

---

The function ParetoDistBoost returns returns pdf, CDF and related information for the Pareto distribution

**Parameters:**

*x*: A real number

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.24.2.1 and 32.24.2.2.

### 32.24.2.1 Density

$$f(x) = \frac{a}{x} \left( \frac{k}{x} \right)^a \quad (32.24.1)$$

### 32.24.2.2 CDF

$$F(x) = 1 - \left( \frac{k}{x} \right)^a = -\text{powm1}(k/x, a) \quad (32.24.2)$$

## 32.24.3 Quantiles

---

Function **ParetoDistInvBoost**(*Prob* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function ParetoDistInvBoost returns returns quantiles and related information for the the Pareto distribution

#### Parameters:

*Prob*: A real number between 0 and 1.

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(y) = k(1 - y)^{-1/a} \quad (32.24.3)$$

## 32.24.4 Properties

---

Function **ParetoDistInfoBoost**(*a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function ParetoDistInfoBoost returns returns moments and related information for the Pareto distribution

#### Parameters:

*a*: A real number greater 0, representing the degrees of freedom

*b*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*.

### 32.24.5 Random Numbers

---

Function **ParetoDistRanBoost**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

---

The function ParetoDistRanBoost returns random numbers following a Pareto distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: A real number greater 0, representing the numerator degrees of freedom

*b*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in below.

## 32.25 Raleigh Distribution

### 32.25.1 Definition

These functions return PDF, CDF, and ICDF of the Rayleigh distribution with scale  $b > 0$  and the support interval  $(0, +\infty)$ :

### 32.25.2 Density and CDF

---

Function **RaleighDistBoost**(*x* As *mpNum*, *n* As *mpNum*, **Output** As *String*) As *mpNumList*

---

The function RaleighDistBoost returns pdf, CDF and related information for the Raleigh distribution

**Parameters:**

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section [32.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [32.25.3](#) and [32.25.4](#).

### 32.25.3 Density

$$f(x) = \frac{x}{b^2} \exp\left(-\frac{x^2}{2b^2}\right) \quad (32.25.1)$$

### 32.25.4 CDF

$$F(x) = 1 - \exp\left(-\frac{x^2}{2b^2}\right) = -\text{expm1}\left(-\frac{x^2}{2b^2}\right) \quad (32.25.2)$$

### 32.25.5 Quantiles

---

Function **RaleighDistInvBoost**(*Prob* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

The function **RaleighDistInvBoost** returns quantiles and related information for the Raleigh distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are below.

$$F^{-1}(y) = b\sqrt{-2 \cdot \ln 1p(-y)} \quad (32.25.3)$$

### 32.25.6 Properties

---

Function **RaleighDistInfoBoost**(*n* As mpNum, *Output* As String) As mpNumList

---

The function **RaleighDistInfoBoost** returns moments and related information for the Raleigh distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 32.25.6.1 Moments and Cumulants

$$\mu_1 = s\sqrt{\pi/2} \quad (32.25.4)$$

### 32.25.7 Random Numbers

---

Function **RaleighDistRanBoost**(*Size* As mpNum, *n* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function **RaleighDistRanBoost** returns random numbers following a Raleigh distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 32.26 Triangular Distribution

### 32.26.1 Definition

These functions return PDF, CDF, and ICDF of the triangular distribution on the support interval  $[a, b]$  with finite  $a < b$  and mode  $c$ ,  $a \leq c \leq b$ .

### 32.26.2 Density and CDF

---

Function **TriangularDistBoost**(*x* As mpNum, *a* As mpNum, *b* As mpNum, *c* As mpNum, **Output** As String) As mpNumList

---

The function TriangularDistBoost returns returns pdf, CDF and related information for the triangular distribution

#### Parameters:

*x*: A real number.

*a*: The left border parameter.

*b*: The right border parameter.

*c*: The mode parameter.

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.26.2.1 and 32.26.2.2.

#### 32.26.2.1 Density

$$f(x) = \begin{cases} 0 & x < a \\ \frac{2(x-a)}{(b-a)(c-a)} & a \leq x < c \\ \frac{2}{b-a} & x = c \\ \frac{2(b-x)}{(b-a)(b-c)} & c < x \leq b \\ 0 & x > b \end{cases} \quad (32.26.1)$$

#### 32.26.2.2 CDF

$$F(x) = \begin{cases} 0 & x < a \\ \frac{(x-a)^2}{(b-a)(c-a)} & a \leq x < c \\ \frac{c-a}{b-a} & x = c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & c < x \leq b \\ 1 & x > b \end{cases} \quad (32.26.2)$$

### 32.26.3 Quantiles

---

Function **TriangularDistInvBoost**(*Prob* As mpNum, *a* As mpNum, *b* As mpNum, *c* As mpNum, **Output** As String) As mpNumList

---

The function TriangularDistInvBoost returns returns quantiles and related information for the the triangular distribution

**Parameters:***Prob*: A real number between 0 and 1.*a*: The left border parameter.*b*: The right border parameter.*c*: The mode parameter.*Output*: A string describing the output choicesSee section 32.1.3.2 for the options for *Prob* and *Output*).

$$F^{-1}(y) = \begin{cases} a + \sqrt{(b-a)(c-a)y} & y < t \\ c & y = t \\ b - \sqrt{(b-a)(b-c)(1-y)} & y > t \end{cases} \quad (32.26.3)$$

where  $t = (c-a)/(b-a)$ .

### 32.26.4 Properties

---

Function **TriangularDistInfoBoost**(*a* As mpNum, *b* As mpNum, *c* As mpNum, *Output* As String)  
As mpNumList

---

The function TriangularDistInfoBoost returns returns moments and related information for the triangular distribution

**Parameters:***a*: The left border parameter.*b*: The right border parameter.*c*: The mode parameter.*Output*: A string describing the output choicesSee section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 32.26.4.1 Moments

$$\mu_1 = \frac{a+b+c}{3} \quad (32.26.4)$$

$$\mu_2 = \frac{a^2 + b^2 + c^2 - ab - ac - bc}{18} \quad (32.26.5)$$

$$\gamma_1 = \frac{\sqrt{2}(a+b-2c)(2a-b-c)(a-2b+c)}{5(a^2+b^2+c^2-ab-ac-bc)^{3/2}} \quad (32.26.6)$$

$$\gamma_2 = -\frac{3}{5} \quad (32.26.7)$$

### 32.26.5 Random Numbers

---

Function **TriangularDistRanBoost**(*Size* As mpNum, *a* As mpNum, *b* As mpNum, *c* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function TriangularDistRanBoost returns returns random numbers following a triangular distribution

**Parameters:***Size*: A positive integer up to  $10^7$ *a*: The left border parameter.*b*: The right border parameter.*c*: The mode parameter.*Generator*: A string describing the random generator*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 32.27 Uniform Distribution

### 32.27.1 Definition

These functions return PDF, CDF, and ICDF of the uniform distribution on the support interval  $[a, b]$  with finite  $a < b$ :

### 32.27.2 Density and CDF

---

Function **UniformDistBoost**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **UniformDistBoost** returns returns pdf, CDF and related information for the uniform distribution

**Parameters:***x*: A real number*a*: The left border parameter.*b*: The right border parameter.*Output*: A string describing the output choices

See section [32.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [32.27.2.1](#) and [32.27.2.2](#).

#### 32.27.2.1 Density

$$f(x) = \frac{1}{b-a} \quad (32.27.1)$$

#### 32.27.2.2 CDF

$$F(x) = \frac{x-a}{b-a} \quad (32.27.2)$$

### 32.27.3 Quantiles

---

Function **UniformDistInvBoost**(*Prob* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **UniformDistInvBoost** returns returns quantiles and related information for the the uniform distribution

**Parameters:***Prob*: A real number between 0 and 1.*a*: The left border parameter.*b*: The right border parameter.*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$F^{-1}(y) = a + y(b - a) \quad (32.27.3)$$

## 32.27.4 Properties

---

### Function **UniformDistInfoBoost**(*a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

---

The function UniformDistInfoBoost returns returns moments and related information for the uniform distribution

**Parameters:***a*: A real number greater 0, representing the degrees of freedom*b*: A real number greater 0, representing the degrees of freedom*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*.

### 32.27.4.1 Moments

$$\mu_1 = \frac{a + b}{2} \quad (32.27.4)$$

$$\mu_2 = \frac{(b - a)^2}{12} \quad (32.27.5)$$

$$\gamma_1 = 0 \quad (32.27.6)$$

$$\gamma_2 = -\frac{6}{5} \quad (32.27.7)$$

## 32.27.5 Random Numbers

---

### Function **UniformDistRanBoost**(*Size* As mpNum, *a* As mpNum, *b* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function UniformDistRanBoost returns returns random numbers following a uniform distribution

**Parameters:***Size*: A positive integer up to  $10^7$ *a*: A real number greater 0, representing the degrees of freedom*b*: A real number greater 0, representing the degrees of freedom*Generator*: A string describing the random generator*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in below.

# Chapter 33

## Noncentral Distribution Functions (based on Boost)

### 33.1 Noncentral Beta-Distribution

#### 33.1.1 Definition

If  $X_1$  and  $X_2$  are independent random variables,  $X_1$  following a non-central  $\chi^2$ -distribution with noncentrality parameter  $\lambda$  and  $2a$  degrees of freedom, and  $X_2$  following a  $\chi^2$ -distribution with  $2b$  degrees of freedom, then the distribution of the ratio  $B = \frac{X_1}{X_1 + X_2}$  is said to follow a non-central Beta-distribution with  $a$  and  $b$  degrees of freedom.

Note: The univariate version of the noncentral distribution of Wilks Lambda: GLM is equivalent to  $W = 1 - B$

See [Tiwari & Yang \(1997\)](#)

#### 33.1.2 Density and CDF

---

Function **NoncentralBetaDistBoost**(*x* As mpNum, *m* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralBetaDistBoost returns pdf, CDF and related information for the central Beta-distribution

##### Parameters:

*x*: A real number

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

##### 33.1.2.1 Density

The density function of the noncentral beta-Distribution is given by ([Wang & Gray, 1993](#)): this needs to be checked, see Paolella 2007:

$$f_{\text{Beta}'}(x; n_1, n_2, \lambda) = e^{-\lambda/2} f_B(x; n_1, n_2) {}_1F_1\left(\frac{1}{2}(m+n), \frac{1}{2}n, \frac{nx\lambda}{2(m+nx)}\right) \quad (33.1.1)$$

### 33.1.2.2 CDF: Infinite Series

The cdf can be calculated using the following infinite series [Benton & Krishnamoorthy \(2003\)](#):

$$\Pr[F \leq x] = F_{B'}(x; a, b, \lambda) = e^{-\lambda/2} \sum_{j=0}^{\infty} \frac{(\lambda/2)^j}{j!} I(a + j, b, x) \quad (33.1.2)$$

### 33.1.3 Quantiles

---

Function **NoncentralBetaDistInvBoost**(*Prob* As mpNum, *m* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralBetaDistInvBoost returns returns quantiles and related information for the the noncentral Beta-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section [32.1.3.2](#) for the options for *Prob* and *Output*). Algorithms and formulas are given below.

### 33.1.4 Properties

---

Function **NoncentralBetaDistInfoBoost**(*m* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralBetaDistInfoBoost returns returns moments and related information for the noncentral Beta-distribution

**Parameters:**

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section [32.1.3.3](#) for the options for *Output*. Algorithms and formulas are given below.

#### 33.1.4.1 Moments of the non-central Beta-distribution

Currently, Boost does not calculate the moments of the noncentral beta distribution.

### 33.1.5 Random Numbers

---

Function **NoncentralBetaDistRanBoost**(*Size* As mpNum, *m* As mpNum, *n* As mpNum, *lambda* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function NoncentralBetaDistRanBoost returns returns random numbers following a noncentral Beta-distribution

**Parameters:***Size*: A positive integer up to  $10^7$ *m*: A real number greater 0, representing the numerator degrees of freedom*n*: A real number greater 0, representing the denominator degrees of freedom*lambda*: A real number greater 0, representing the noncentrality parameter*Generator*: A string describing the random generator*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

Random numbers from a non-central Beta-distribution with integer or half-integer  $p$  and  $q$  values is easily obtained using the definition above i.e. by using a random number from a non-central chi-square distribution and another from a (central) chi-square distribution.

## 33.2 Noncentral Chi-Square Distribution

### 33.2.1 Definition

Let  $X_1, X_2, \dots, X_n$  be independent and identically distributed random variables each following a normal distribution with mean  $\mu_j$  and unit variance. Then  $\chi^2 = \sum_{j=1}^n X_j$  is said to follow a non-central  $\chi^2$ -distribution with  $n$  degrees of freedom and noncentrality parameter  $\lambda = \sum_{j=1}^n (\mu_j - \mu)$ .

### 33.2.2 Density and CDF

---

Function **NoncentralCDistBoost**(*x* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function NoncentralCDistBoost returns returns pdf, CDF and related information for the noncentral  $\chi^2$ -distribution

**Parameters:***x*: A real number*n*: A real number greater 0, representing the degrees of freedom*lambda*: A real number greater 0, representing the noncentrality parameter*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*.

#### 33.2.2.1 CDF: General formulas

The cdf of a noncentral chi-square variable with  $n$  degrees of freedom and  $\lambda$  is given by

$$\Pr [\chi^2 \leq x] = F_{\chi^2}(n, x; \lambda) = \int_0^x f_{\chi^2}(n, t; \lambda) dt \quad (33.2.1)$$

#### 33.2.2.2 CDF: Infinite series in terms of the central cdf

The cdf of a noncentral chi-square variable with  $n$  degrees of freedom and  $\lambda$  is given by

$$F_{\chi^2}(n, x; \lambda) = e^{-\lambda/2} \sum_{j=0}^{\infty} \frac{(\lambda/2)^j}{j!} F_{\chi^2}(n + 2 + j, x) \quad (33.2.2)$$

### 33.2.2.3 CDF: Infinite series in terms of the central pdf

Ding (1992) gives the following representation (this is used by Boost for small lambda):

$$F_{\chi^2}(n, x; \lambda) = 2e^{-\lambda/2} \sum_{i=0}^{\infty} f_{\chi^2}(n + 2 + 2i, x) \left( \sum_{k=0}^i \frac{(\lambda/2)^k}{k!} \right) \quad (33.2.3)$$

## 33.2.3 Quantiles

---

Function **NoncentralCDistInvBoost**(*Prob* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralCDistInvBoost returns quantiles and related information for the noncentral  $\chi^2$ -distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below

The quantile is approximated as

$$\chi_{n,\lambda,\alpha}^2 \approx (1 + b)\chi_{n_1,\lambda,\alpha}^2, \quad \text{where } n_1 = \frac{(n + \lambda)^2}{n + 2\lambda}, \quad b = \frac{\lambda}{n + \lambda} \quad (33.2.4)$$

## 33.2.4 Properties

---

Function **NoncentralCDistInfoBoost**(*n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralCDistInfoBoost returns moments and related information for the noncentral  $\chi^2$ -distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

### 33.2.4.1 Moments and Cumulants

The cumulants of noncentral  $\chi^2$  are given by

$$\kappa_r(n, \lambda) = 2^{r-1}(r-1)!(n + r\lambda) \quad (33.2.5)$$

### 33.2.5 Random Numbers

---

Function **NoncentralCDistRanBoost**(*Size* As mpNum, *n* As mpNum, *lambda* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

The function NoncentralCDistRanBoost returns random numbers following a noncentral  $\chi^2$ -distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below

Random numbers from a non-central chi-square distribution is easily obtained using the definition above by e.g.

1. Put  $\mu = \sqrt{\lambda/n}$
2. Sum  $n$  random numbers from a normal distribution with mean  $\mu$  and variance unity. Note that this is not a unique choice. The only requirement is that  $\lambda = \sum \mu_i^2$ .
3. Return the sum as a random number from a non-central chi-square distribution with  $n$  degrees of freedom and non-central parameter  $\lambda$ .

This ought to be sufficient for most applications but if needed more efficient techniques may easily be developed e.g. using more general techniques.

## 33.3 NonCentral F-Distribution

### 33.3.1 Definition

If  $X_1$  and  $X_2$  are independent random variables,  $X_1$  following a non-central  $\chi^2$ -distribution with noncentrality parameter  $\lambda$  and  $m$  degrees of freedom, and  $X_2$  following a  $\chi^2$ -distribution with  $n$  degrees of freedom, then the distribution of the ratio  $F = \frac{X_1/m}{X_2/n}$  is said to follow a non-central F-distribution with noncentrality parameter  $\lambda$  and  $m$  and  $n$  degrees of freedom.

### 33.3.2 Density and CDF

---

Function **NoncentralFDistBoost**(*x* As mpNum, *m* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralFDistBoost returns pdf, CDF and related information for the noncentral  $F$ -distribution

**Parameters:**

*x*: A real number

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

### 33.3.2.1 Density

### 33.3.2.2 CDF (singly noncentral: Infinite Series)

The cdf of a variable following a (singly) noncentral F-distribution with  $n$  and  $m$  degrees of freedom and noncentrality parameter  $\lambda_1$  and is given by

$$\Pr[F \leq x] = F_{F'}(x; m, n, \lambda) = e^{-\lambda} \sum_{j=0}^{\infty} \frac{(\lambda/2)^j}{j!} F(m + 2j, n, x) \quad (33.3.1)$$

### 33.3.3 Quantiles

---

Function **NoncentralFDistInvBoost**(*Prob* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function NoncentralFDistInvBoost returns returns quantiles and related information for the the noncentral  $F$ -distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

### 33.3.4 Properties

---

Function **NoncentralFDistInfoBoost**(*m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function NoncentralFDistInfoBoost returns returns moments and related information for the noncentral  $F$ -distribution

**Parameters:**

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 33.3.4.1 Moments (singly noncentral)

The algebraic moments (defined for  $f_2 > 2r$ ) are given by

$$\mu'_r = \frac{\Gamma(\frac{1}{2}f_1 + r) - \Gamma(\frac{1}{2}f_2 - r)}{\Gamma(\frac{1}{2}f_2)} \sum_{j=0}^r \binom{r}{j} \frac{\frac{1}{2}\lambda f_1)^j}{\Gamma(\frac{1}{2}f_1 + j)}, \quad \text{for } f_2 > 2r. \quad (33.3.2)$$

The first four raw moments (defined for  $n > 2k$ ) are given by

$$\begin{aligned}\mu'_1 &= \frac{n}{m} \frac{m + \lambda}{n - 2} \\ \mu'_2 &= \left(\frac{n}{m}\right)^2 \frac{\lambda^2 + (2\lambda + m)(m + 2)}{(n - 2)(n - 4)} \\ \mu'_3 &= \left(\frac{n}{m}\right)^3 \frac{\lambda^3 + 3(m + 4)\lambda^2 + (3\lambda + m)(m + 4)(m + 2)}{(n - 2)(n - 4)(n - 6)} \\ \mu'_4 &= \left(\frac{n}{m}\right)^4 \frac{\lambda^3 + 4(m + 6)\lambda^3 + 6(m + 6)(m + 4)\lambda^2 + (4\lambda + m)(m + 6)(m + 4)(m + 2)}{(n - 2)(n - 4)(n - 6)(n - 8)}\end{aligned}$$

### 33.3.4.2 Relationships to other distributions (singly noncentral)

$$F_{F'}(x; m, n, \lambda) = F_B\left(\frac{1}{2}n, \frac{1}{2}m, \frac{mx}{mx + n}; \lambda\right) \quad (33.3.3)$$

## 33.3.5 Random Numbers

---

Function **NoncentralFDistRanBoost**(*Size* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **NoncentralFDistRanBoost** returns random numbers following a noncentral *F*-distribution

#### Parameters:

*Size*: A positive integer up to  $10^7$

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

Random numbers from a non-central F-distribution is easily obtained using the definition in terms of the ratio between two independent random numbers from central and non-central chi-square distributions. This ought to be sufficient for most applications but if needed more efficient techniques may easily be developed e.g. using more general techniques.

## 33.4 Noncentral Student's t-Distribution

### 33.4.1 Definition

If  $X$  is a random variable following a normal distribution with mean  $\delta$  and variance unity and  $\chi^2$  is a random variable following an independent  $\chi^2$ -distribution with  $n$  degrees of freedom, then the distribution of the ratio  $\frac{X}{\sqrt{\chi^2/n}}$  is called noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$ .

### 33.4.2 Density and CDF

---

Function **NoncentralTDistBoost**(*x* As mpNum, *n* As mpNum, *delta* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralTDistBoost returns returns pdf, CDF and related information for the noncentral *t*-distribution

**Parameters:**

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 37.8.2 and 37.8.2.3.

#### 33.4.2.1 Density (singly noncentral): Infinite series

The pdf of a variable following a noncentral t-distribution with *n* degrees of freedom and noncentrality parameter  $\delta$  is given by (Bristow *et al.*, 2013)

$$f_{t'}(n, x, \delta) = \frac{nt}{n^2 + 2nt^2 + t^4} + \frac{1}{2} \sum_{i=0}^{\infty} P_i I'_x \left( i + \frac{1}{2}, \frac{n}{2} \right) + \frac{\delta}{\sqrt{2}} Q_i I'_x \left( i + 1, \frac{n}{2} \right), \quad \text{and} \quad (33.4.1)$$

$I'_x(\cdot, \cdot)$  denotes the derivative of the (normalized) incomplete beta function (see section ??), and  $P_i$  and  $Q_i$  are defined in equation 37.8.15.

#### 33.4.2.2 CDF (singly noncentral): Infinite series

The cdf of a variable following a noncentral t-distribution with *n* degrees of freedom and noncentrality parameter  $\delta$  is given by (Benton & Krishnamoorthy, 2003; Bristow *et al.*, 2013)

$$F_{t'}(n, x, \delta) = \Phi(-\delta) + \frac{1}{2} \sum_{i=0}^{\infty} P_i I_x \left( i + \frac{1}{2}, \frac{n}{2} \right) + \frac{\delta}{\sqrt{2}} Q_i I_x \left( i + 1, \frac{n}{2} \right), \quad \text{and} \quad (33.4.2)$$

$$1 - F_{t'}(n, x, \delta) = \frac{1}{2} \sum_{i=0}^{\infty} P_i I_y \left( \frac{n}{2}, i + \frac{1}{2} \right) + \frac{\delta}{\sqrt{2}} Q_i I_y \left( \frac{n}{2}, i + 1 \right), \quad \text{where} \quad (33.4.3)$$

$$\lambda = \frac{1}{2}\delta^2; \quad P_i = \frac{e^{-\lambda}\lambda^i}{i!}; \quad Q_i = \frac{e^{-\lambda}\lambda^i}{\Gamma(i + 3/2)}; \quad x = \frac{t^2}{n + t^2}; \quad y = 1 - x, \quad (33.4.4)$$

### 33.4.3 Quantiles

---

Function **NoncentralTDistInvBoost**(*Prob* As mpNum, *n* As mpNum, *delta* As mpNum, *Output* As String) As mpNumList

---

The function NoncentralTDistInvBoost returns quantiles and related information for the noncentral *t*-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below:

### 33.4.4 Properties

---

Function **NoncentralTDistInfoBoost**(*n* As *mpNum*, *delta* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function NoncentralTDistInfoBoost returns moments and related information for the noncentral *t*-distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 33.4.4.1 Moments (singly noncentral)

The algebraic moments (defined for  $n > r$ ) are given by

$$\mu'_r = \left(\frac{1}{2}n\right)^{r/2} \frac{\Gamma\left(\frac{1}{2}(n-r)\right)}{\Gamma\left(\frac{1}{2}n\right)} \sum_{i=0}^{\lfloor r/2 \rfloor} \binom{r}{2i} \frac{(2i)!}{2^i i!} \delta^{r-2i}. \quad (33.4.5)$$

The first four raw moments are given by

$$E(t) = \delta \sqrt{\frac{1}{2}n} \frac{\Gamma\left(\frac{1}{2}(n-1)\right)}{\Gamma\left(\frac{1}{2}n\right)} \quad (33.4.6)$$

$$E(t^2) = (\delta^2 + 1) \frac{n}{n-2} \quad (33.4.7)$$

$$E(t^3) = \delta(\delta^2 + 3) \sqrt{\frac{1}{8}n^3} \frac{\Gamma\left(\frac{1}{2}(n-3)\right)}{\Gamma\left(\frac{1}{2}n\right)} \quad (33.4.8)$$

$$E(t^4) = (\delta^4 + 6\delta^2 + 3) \frac{n^2}{(n-2)(n-4)} \quad (33.4.9)$$

### 33.4.5 Random Numbers

---

Function **NoncentralTDistRanBoost**(*Size* As *mpNum*, *n* As *mpNum*, *delta* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function NoncentralTDistRanBoost returns random numbers following a noncentral *t*-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below:

Random numbers from a non-central t-distribution is easily obtained using the definition in terms of the ratio between two independent random numbers from a normal and a central chi-square distribution. This ought to be sufficient for most applications but if needed more efficient techniques may easily be developed e.g. using more general techniques.

## 33.5 Skew Normal Distribution

### 33.5.1 Definition

In probability theory and statistics, the skew normal distribution is a continuous probability distribution that generalises the normal distribution to allow for non-zero skewness.

See also [http://en.wikipedia.org/wiki/Skew\\_normal\\_distribution](http://en.wikipedia.org/wiki/Skew_normal_distribution).

### 33.5.2 Density and CDF

---

Function **SkewNormalDistBoost**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *Output* As *String*) As *mpNumList*

---

The function **SkewNormalDistBoost** returns returns pdf, CDF and related information for the skew normal distribution

#### Parameters:

*x*: A real number.

*a*: The location parameter.

*b*: The scale parameter

*c*: The shape parameter

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 37.3.2.1 and 37.3.2.2.

#### 33.5.2.1 Density

The probability density function with location parameter  $\xi$ , scale parameter  $\omega$ , and shape parameter  $\alpha$  is

$$f(\xi; \omega, \alpha) = \frac{2}{\omega} \phi\left(\frac{x - \xi}{\omega}\right) \Phi\left(\alpha\left(\frac{x - \xi}{\omega}\right)\right) \quad (33.5.1)$$

The probability density function with location parameter  $a$ , scale parameter  $b$ , and shape parameter  $c$  is

$$f(x; a, b, c) = \frac{2}{b} \phi\left(\frac{x - a}{b}\right) \Phi\left(c\left(\frac{x - a}{b}\right)\right) \quad (33.5.2)$$

### 33.5.2.2 CDF

$$F(\xi; \omega, \alpha) = \Phi\left(\frac{x - \xi}{\omega}\right) - 2T\left(\frac{x - \xi}{\omega}, \alpha\right) \quad (33.5.3)$$

$$F(a, b, c) = \Phi\left(\frac{x - a}{b}\right) - 2T\left(\frac{x - a}{b}, c\right) \quad (33.5.4)$$

### 33.5.3 Quantiles

---

Function **SkewNormalDistInvBoost**(*Prob* As mpNum, *a* As mpNum, *b* As mpNum, *c* As mpNum, *Output* As String) As mpNumList

---

The function **SkewNormalDistInvBoost** returns returns quantiles and related information for the the skew normal distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*a*: The location parameter.

*b*: The scale parameter

*c*: The shape parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). The quantile is determined using an iterative alogorithm.

### 33.5.4 Properties

---

Function **SkewNormalDistInfoBoost**(*a* As mpNum, *b* As mpNum, *c* As mpNum, *Output* As String) As mpNumList

---

The function **SkewNormalDistInfoBoost** returns returns moments and related information for the skew normal distribution

**Parameters:**

*a*: The location parameter.

*b*: The scale parameter

*c*: The shape parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 33.5.4.1 Moments

$$\mu_1 = a + bd\sqrt{\frac{2}{\pi}}, \quad \text{where } d = \frac{c}{\sqrt{1+c^2}} \quad (33.5.5)$$

$$\mu_2 = b^2 \left(1 - \frac{2d^2}{\pi}\right) \quad (33.5.6)$$

$$\gamma_1 = \frac{4 - \pi}{2} \frac{\left(d\sqrt{2/\pi}\right)^3}{(1 - 2d^2/\pi)^{3/2}} \quad (33.5.7)$$

$$\gamma_2 = 2(\pi - 3) \frac{(d\sqrt{2/\pi})^4}{(1 - 2d^2/\pi)^2} \quad (33.5.8)$$

### 33.5.5 Random Numbers

---

Function **SkewNormalDistRanBoost**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

The function **SkewNormalDistRanBoost** returns random numbers following a skew normal distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: The location parameter.

*b*: The scale parameter

*c*: The shape parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

## 33.6 Owen's T-Function

### 33.6.1 Owen's T-Function

---

Function **TOwenBoost**(*h* As *mpNum*, *a* As *mpNum*) As *mpNum*

---

The function **TOwenBoost** returns Owen's T-Function

**Parameters:**

*h*: A real number.

*a*: A real number.

Owen's T-Function is defined as (Owen, 1956):

$$T(h, a) = \frac{1}{2\pi} \int_0^a \frac{\exp\left[-\frac{1}{2}h^2(1+x^2)\right]}{1+x^2} dx \quad (33.6.1)$$

The implementation uses the algorithm described in Patefield & Tand (2000).

# Chapter 34

## Ordinary Differential Equations

The procedures in this chapter are based on Boost.Numeric.Odeint (see [Ahnert & Mulansky \(2013\)](#)), a library for solving initial value problems (IVP) of ordinary differential equations. Mathematically, these problems are formulated as follows:

$$x'(t) = f(x, t), x(0) = x_0.$$

$x$  and  $f$  can be vectors and the solution is some function  $x(t)$  fulfilling both equations above. In the following we will refer to  $x'(t)$  also  $dx/dt$  which is also our notation for the derivative in the source code.

Ordinary differential equations occur nearly everywhere in natural sciences. For example, the whole Newtonian mechanics are described by second order differential equations. Be sure, you will find them in every discipline. They also occur if partial differential equations (PDEs) are discretized. Then, a system of coupled ordinary differential occurs, sometimes also referred as lattices ODEs.

Numerical approximations for the solution  $x(t)$  are calculated iteratively. The easiest algorithm is the Euler scheme, where starting at  $x(0)$  one finds  $x(dt) = x(0) + dt f(x(0), 0)$ . Now one can use  $x(dt)$  and obtain  $x(2dt)$  in a similar way and so on. The Euler method is of order 1, that means the error at each step is  $\approx dt^2$ . This is, of course, not very satisfactory, which is why the Euler method is rarely used for real life problems and serves just as illustrative example.

The main focus of odeint is to provide numerical methods implemented in a way where the algorithm is completely independent on the data structure used to represent the state  $x$ . In doing so, odeint is applicable for a broad variety of situations and it can be used with many other libraries. Besides the usual case where the state is defined as a `std::vector` or a `boost::array`, we provide native support for the following libraries:

General Literature includes:

General information about numerical integration of ordinary differential equations:

[Press \*et al.\* \(2007\)](#)

[Hairer \*et al.\* \(2009\)](#)

[Hairer & Wanner \(2010\)](#)

Symplectic integration of numerical integration:

[Hairer \*et al.\* \(2006\)](#)

[Leimkuhler & Reich \(2005\)](#)

Special symplectic methods:

[Yoshida \(1990\)](#)

[McLachlan \(1995\)](#)

Special systems:

Fermi-Pasta-Ulam nonlinear lattice oscillations

[Pikovsky \*et al.\* \(2001\)](#)

## 34.1 Defining the ODE System

The routines solve the general n-dimensional first-order system,

$$\frac{dy_i(t)}{dt} = f_i(t, y_1(t), \dots, y_n(t)) \quad (34.1.1)$$

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 system datatype.

## 34.2 Stepping Functions

Solving ordinary differential equation numerically is usually done iteratively, that is a given state of an ordinary differential equation is iterated forward  $x(t) \rightarrow x(t + dt) \rightarrow x(t + 2dt)$ . The steppers in `odeint` perform one single step. The most general stepper type is described by the Stepper concept. The stepper concepts of `odeint` are described in detail in section Concepts, here we briefly present the mathematical and numerical details of the steppers. The Stepper has two versions of the `do_step` method, one with an in-place transform of the current state and one with an out-of-place transform:

```
do_step( sys , inout , t , dt )
do_step( sys , in , t , out , dt )
```

The first parameter is always the system function - a function describing the ODE. In the first version the second parameter is the step which is here updated in-place and the third and the fourth parameters are the time and step size (the time step). After a call to `do_step` the state `inout` is updated and now represents an approximate solution of the ODE at time  $t+dt$ . In the second version the second argument is the state of the ODE at time  $t$ , the third argument is  $t$ , the fourth argument is the approximate solution at time  $t+dt$  which is filled by `do_step` and the fifth argument is the time step. Note that these functions do not change the time  $t$ .

System functions

Up to now, we have nothing said about the system function. This function depends on the stepper. For the explicit Runge-Kutta steppers this function can be a simple callable object hence a simple (global) C-function or a functor. The parameter syntax is `sys(x, dxdt, t)` and it is assumed that it calculates  $dx/dt = f(x, t)$ . The function structure in most cases looks like:

```
void sys( const state_type & /*x*/ , state_type & /*dxdt*/ , const double /*t*/ ) // ...
```

Other types of system functions might represent Hamiltonian systems or systems which also compute the Jacobian needed in implicit steppers. For information which stepper uses which system function see the stepper table below. It might be possible that `odeint` will introduce new system types in near future. Since the system function is strongly related to the stepper type, such an introduction of a new stepper might result in a new type of system function.

Explicit steppers A first specialization are the explicit steppers. Explicit means that the new state of the ode can be computed explicitly from the current state without solving implicit equations. Such steppers have in common that they evaluate the system at time  $t$  such that the result of  $f(x, t)$  can be passed to the stepper. In `odeint`, the explicit stepper have two additional methods `Which` steppers should be used in which situation

`odeint` provides a quite large number of different steppers such that the user is left with the question of which stepper fits his needs. Our personal recommendations are:

`runge_kutta_dopri5` is maybe the best default stepper. It has step size control as well as dense-output functionality. Simple create a dense-output stepper by `make_dense_output( 1.0e-6 , 1.0e-5 , runge_kutta_dopri5 )`. `state_type` `i()`. `runge_kutta4` is a good stepper for constant step sizes. It

is widely used and very well known. If you need to create artificial time series this stepper should be the first choice. 'runge\_kutta\_fehlberg78' is similar to the 'runge\_kutta4' with the advantage that it has higher precision. It can also be used with step size control. adams\_bashforth\_moulton is very well suited for ODEs where the r.h.s. is expensive (in terms of computation time). It will calculate the system function only once during each step.

### 34.2.1 Explicit Euler

In mathematics and computational science, the Euler method is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value. It is the most basic explicit method for numerical integration of ordinary differential equations and is the simplest Runge-Kutta method. The Euler method is named after Leonhard Euler, who treated it in his book *Institutionum calculi integralis* (published 1768-70).[1]

The Euler method is a first-order method, which means that the local error (error per step) is proportional to the square of the step size, and the global error (error at a given time) is proportional to the step size. The Euler method often serves as the basis to construct more complicated methods.

### 34.2.2 Modified Midpoint

In numerical analysis, a branch of applied mathematics, the midpoint method is a one-step method for numerically solving the differential equation,

$$y'(t) = f(t, y(t)), y(t_0) = y_0$$

and is given by the formula

$$y_{n+1} = y_n + h f(t_n + h/2, y_n + (h/2) f(t_n, y_n)),$$

for  $n = 0, 1, 2, \dots$ . Here,  $h$  is the step size - a small positive number,  $t_n = t_0 + nh$  and  $y_n$  is the computed approximate value of  $y(t_n)$ . The midpoint method is also known as the modified Euler method.[1]

The name of the method comes from the fact that in the formula above the function  $f$  is evaluated at  $t = t_n + h/2$  which is the midpoint between  $t_n$  at which the value of  $y(t)$  is known and  $t_{n+1}$  at which the value of  $y(t)$  needs to be found.

The local error at each step of the midpoint method is of order  $O(h^3)$ , giving a global error of order  $O(h^2)$ . Thus, while more computationally intensive than Euler's method, the midpoint method generally gives more accurate results.

The method is an example of a class of higher-order methods known as Runge-Kutta methods.

### 34.2.3 Runge-Kutta 4

In numerical analysis, the Runge-Kutta methods are an important family of implicit and explicit iterative methods, which are used in temporal discretization for the approximation of solutions of ordinary differential equations. These techniques were developed around 1900 by the German mathematicians C. Runge and M. W. Kutta.

See the article on numerical methods for ordinary differential equations for more background and other methods. See also List of Runge-Kutta methods.

One member of the family of Runge-Kutta methods is often referred to as "RK4", "classical Runge-Kutta method" or simply as "the Runge-Kutta method".

Let an initial value problem be specified as follows.

Here,  $y$  is an unknown function (scalar or vector) of time  $t$  which we would like to approximate; we are told that , the rate at which  $y$  changes, is a function of  $t$  and of  $y$  itself. At the initial time the corresponding  $y$ -value is . The function  $f$  and the data , are given.

### 34.2.4 Cash-Karp

In numerical analysis, the Cash-Karp method is a method for solving ordinary differential equations (ODEs). It was proposed by Professor Jeff R. Cash [1] from Imperial College London and Alan H. Karp from IBM Scientific Center. The method is a member of the Runge-Kutta family of ODE solvers. More specifically, it uses six function evaluations to calculate fourth- and fifth-order accurate solutions. The difference between these solutions is then taken to be the error of the (fourth order) solution. This error estimate is very convenient for adaptive stepsize integration algorithms. Other similar integration methods are Fehlberg (RKF) and Dormand-Prince (RKDP).

J. R. Cash, A. H. Karp. "A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides", ACM Transactions on Mathematical Software 16: 201-222, 1990. doi:10.1145/79505.79507.

Shampine, Lawrence F. (1986), "Some Practical Runge-Kutta Formulas", Mathematics of Computation (American Mathematical Society) 46 (173): 135–150, doi:10.2307/2008219, JSTOR 2008219 .

### 34.2.5 Dormand-Prince 5

In numerical analysis, the Dormand-Prince method, or DOPRI method, is an explicit method for solving ordinary differential equations (Dormand & Prince, 1980). The method is a member of the Runge-Kutta family of ODE solvers. More specifically, it uses six function evaluations to calculate fourth- and fifth-order accurate solutions. The difference between these solutions is then taken to be the error of the (fourth-order) solution. This error estimate is very convenient for adaptive stepsize integration algorithms. Other similar integration methods are Fehlberg (RKF) and Cash-Karp (RKCK).

The Dormand-Prince method has seven stages, but it uses only six function evaluations per step because it has the FSAL (First Same As Last) property: the last stage is evaluated at the same point as the first stage of the next step. Dormand and Prince choose the coefficients of their method to minimize the error of the fifth-order solution. This is the main difference with the Fehlberg method, which was constructed so that the fourth-order solution has a small error. For this reason, the Dormand-Prince method is more suitable when the higher-order solution is used to continue the integration, a practice known as local extrapolation (Shampine 1986; Hairer, Nørsett & Wanner 2008, pp. 178–179).

Dormand-Prince is currently the default method in MATLAB and GNU Octave's ode45 solver and is the default choice for the Simulink's model explorer solver. A Fortran free software implementation of the algorithm called DOPRI5 is also available.[1]

### 34.2.6 Fehlberg 78

In mathematics, the Runge-Kutta-Fehlberg method (or Fehlberg method) is an algorithm in numerical analysis for the numerical solution of ordinary differential equations. It was developed by the German mathematician Erwin Fehlberg and is based on the large class of Runge-Kutta methods.

The novelty of Fehlberg's method is that it is an embedded method from the Runge-Kutta family, meaning that identical function evaluations are used in conjunction with each other to create methods of varying order and similar error constants. The method presented in Fehlberg's 1969 paper has been dubbed the RKF45 method, and is a method of order  $O(h^4)$  with an error estimator of order  $O(h^5)$ .<sup>[1]</sup> By performing one extra calculation, the error in the solution can be estimated and controlled by using the higher-order embedded method that allows for an adaptive stepsize to be determined automatically.

Erwin Fehlberg (1970). "Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wärmeleitungssprobleme," Computing (Arch. Elektron. Rechnen), vol. 6, pp. 61–71. doi:10.1007/BF02241732

### 34.2.7 Adams-Bashforth

Three families of linear multistep methods are commonly used: Adams-Bashforth methods, Adams-Moulton methods, and the backward differentiation formulas (BDFs).

Adams-Bashforth methods[edit]  
The Adams-Bashforth methods are explicit methods. The coefficients are and , while the are chosen such that the methods has order s (this determines the methods uniquely).

The Adams-Bashforth methods with  $s = 1, 2, 3, 4, 5$  are (Hairer, Nørsett & Wanner 1993, §III.1; Butcher 2003, p. 103):

### 34.2.8 Adams-Moulton

The Adams-Moulton methods are similar to the Adams-Bashforth methods in that they also have and . Again the b coefficients are chosen to obtain the highest order possible. However, the Adams-Moulton methods are implicit methods. By removing the restriction that , an s-step Adams-Moulton method can reach order , while an s-step Adams-Bashforth methods has only order s.

The Adams-Moulton methods with  $s = 0, 1, 2, 3, 4$  are (Hairer, Nørsett & Wanner 1993, §III.1; Quarteroni, Sacco & Saleri 2000):

### 34.2.9 Adams-Bashforth-Moulton

The methods of Euler, Heun, Taylor and Runge-Kutta are called single-step methods because they use only the information from one previous point to compute the successive point, that is, only the initial point is used to compute and in general is needed to compute . After several points have been found it is feasible to use several prior points in the calculation. The Adams-Bashforth-Moulton method uses in the calculation of . This method is not self-starting; four initial points , , , and must be given in advance in order to generate the points .

A desirable feature of a multistep method is that the local truncation error (L. T. E.) can be determined and a correction term can be included, which improves the accuracy of the answer at each step. Also, it is possible to determine if the step size is small enough to obtain an accurate value for , yet large enough so that unnecessary and time-consuming calculations are eliminated. If the code for the subroutine is fine-tuned, then the combination of a predictor and corrector requires only two function evaluations of  $f(t,y)$  per step.

See also: <http://mathfaculty.fullerton.edu/mathews/n2003/AdamsBashforthMod.html>

### 34.2.10 Controlled Runge-Kutta

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 34.2.11 Dense Output Runge-Kutta

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 34.2.12 Bulirsch-Stoer

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

### 34.2.13 Bulirsch-Stoer Dense Output

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

### 34.2.14 Implicit Euler

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 34.2.15 Rosenbrock 4

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et

nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 34.2.16 Controlled Rosenbrock 4

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 34.2.17 Dense Output Rosenbrock 4

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 34.2.18 Symplectic Euler

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

### 34.2.19 Symplectic RKN McLachlan

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

## 34.3 Integrate functions: Evolution

Integrate functions perform the time evolution of a given ODE from some starting time  $t_0$  to a given end time  $t_1$  and starting at state  $x_0$  by subsequent calls of a given stepper's `do_step` function.

Additionally, the user can provide an observer to analyze the state during time evolution. There are five different integrate functions which have different strategies on when to call the observer function during integration.

All of the `integrate` functions except `integrate_n_steps` can be called with any stepper following one of the stepper concepts: `Stepper` , `Error Stepper` , `Controlled Stepper` , `Dense Output Stepper`. Depending on the abilities of the stepper, the `integrate` functions make use of step-size control or dense output.

## **Part VII**

# **Application Examples**

# Chapter 35

## Examples: Multivariate Special Functions

### 35.1 Functions Of Matrix Arguments

#### 35.1.1 Multivariate Gamma function $\Gamma_p(x)$

---

Function **Gammap**(*p* As mpNum, *x* As mpNum) As mpNum

---

**NOT YET IMPLEMENTED**

---

The function **Gammap** returns the multivariate gamma function.

**Parameters:**

*p*: An integer greater than 0.

*x*: A real number or an array of real numbers.

The multivariate gamma function is defined by

$$\Gamma_p(x) = \pi^{p(p-1)/4} \prod_{i=1}^p \Gamma\left(x - \frac{1}{2}(i-1)\right), \quad \Re(x) > \frac{1}{2}(m-1) \quad (35.1.1)$$

$$\Gamma_p(s_1, \dots, s_p) = \pi^{p(p-1)/4} \prod_{i=1}^p \Gamma\left(s_i - \frac{1}{2}(i-1)\right), \quad \Re(s_i) > \frac{1}{2}(i-1), \quad j = 1, \dots, m. \quad (35.1.2)$$

$$\Gamma_p(a, \dots, a) = \Gamma_p(a) \quad (35.1.3)$$

where  $\Gamma(\cdot)$  is the usual (scalar) gamma function (see section ??).

### 35.1.2 Zonal Polynomials

#### 35.1.2.1 Definitions

A partition  $\kappa = (k_1, \dots, k_m)$  is a vector of nonnegative integers, listed in nonincreasing order. Also,  $|\kappa|$  denotes  $k_1 + \dots + k_m$ , the weight of  $\kappa$ ;  $\ell(\kappa)$  denotes the number of nonzero  $k_j$ ;  $a + \kappa$  denotes the vector  $(a + k_1, \dots, a + k_m)$ .

The partitional shifted factorial is given by (Olver *et al.*, 2010)

$$[a]_\kappa = \frac{\Gamma_m(a + \kappa)}{\Gamma_m(a)} = \prod_{j=1}^{\ell(\kappa)} \left( a - \frac{1}{2}(j-1) \right)_{k_j}, \quad (35.1.4)$$

where  $(a)_k$  is the Pochammer symbol (see section 7.3).

For any partition  $\kappa$ , the zonal polynomial  $Z_\kappa : \mathbf{S} \rightarrow \mathbb{R}$  is defined by the properties (Olver *et al.*, 2010)

$$Z_\kappa(\mathbf{I}) = |\kappa|! 2^{2|\kappa|} [m/2]_\kappa \frac{\prod_{\substack{1 \leq j < l \leq \ell(\kappa) \\ \ell(\kappa)}} (2k_j - 2k_l - j + l)}{\prod_{j=1}^{\ell(\kappa)} (2k_j + \ell(\kappa) - j)!}, \quad \text{and} \quad (35.1.5)$$

$$Z_\kappa(\mathbf{T}) = Z_\kappa(\mathbf{I}) |\mathbf{T}|^{k_m} \int_{\mathbf{O}(m)} \prod_{j=1}^{m-1} |(\mathbf{HTH}^{-1})_j|^{k_j - k_{j+1}} d\mathbf{H} \quad (35.1.6)$$

See Muirhead (1982), pp. 68-72, for the definition and properties of the Haar measure  $d\mathbf{H}$ . Alternative notations for the zonal polynomials are  $C_\kappa(\mathbf{T})$  Muirhead (1982), pp. 227-239.

#### 35.1.2.2 Properties

Zonal polynomials have the following properties (Olver *et al.*, 2010):

$$Z_\kappa(0) = \begin{cases} 1, & \kappa = (0, \dots, 0), \\ 0, & \kappa \neq (0, \dots, 0). \end{cases} \quad (35.1.7)$$

$$Z_\kappa(\mathbf{HTH}^{-1}) = Z_\kappa(\mathbf{T}), \quad \mathbf{H} \in \mathbf{O}(m). \quad (35.1.8)$$

Therefore  $Z_\kappa(\mathbf{T})$  is a symmetric polynomial in the eigenvalues of  $\mathbf{T}$ . Also, for  $k = 0, 1, 2, \dots$ ,

$$\sum_{|\kappa|=k} Z_\kappa(\mathbf{T}) = (\text{tr } \mathbf{T})^k. \quad (35.1.9)$$

#### 35.1.2.3 Implementation

Gupta & Richards (1979) describe an algorithm for the calculation of zonal polynomials for  $2 \times 2$  and  $3 \times 3$  matrices.

James (1968, 1964); McLaren (1976) describe a general algorithm.

### 35.1.3 Gauss Hypergeometric Function of Matrix Argument

---

Function **Hypergeometric2F1Matrix**(*a* As mpNum, *b* As mpNum, *c* As mpNum, *T* As mpNum)  
As mpNum

---

NOT YET IMPLEMENTED

---

The function `Hypergeometric2F1Matrix` returns the Gauss hypergeometric function for matrix argument.

**Parameters:**

*a*: A real number.

*b*: A real number.

*c*: A real number.

*T*: A real matrix.

The Gauss hypergeometric function for matrix argument,  ${}_2F_1(a, b; c; \mathbf{T})$  is defined by (Olver *et al.*, 2010)

$${}_2F_1(a, b; c; \mathbf{T}) = \sum_{k=0}^{\infty} \frac{1}{k!} \sum_{|\kappa|=k} \frac{[a]_{\kappa} [b]_{\kappa}}{[c]_{\kappa}} Z_{\kappa}(\mathbf{T}) \quad (35.1.10)$$

with  $-c + \frac{1}{2}(j+1) \notin \mathbb{N}$ ,  $1 \leq j \leq m$ ;  $\|\mathbf{T}\| < 1$ . Here  $Z_{\kappa}(\mathbf{T})$  is a zonal polynomial, as described in section 35.1.2.

See also Koev & Edelman (2006) for an "exact" calculation.

#### 35.1.3.1 Laplace approximation

The following closed-form approximation based on a Laplace approximation has been derived by Butler & Wood (2002):

$${}_2F_1(a, b; c; X) \approx \frac{c^{pc-p(p+1)/4}}{\sqrt{R_{2,1}}} \times \prod_{i=1}^p \left[ \left( \frac{y_i}{a} \right)^a \left( \frac{1-y_i}{c-a} \right)^{c-a} (1-x_i y_i)^{-b} \right], \quad (35.1.11)$$

where  $X = \text{diag}(x_1, \dots, x_p)$ ,  $S_i = x_i y_i (1-y_i) / (1-x_i y_i)$ ,  $\tau_i = x_i(b-a) - c$ ,

$$y_i = \frac{2a}{\sqrt{t_i^2 - 4ax_i(c-b) - \tau_i}}, \text{ and } R_{2,1} = \prod_{i=1}^p \prod_{j=i}^p \left[ \frac{y_i y_j}{a} + \frac{(1-y_i)(1-y_j)}{c-a} - \frac{b}{a(c-a)} S_i S_j \right].$$

### 35.1.4 Confluent Hypergeometric Function for Matrix Argument

---

Function **Hypergeometric1F1Matrix(*a* As mpNum, *b* As mpNum, *T* As mpNum) As mpNum**

**NOT YET IMPLEMENTED**

---

The function **Hypergeometric1F1Matrix** returns Kummer's confluent hypergeometric function for matrix argument.

**Parameters:**

*a*: A real number.

*b*: A real number.

*T*: A real matrix.

Kummer's confluent hypergeometric function for matrix argument  ${}_1F_1(a; b; \mathbf{T})$  is defined by the series ([Olver et al., 2010](#))

$${}_1F_1(a; b; \mathbf{T}) = \sum_{k=0}^{\infty} \frac{1}{k!} \sum_{|\kappa|=k} \frac{[a]_{\kappa}}{[b]_{\kappa}} Z_{\kappa}(\mathbf{T}) \quad (35.1.12)$$

with  $-b + \frac{1}{2}(j+1) \notin \mathbb{N}$ ,  $1 \leq j \leq m$ . Here  $Z_{\kappa}(\mathbf{T})$  is a zonal polynomial, as described in section [35.1.2](#).

See also [Koev & Edelman \(2006\)](#) for an "exact" calculation.

#### 35.1.4.1 Laplace approximation

The following closed-form approximation based on a Laplace approximation has been derived by [Butler & Wood \(2002\)](#), with  $X = \text{diag}(x_1, \dots, x_p)$ :

$${}_1F_1(a; b; X) \approx \frac{b^{pb-p(p+1)/4}}{\sqrt{R_{1,1}}} \times \prod_{i=1}^p \left[ \left( \frac{y_i}{a} \right)^a \left( \frac{1-y_i}{b-a} \right)^{b-a} e^{x_i y_i} \right], \quad (35.1.13)$$

where  $y_i = \frac{2a}{b - x_i + \sqrt{(x_i - b)^2 + 4ax_i}}$ , and  $R_{1,1} = \prod_{i=1}^p \prod_{j=i}^p \left[ \frac{y_i y_j}{a} + \frac{(1-y_i)(1-y_j)}{b-a} \right]$ .

### 35.1.5 Confluent Hypergeometric Limit Function for Matrix Argument

---

Function **Hypergeometric0F1Matrix(*n* As mpNum, *T* As mpNum) As mpNum**

---

NOT YET IMPLEMENTED

---

The function **Hypergeometric0F1Matrix** returns the confluent hypergeometric limit function for matrix argument.

**Parameters:**

*n*: A real number.

*T*: A real matrix.

This function returns the confluent hypergeometric limit function for matrix argument  ${}_0F_1(a; X)$ , defined by the series ([Butler & Wood, 2003](#); [Olver \*et al.\*, 2010](#))

$${}_0F_1\left(\frac{1}{2}n; \frac{1}{4}XX^T\right) = \sum_{k=0}^{\infty} \frac{1}{k!} \sum_{|\kappa|=k} \frac{1}{\left[\frac{1}{2}n\right]_{\kappa}} Z_{\kappa}(\mathbf{X}) \quad (35.1.14)$$

Here  $Z_{\kappa}(\mathbf{X})$  is a zonal polynomial, as described in section [35.1.2](#).

#### 35.1.5.1 Laplace approximation

The following closed-form approximation based on a Laplace approximation has been derived by [Butler & Wood \(2003\)](#), with  $X = \text{diag}(x_1, \dots, x_p)$ :

$${}_0F_1\left(\frac{1}{2}n; \frac{1}{4}XX^T\right) \approx \frac{1}{\sqrt{R_{0,1}}} \times \prod_{i=1}^p \left[ (1 - y_i)^{n/2} e^{x_i y_i} \right], \quad (35.1.15)$$

$$\text{where } y_i = \frac{2x_i/n}{\sqrt{(2x_i/n)^2 + 1 + 1}}, \text{ and } R_{0,1} = \prod_{i=1}^p \prod_{j=i}^p (1 - y_i^2 y_j^2).$$

# Chapter 36

## Examples: Moments, cumulants, and expansions

### 36.1 Moments and cumulants

The following relations hold between central moments, cumulants, and moments about the origin:  
Central moments from null moments:

$$\mu_n = \sum_{j=0}^n \binom{n}{j} (-1)^{n-j} \mu'_j (\mu_1)^{n-j} \quad (36.1.1)$$

Null moments from central moments:

$$\mu'_n = \sum_{r=0}^n \binom{n}{r} (\mu_1)^r \quad (36.1.2)$$

Cumulants from null moments:

$$\kappa_r = \mu'_r - \sum_{j=1}^{r-1} \binom{r-1}{j-1} \mu'_{r-j} \kappa_j \quad (36.1.3)$$

Cumulants from central moments:

$$\kappa_r = \mu_r - \sum_{j=2}^{r-1} \binom{r-1}{j-1} \mu_{r-j} \kappa_j, \quad r > 1, \quad \kappa_1 = \mu_1. \quad (36.1.4)$$

---

Function **CentralMomentsToRawMoments**(*Central* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function **CentralMomentsToRawMoments** returns raw moments calculated from central moments.

**Parameter:**

*Central*: A real vector.

---

Function **RawMomentsToCentralMoments**(*Raw* As mpNum) As mpNum

---

NOT YET IMPLEMENTED

---

The function `RawMomentsToCentralMoments` returns central moments calculated from raw moments.

**Parameter:**

*Raw*: A real vector.

---

Function **CentralMomentsToCumulants**(*Central* As *mpNum*) As *mpNum*

---

NOT YET IMPLEMENTED

---

The function `CentralMomentsToCumulants` returns cumulants calculated from central moments.

**Parameter:**

*Central*: A real vector.

---

Function **CumulantsToCentralMoments**(*Cumulants* As *mpNum*) As *mpNum*

---

NOT YET IMPLEMENTED

---

The function `CumulantsToCentralMoments` returns central moments calculated from cumulants.

**Parameter:**

*Cumulants*: A real vector.

---

Function **RawMomentsToCumulants**(*Raw* As *mpNum*) As *mpNum*

---

NOT YET IMPLEMENTED

---

The function `RawMomentsToCumulants` returns cumulants calculated from raw moments.

**Parameter:**

*Raw*: A real vector.

---

Function **CumulantsToRawMoments**(*Cumulants* As *mpNum*) As *mpNum*

---

NOT YET IMPLEMENTED

---

The function `CumulantsToRawMoments` returns raw moments calculated from cumulants.

**Parameter:**

*Cumulants*: A real vector.

## 36.2 The Edgeworth expansion

### 36.2.1 Continuous variates

Many distribution function can be approximated by an expansion of the normal distribution, provided that the cumulants of the distribution are known (Lee & Lin, 1992).

$$f(x) = \phi(x) + \sum_{i=1}^{\infty} \sum_{\pi(i)} \omega_{\pi(i)} Z^{(D(\pi(i)))}(x) \quad (36.2.1)$$

$$F(x) = \Phi(x) + \sum_{i=1}^{\infty} \sum_{\pi(i)} \omega_{\pi(i)} Z^{(D(\pi(i))-1)}(x) \quad (36.2.2)$$

where the summation is extended over the partitions

$$\pi(i) = (s_1^{m_1}, \dots, s_k^{m_k}) \text{ such that } s_1 > \dots > s_k \text{ and } i = \sum_{i=1}^k m_i s_i, \quad (36.2.3)$$

$$\omega_{\pi(i)} = \prod_{i=1}^k \frac{\gamma_{s_i}^{m_i}}{m_i!}, \quad \gamma_i = \frac{(-1)^i \kappa_i}{\sqrt{\kappa_2^i} (i+2)!}, \text{ and } D(\pi(i)) = \sum_{i=1}^k m_i (s_i + 2). \quad (36.2.4)$$

Using the first 4 cumulants, we have

$$\Pr \left[ \frac{u - \mu}{\sigma} \right] = \Phi(x) + \phi(x) \left[ \frac{\gamma_1}{6} (1 - x^2) + \frac{\gamma_2}{24} (3x - x^3) + \frac{\gamma_1^2}{72} (15x - 10x^3 + x^5) \right] + O(N^{-3/2}) \quad (36.2.5)$$

#### 36.2.1.1 Wilks, Hotelling's T2, Pillai's V

Example: moments of T2: Davis (1968) , equation 7.13

moments of V: Davis (1970a) Davis 1970, equation 2.11 (and section 4).

Box-omega of V: Davis (1970a), equations 5.2 to 5.4

Box-omega of T2: Davis (1970a), equations 5.6, and Davis (1970b), equations 3.10 to 3.11

### 36.2.2 Lattice (discrete) variates

Deccribe formula for Kendall's tau.

### 36.3 The Cornish-Fisher expansion

Many distribution function can be approximated by an expansion of the normal distribution, provided that the cumulants of the distribution are known. Cornish and Fisher derived an inversion formula ([Fisher & Cornish, 1960](#)), which has been generalized by [Lee & Lin \(1992\)](#). Lee's algorithm is based on the following formula for the adjustment of order  $n^{-k/2}$ :

$$\delta_k = a_k H^{k+1} + \sum_{(\kappa*)} \frac{(-1)^\pi}{j_1! \dots j_m!} (\delta_{k_1} - a_{k_1} H^{k_1+1})^{j_1} \dots (\delta_{k_m} - a_{k_m} H^{k_m+1})^{j_m} H^{\pi-1} \quad (36.3.1)$$

where  $\delta_i$  is the  $i$ th-order adjustment (of magnitude  $O(n^{i/2})$ ),  $a_i = \kappa_i / i! \sigma^i$ , where  $\kappa_i$  is the  $i$ th cumulant, and  $(\kappa*)$  is the set of all partitions of  $k$  having two or more parts. The typical term shown in the summand is due to the partition having  $j_1$  occurrences of  $k_1$  and so on;  $\pi$  is the number of parts in the partition; and the powers of  $H$  are to be multiplied out formally, with  $H^j$  replaced by the Hermite polynomial  $H_j(z)$  (with  $z$  representing the normal deviate) in the final result. A recurrence formula connects the  $\delta_k(H)$ :

$$\delta'_k = a'_k H^{k+1} + \sum_{j=1}^{k-1} \binom{k-1}{j} \delta'_{k-1}(H) (\delta'_j - a'_j H^{j+1}), \quad (36.3.2)$$

where  $\delta'_h(H) = h! \delta_h(H)$ ,  $\delta'_h = h! \delta_h$ ,  $a'_h = h! h_h$  and  $\delta_h(H)$  means an expression like the right-hand side of equation (36.3.1), formally a polynomial in  $H$ , and  $\delta_h$  is a numerical value. In this formula the adjustment of a given order is calculated recursively, the values of lower order adjustments being used in each stage of calculation.

Using the first 4 cumulants, we have

$$u = \mu + \sigma \left[ z + \frac{\gamma_1}{6}(z^2 - 1) + \frac{\gamma_2}{24}(z^3 - 3z) - \frac{\gamma_1^2}{36}(2z^3 - 5z) \right] + O(N^{-3/2}) \quad (36.3.3)$$

## 36.4 Saddlepoint approximations

Suppose the moment generating function (MGF) of the random variable  $X$  is analytic and given by  $M(t)$  for  $t$  in some open neighborhood of zero. Let  $K(t) = \log(M(t))$  be the Cumulant Generating Function(CGF) of  $X$ , and denote by  $K'(t)$ ,  $K''(t)$ ,  $K^{(3)}(t)$  and  $K^{(4)}(t)$  the first, second, third and fourth derivative of  $K(t)$ , respectively. Let  $F(x)$  and  $f(x)$  denote the CDF and pdf of  $X$ , respectively, and  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the CDF (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively.

Let  $s$  be the solution to the saddlepoint equation

$$K'(s) = x. \quad (36.4.1)$$

which in general needs to be solved numerically (sometimes closed form solutions exist), and define

$$w = w(s) = \operatorname{sgn}(s) \sqrt{2(sK'(s) - K(s))} \quad (36.4.2)$$

$$u = u(s) = s \sqrt{K''(s)} \quad (36.4.3)$$

$$r = r(s) = w(s) + \frac{1}{w(s)} \log \frac{u(s)}{w(s)} \quad (36.4.4)$$

### 36.4.1 First order approximations

Add: formulas for  $s = 0$ .

Jensen (1992) gives the following formula:

$$F(x) \approx \Phi(r) =: P1. \quad (36.4.5)$$

Lugannani & Rice (1980) give the following formula:

$$F(x) \approx \Phi(w) + \phi(w) \left( \frac{1}{w} - \frac{1}{u} \right) =: P3. \quad (36.4.6)$$

### 36.4.2 Second order approximations

Daniels (1954, 1987) gives the following formulas:

$$F(x) \approx P3 + \phi(w) \left( \frac{\kappa_4}{8u} - \frac{5}{24u} \kappa_3^2 - \frac{1}{u^3} - \frac{\kappa_3}{2u^2} + \frac{1}{w^3} \right) =: P4, \quad (36.4.7)$$

$$f(x) \approx \phi(w) \frac{s}{u} \left( 1 + \frac{\kappa_4}{8} - \frac{5}{24} \kappa_3^2 \right), \quad \text{where} \quad (36.4.8)$$

$$\kappa_3 = K^{(3)}(s)/K''(s)^{3/2} \quad \kappa_4 = K^{(4)}(s)/K''(s)^2 \quad (36.4.9)$$

## 36.5 Inverse Saddlepoint approximations

We assume that we have defined a function  $F_s(x)$  to convert from  $x$  to  $s$ , and also an inverse function  $F_x(s)$  to convert from  $s$  to  $x$  (this could be, but does not have to be, equation 36.4.1).

To obtain an approximation to the  $\alpha$ -quantile of  $F(x)$ , we start with a reasonable estimate, say,  $x_0$ , and convert this to the corresponding saddlepoint, say  $s_0$ .

Let  $s$  be a saddlepoint, and let  $w(s)$  and  $u(s)$  be functions of  $s$ , which could be (but do not have to be) defined as in equations 36.4.2 and 36.4.3

Let  $z_\alpha$  be the  $\alpha$ -quantile of the standard normal distribution. We wish to solve for  $s$  in the equation

$$h(s) = w(s) + \frac{v(s)}{w(s)} - z_\alpha = 0, \quad \text{where } v(s) = \log \frac{u(s)}{w(s)}, \quad \text{and} \quad (36.5.1)$$

$$h'(s) = w'(s) + \frac{w(s)u'(s) - u(s)w'(s)(v(s) + 1)}{u(s)w(s)^2} \quad (36.5.2)$$

Starting with  $n = 0$ , we compute successive approximations as

$$s_{n+1} = s_n - \frac{h(s_n)}{h'(s_n)} \quad (36.5.3)$$

The initial steps can be run with

$$s_{n+1} = s_n - \frac{w(s_n) - z_\alpha}{w'(s_n)} \quad (36.5.4)$$

since  $w(s)$  is the dominant term in the right hand side of equation 36.5.1.

If  $w(s)$  and  $u(s)$  are defined as in equations 36.4.2 and 36.4.3, then

$$w'(s) = \text{sgn}(s) \frac{sK''(s)}{\sqrt{2sK'(s) - 2K(s)}} = \text{sgn}(s) \frac{sK''(s)}{w(s)} \quad (36.5.5)$$

$$u'(s) = \frac{sK^{(3)}(s) + 2K''(s)}{2\sqrt{K''(s)}} \quad (36.5.6)$$

See also Wang (1995)

## 36.6 The Box-Davis expansion for a class of multivariate distributions

### 36.6.1 Definition

[Davis \(1971\)](#) considers a general class of random variables with absolutely continuous distribution functions  $F(x)$ , whose cumulant generating function may be validly represented by an asymptotic series

$$K(\theta) \sim -\frac{1}{2}f \log(1 - 2i\theta) + \sum_{r=1}^{\infty} \omega_r \left[ (1 - 2i\theta)^{-r} - 1 \right], \quad (36.6.1)$$

corresponding to a limiting chi-squared distribution with  $f$  degrees of freedom.

Examples are given in sections [36.6.4](#), [36.6.5](#) and [36.6.6](#).

A special subset of the above are random variables  $W(0 \leq W \leq 1)$  with the  $h$ th moment

$$\mathbb{E}[W^h] = K \left( \frac{\prod_{j=1}^b y_j^{y_j}}{\prod_{k=1}^a x_k^{x_k}} \right)^h \frac{\prod_{k=1}^a \Gamma[x_k(1+h) + \xi_k]}{\prod_{j=1}^b \Gamma[y_j(1+h) + \eta_j]}, \quad h = 0, 1, \dots, \quad (36.6.2)$$

where  $\Gamma(\cdot)$  denotes the Gamma function (see section [??.](#)),  $K$  is a constant such that  $e[W^0] = 1$ , i.e.

$$K = \frac{\prod_{j=1}^b \Gamma(y_j + \eta_j)}{\prod_{k=1}^a \Gamma(x_k + \xi_k)}, \quad \text{and} \quad \sum_{k=1}^a x_k = \sum_{j=1}^b y_j. \quad (36.6.3)$$

We define

$$f = -2 \left[ \sum_{k=1}^a \xi_k - \sum_{j=1}^b \eta_j - \frac{1}{2}(a-b) \right], \quad (36.6.4)$$

$$\rho = 1 - \frac{1}{f} \left[ \sum_{k=1}^a x_k^{-1} \left( \xi_k^2 - \xi_k + \frac{1}{6} \right) - \sum_{j=1}^b y_j^{-1} \left( \eta_j^2 - \eta_j + \frac{1}{6} \right) \right], \quad (36.6.5)$$

$$\omega_r = \frac{(-1)^{r+1}}{r(r+1)} \left[ \sum_{k=1}^a \frac{B_{r+1}(\beta_k + \xi_k)}{(\rho x_k)^r} - \sum_{j=1}^b \frac{B_{r+1}(\epsilon_j + \eta_j)}{(\rho y_j)^r} \right], \quad (36.6.6)$$

where  $\rho$  is chosen in such a way that  $\omega_1 = 0$ ,  $\beta_k = (1 - \rho)x_k$ ,  $\epsilon_j = (1 - \rho)y_j$ , and  $B_{r+1}(x)$  denotes the Bernoulli polynomial of degree  $r + 1$  (see section [14.2](#)).

We assume  $x_k = c_k \theta$  and  $y_j = d_j \theta$ , where  $c_k$  and  $d_j$  will be constant and  $\theta$  will vary. Then  $M = -2\rho \log(W)$  follows asymptotically (with  $\theta \rightarrow \infty$ ) a  $\chi^2$ -distribution with  $f$  degrees of freedom, and the pdf, CDF and inverse CDF of  $M$  can be developed into asymptotic expansions including terms of order  $O(\theta^{-r})$ , depending only on  $f$  and  $\omega_i (i = 1 \dots r)$ .

The first comprehensive review of this type of expansion for the CDF is due to [Box \(1949\)](#). The corresponding expansion for the inverse CDF is due to [Davis \(1971\)](#). We refer therefore to this system of expansion as the Box-Davis expansion. For additional information, see [Anderson \(2003\)](#).

Many test-criteria of classical univariate and multivariate statistics have moments of the form given in [36.6.2](#) and allow therefore evaluation of their pdf, CDF and inverse CDF by the Box-Davis expansion. Some of them are reviewed in sections [36.6.4](#) and [36.6.5](#).

For a subset of the random variables  $W$  defined in equation (36.6.2) there exists a power transformation  $U = W^{2/N}$ , such that

$$\mathbb{E}[U^h] = \prod_{j=1}^p \frac{\Gamma(c_j)\Gamma(b_j + h)}{\Gamma(b_j)\Gamma(c_j + h)}, \quad (36.6.7)$$

i.e.  $U$  is distributed as the product of  $p$  independent random variables which have a beta distribution with  $b_j$  and  $c_j - b_j$  degrees of freedom. This allows to alternatively calculate their pdf and CDF by the algorithms given in 36.7. Some examples are reviewed in section 36.6.4.

### 36.6.2 Density and CDF

---

Function **BoxDavisDist**(*M* As *mpNum*, *f* As *mpNum*, *omega* As *mpNum*[], *Output* As *String*)  
As *mpNumList*

---

**NOT YET IMPLEMENTED**

---

The function BoxDavisDist returns pdf, CDF and related information for the Box-Davis-distribution

**Parameters:**

*M*: A real number greater 0, defined as  $M = -2\rho \log(W)$

*f*: A real number greater 0, representing the degrees of freedom for the  $\chi^2$ -expansion

*omega*: An array of real numbers, representing the coefficients  $\omega_i$

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given below.

#### 36.6.2.1 CDF: General formulas

Box (1949) suggested the following approximation to a class of multivariate criteria (see also Siotani *et al.* (1985)):

$$F_{Box}(x, n; \omega_i) = F_{\chi^2}(n, x) - 2f_{\chi^2}(n, x) \sum_{i=1}^{\infty} \sum_{\pi(i)} \omega_{\pi(i)} h_{\pi(i)}(x) \quad (36.6.8)$$

where the summation is extended over the partitions

$$\pi(i) = (s_1^{m_1}, \dots, s_k^{m_k}) \text{ such that } s_1 > \dots > s_k \text{ and } i = \sum_{i=1}^k m_i s_i, \quad (36.6.9)$$

$$\text{where } \omega_{\pi(i)} = \prod_{i=1}^k \frac{\omega_{s_i}^{m_i}}{m_i!}, \quad (36.6.10)$$

and  $h_{\pi(i)}(x)$  can be computed as

$$h_i = \frac{\sum_{r=1}^i x^r}{\prod_{j=1}^r f + 2j - 2}, \quad (36.6.11)$$

$$h_{p,q} = h_{p+q} - (h_p + h_q) \quad (36.6.12)$$

$$h_{p,q,r} = h_{p+q+r} - (h_{p+q} + h_{p+r} + h_{q+r}) + (h_p + h_q + h_r) \quad (36.6.13)$$

and the extension for higher orders is obvious. For  $\omega_1 = 0$ , the expansion is given by, including terms of order  $O(\theta^{-7})$ :

$$\begin{aligned} S &= \omega_2 h_2 \\ &+ \omega_3 h_3 \\ &+ \omega_4 h_4 + \frac{1}{2}\omega_2^2(h_4 - 2h_2) \\ &+ \omega_5 h_5 + \omega_3 \omega_2(h_5 - h_3 - h_2) \\ &+ \omega_6 h_6 + \omega_4 \omega_2(h_6 - h_4 - h_2) + \frac{1}{2}\omega_3^2(h_6 - 2h_3) + \omega_2^3(h_6 - 3h_4 + 3h_2)/6 \\ &+ \omega_7 h_7 + \omega_5 \omega_2(h_7 - h_5 - h_2) + \omega_4 \omega_3(h_7 - h_4 - h_3) + \frac{1}{2}\omega_3 \omega_2^2(h_7 - 2h_5 - h_4 + h_3 + 2h_2) \end{aligned} \quad (36.6.14)$$

For  $\omega_1 \neq 0$ , the expansion is given by, including terms of order  $O(\theta^{-4})$ :

$$\begin{aligned} S &= \omega_1 h_1 \\ &+ \omega_2 h_2 + \frac{1}{2}\omega_1^2(h_2 - 2h_1) \\ &+ \omega_3 h_3 + \omega_2 \omega_1(h_3 - h_2 - h_1) + \omega_1^3(h_3 - 3h_2 + 3h_1)/6 \\ &+ \omega_4 h_4 + \omega_3 \omega_1(h_4 - h_3 - h_1) + \frac{1}{2}\omega_2 \omega_1^2(h_4 - 2h_3 + 2h_1) + \frac{1}{2}\omega_2^2(h_4 - 2h_2) \\ &+ \omega_1^4(h_4 - 4h_3 + 6h_2 - 4h_1) \end{aligned} \quad (36.6.15)$$

### 36.6.3 Quantiles

---

Function **BoxDavisDistInv**(*Prob* As mpNum, *f* As mpNum, *omega* As mpNum[], *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **BoxDavisDistInv** returns quantiles and related information for the Box-Davis-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*f*: A real number greater 0, representing the degrees of freedom for the  $\chi^2$ -expansion

*omega*: An array of real numbers, representing the coefficients  $\omega_i$

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below

#### 36.6.3.1 Davis Inversion Formula

Davis (1971) derived the following approximate inversion formula, assuming  $\omega_1 = 0$ :

The expansion 36.6.8 may be inverted to express an arbitrary  $100(1 - \alpha)\%$  point  $x_\alpha$  in terms of the corresponding chi-squared percentile  $u = \chi_{f,\alpha}^2$ .

$$F_{Box}^{-1}(\alpha, n; \omega_i) = u - 2 \sum_{i=1}^{\infty} \sum_{\pi(i)} \omega_{\pi(i)} P_{\pi(i)}(u) \quad (36.6.16)$$

where the summation and  $\omega_{\pi(i)}$  are defined as in equations 36.6.9 and 36.6.10, and the polynomials  $P_{\pi(i)}(u)$  are given by

$$P_{\pi(i)}(u) = \sum_{r=1}^{l_\pi} D_{(r)} h_\pi^{(r)}(u), \quad \text{where} \quad (36.6.17)$$

$$D_1 = 1, \quad D_r = 2 \frac{d}{du} [-(r-1)(1-(f-2)/u)], D_{(r)} = D_1 D_2 \dots D_r. \quad (36.6.18)$$

In particular,  $P_{[n]} = h_n (n = 1, 2, \dots)$ , with  $h_\pi$  defined in equations 36.6.11 to 36.6.13

Below the  $P_{\pi(i)}(u)$  for those partitions of integers 2 to 7 which do not contain unity are given:

$$P_{2,2} = -\frac{8u^4(f+3)}{(f_2 f_4)} + \frac{8u^3}{(f_2 f_3)} + \frac{6u^2}{(f f_2)} + \frac{2u}{f^2} \quad (36.6.19)$$

$$P_{3,2} = -\frac{12u^5(f+4)}{(f_2 f_5)} - \frac{2u^4(f-6)}{(f_2 f_4)} + \frac{2u^3(3f+10)}{(f_2 f_3)} + \frac{6u^2}{(f f_2)} + \frac{2u}{f^2} \quad (36.6.20)$$

$$\begin{aligned} P_{4,2} = & -16u^6(f+5)/(f_2 f_6) - 4u^5(f-4)/(f_2 f_5) + 2u^4(3f+14)/(f_2 f_4) \\ & + 2u^3(3f+10)/(f_2 f_3) + 6u^2/(f f_2) + 2u/f^2 \end{aligned} \quad (36.6.21)$$

$$\begin{aligned} P_{3,3} = & -6u^6(3f^2+30f+80)/(f_3 f_6) - 6u^5(f_2+2f-16)/(f_3 f_5) \\ & + 4u^4(f+12)/(f_2 f_4) + 4u^3(3f+8)/(f_2 f_3) + 6u^2/(f f_2) + 2u/f^2 \end{aligned} \quad (36.6.22)$$

$$\begin{aligned} P_{2,2,2} = & 32u^6(7f^2+62f+120)/(f_2^2 f_6) - 32u^5(2f^2+37f+96)/(f_2^2 f_5) \\ & + -8u^4(23f^2+124f+132)/(f_2^2 f_4) - 8u^3(f-10)/(f f_2 f_3) \\ & + 28u^2/(f^2 f_2) + 4u/f^3; \end{aligned} \quad (36.6.23)$$

$$P_{5,2} = -\frac{20u^7(f+6)}{(f_2 f_7)} - \frac{2u^6(3f-10)}{(f_2 f_6)} + \frac{2u}{f^2} + 2 \sum_{r=2}^5 \frac{u^r(3f+4r-2)}{f_2 f_r} \quad (36.6.24)$$

$$\begin{aligned} P_{4,3} = & -24u^7(f_2+12f+40)/(f_3 f_7) - 2u^6(5f_2+18f-80)/(f_3 f_6) \\ & + 2u^5(f_2+42f+176)/(f_3 f_5) + 4u^4(3f+16)/(f_2 f_4) \\ & + 4u^3(3f+8)/(f_2 f_3) + 6u^2/(f f_2) + 2u/f^2; \end{aligned} \quad (36.6.25)$$

$$\begin{aligned} P_{3,2,2} = & 192u^7(2f^3+31f^2+154f+240)/(f_2 f_3 f_7) \\ & + -16u^6(4f^3+153f^2+1106f+2160)/(f_2 f_3 f_6) - 8u^5(35f^3 \\ & + 420f^2+1540f+1632)/(f_2 f_3 f_5) - 4u^4(25f^2+80f+12)/(f_2^2 f_4) \\ & + 4u^3(7f+38)/(f f_2 f_3) + 28u^2/(f^2 f_3) + 4u/f^3; \end{aligned} \quad (36.6.26)$$

### 36.6.4 Special Cases: Products of beta variables

#### 36.6.4.1 Test of a given covariance matrix

The modified likelihood criterion for testing the hypothesis that a sample of size  $N$  is drawn from a  $p$ -variate normal population with a given matrix  $\Sigma_0$  is a power of

$$W = e^{\frac{1}{2}pn} |S\Sigma_0^{-1}|^{\frac{1}{2}n} \left( \text{tr}(S\Sigma_0^{-1})/p \right)^{\frac{1}{2}pn} \quad (36.6.27)$$

where  $n = N - 1$  and  $S$  is the sample covariance matrix. We have

$$f = \frac{1}{2}(p-1)(p+2); \quad \rho = 1 - \frac{2p^2 + p + 2}{6pn} \quad (36.6.28)$$

$$\omega_r = \frac{2(-1)^r}{r(r+1)(r+2)\rho^r} \sum_{s=1}^{r+2} \binom{r+2}{s+1} (1-\rho)^{r+1-s} \frac{\delta_s + \frac{1}{2}(s+1)B_s/p^{s-1}}{\binom{\frac{1}{2}^{s-1}}{2}}, \quad (36.6.29)$$

where  $B_s$  are the Bernoulli numbers.

#### 36.6.4.2 Test of a given mean and covariance matrix

The modified likelihood criterion for testing the hypothesis that a sample of size  $N$  is drawn from a  $p$ -variate normal population with a given matrix  $\Sigma_0$  is a power of

$$W = e^{\frac{1}{2}pn} |S\Sigma_0^{-1}|^{\frac{1}{2}n} \left( \text{tr}(S\Sigma_0^{-1})/p \right)^{\frac{1}{2}pn} \quad (36.6.30)$$

where  $n = N - 1$  and  $S$  is the sample covariance matrix. We have

$$f = \frac{1}{2}(p-1)(p+2); \quad \rho = 1 - \frac{2p^2 + p + 2}{6pn} \quad (36.6.31)$$

$$\omega_r = \frac{2(-1)^r}{r(r+1)(r+2)\rho^r} \sum_{s=1}^{r+2} \binom{r+2}{s+1} (1-\rho)^{r+1-s} \frac{\delta_s + \frac{1}{2}(s+1)B_s/p^{s-1}}{\binom{\frac{1}{2}^{s-1}}{2}}, \quad (36.6.32)$$

where  $B_s$  are the Bernoulli numbers.

#### 36.6.4.3 Wilks U (MANOVA)

$$f = pq; \quad \rho = 1 - \frac{p+q+1}{2(N-1)} \quad (36.6.33)$$

$$\omega_r = \frac{(-2)^r}{r(r+1)\mu^r} \sum_{j=0}^{q-1} B_{r+1}((\beta - j)/2) - B_{r+1}((\beta - p - j)/2), \quad \text{where} \quad (36.6.34)$$

$$\beta = \frac{p+q+1}{2}; \quad \mu = N - \frac{p+q+3}{2}. \quad (36.6.35)$$

[Wakaki \(2006\)](#) gives the  $s$ th order cumulant  $\kappa^s$  of  $-\log \Lambda$  as

$$\kappa^s = (-1)^s \left( \psi_q^{(s-1)} \left( \frac{n-p+q}{2} \right) - \psi_q^{(s-1)} \left( \frac{n+q}{2} \right) \right) \quad (36.6.36)$$

### 36.6.4.4 Test for a correlation matrix being the identity matrix

$Pr[W \leq W_0] = F_{betaprod}(W, b_i = 1 \dots p-1, c_i = 1 \dots p-1)$ , where  $b(i) = (n-i)/2$ ;  $c(i) = n/2$ ;  
 $n = N-1$ ;

Lit.: Anderson (1958), p. 269

### 36.6.4.5 Test for independence of k sets of variates

Given  $N$  observations from a  $p$  variate normal population, suppose that the variates are partitioned into  $k$  groups of sizes  $p_i (i = 1, \dots, k; \sum p_i = p)$ , and it required to test the independence of the groups

$$\omega = \frac{(-1)^{r+1}}{r(r+1)(r+2)\mu^r} \sum_{s=0}^{r+1} \binom{s+1}{r+2} 2^s \delta_s(pl) \left[ \beta^{r+1-s} - \left( \beta - \sum_{n=1}^{l-1} p_n \right)^{r+1-s} \right] \quad (36.6.37)$$

$$f = \frac{1}{2}S_2; \quad \rho = 1 - \beta/v; \quad \beta = \frac{2S_3 + 3S_2}{12f}; \quad \mu = v\rho; \quad S_i(\sum_l p_l)^i - \sum_l (p_l)^i. \quad (36.6.38)$$

Under the null hypothesis,  $U$  is distributed as  $U = \prod_{i=2}^q \prod_{j=1}^{p(i)} X_{ij}$ , where the  $X_{ij}$  are independent and  $X_{ij}$  has the density  $\beta[x], (n-pp_i+1-j)/2, p_i/2]$ .

Lit.: Anderson 1984, p.383 and 386

### 36.6.4.6 Test for equality of k independent variances (Bartlett)

$L$  = Bartlett's statistic

$p$  = number of samples

$W = L^k$

Exact distribution, algorithm for equal sample sizes

$Pr[L \leq L_0] = F_{betaprod}(L^k, b_i = 1 \dots p, c_i = 1 \dots p)$ ,

where  $b(i) = \frac{1}{2}(n-1)$ ;  $c(i) = \frac{1}{2}(n-1) + i/p$ ;

Lit.: Glaser (1980)

Exact distribution, algorithm for unequal sample sizes

$Pr[L \leq L_0] = F_{betaprod}(L^k, b_i = 1 \dots p, c_i = 1 \dots p)$ ,

where  $b(i) = \beta_i + \frac{1}{2}$ ;  $c(i) = \gamma_i + \frac{1}{2}$ ;

$N_i$  = size of  $i$ th sample

$n_i = N_i - 1$

$T = gcd(n_1, \dots, n_p)$

$d_i = n_i/T$

$d = d_1 + \dots + d_p$

$N = d - 1$

$\gamma_i = (i-1)/d$

The  $\beta_i$  are defined by the following algorithm:

$t=0$ ;

for  $i=1$  to  $k$  do begin

for  $j=1$  to  $d_i$  do begin

$t=t+1$

$\beta_j = (j-1)/d_i$

end

end

sort( $\beta$ )

Lit.: Glaser (1980), Gupta (1984)

[Nagarsenker \(1984\)](#)

Haarsey, Cyr(1982), Glaser(1980), Dyer(1980)

### 36.6.5 Other Test Criteria concerning Covariance Matrices

#### 36.6.5.1 Test of Sphericity (Mauchley)

The likelihood criterion for testing the hypothesis that a sample of size  $N$  is drawn from a  $p$ -variate normal population whose covaraince matrix is proportional to a given matrix  $\Sigma_0$  is a power of

$$W = |S\Sigma_0^{-1}|^{\frac{1}{2}n} \left( \text{tr}(S\Sigma_0^{-1})/p \right)^{\frac{1}{2}pn} \quad (36.6.39)$$

where  $n = N - 1$  and  $S$  is the sample covariance matrix. We have

$$f = \frac{1}{2}(p-1)(p+2); \quad \rho = 1 - \frac{2p^2 + p + 2}{6pn} \quad (36.6.40)$$

$$\omega_r = \frac{2(-1)^r}{r(r+1)(r+2)\rho^r} \sum_{s=1}^{r+2} \binom{r+2}{s+1} (1-\rho)^{r+1-s} \frac{\delta_s + \frac{1}{2}(s+1)B_s/p^{s-1}}{\left(\frac{1}{2}^{s-1}\right)}, \quad (36.6.41)$$

where  $B_s$  are the Bernoulli numbers.

#### 36.6.5.2 Test for equality of $k$ independent covariance matrices (Bartlett)

Let  $p$  be the number of variables,  $k$  the number of groups,  $v_i$  the sample size in groups  $i$ ,  $N = \sum_i v_i$ .

$$\omega = \frac{(-1)^r k}{r(r+1)(r+2)\mu^r} \sum_{s=1}^{r+1} \binom{s+1}{r+2} 2^s \delta_s \gamma_s \beta^{r+1-s} \quad (36.6.42)$$

$$\gamma_s = \frac{1}{k} \sum_{i=1}^k \left( \frac{v_i}{v} \right)^{s-1} - \frac{1}{k^s}; \quad \text{for equal } v_i : \gamma_s = 1 - \frac{1}{k^s} \quad (36.6.43)$$

$$f = \frac{p(p+1)(k-1)}{2}; \quad \rho = \frac{2p^2 + 3p - 1}{6(p+1)(k-1)} \left( -\frac{1}{N} + \sum_{i=1}^k \frac{1}{n_i} \right); \quad (36.6.44)$$

$$v = \frac{N}{k}; \quad \mu + \rho v = \frac{N\rho}{k}; \quad \beta = (1-\rho)v. \quad (36.6.45)$$

Korin (1969), Anderson 1984, p.420

### 36.6.6 The Lawley-Hotelling and Pillai traces

The Lawley-Hotelling generalized  $T_0^2$  and Pillai's  $V$  statistic, defined respectively by

$$T_0^2 = \text{ntr}(AB^{-1}), \quad V = \text{ntr}(A(A+B)^{-1}), \quad (36.6.46)$$

have been suggested as alternatives to Wilk's criterion for testing multivariate linear hypotheses. Here  $A$  and  $B$  are independent  $p \times p$  Wishart matrices on  $q$  and  $n$  degrees of freedom respectively. As  $n \rightarrow \infty$ , both criteria are asymptotically distributed as  $\chi^2_{pq}$ , and their cumulant generating functions have expansions as in equation (36.6.1).

Davis (1968) has found the moments for both criteria and has given recurrence relations for them, together with a formula relating the moments for  $T_0^2$  and  $V$ .

Davis (1970b) has established general formulas for the  $\omega_r$  of both criteria and has given recurrence relations for them, together with a formula relating the  $\omega_r$  for  $T_0^2$  and  $V$ .

### 36.6.6.1 Pillai's V, Central distribution

Central distribution: Coefficients for Davis expansion:

$$s = 1, \quad k = m + 1, \quad a = 2k + n_1 \quad (36.6.47)$$

$$\omega_1 = mn_1 k / (2n_2) \quad (36.6.48)$$

$$2r\omega_r = 2(r-1)\omega_r - 1 - s(1 - k/n_2)c_{1,r}, \quad r = 2, 3, \dots \quad (36.6.49)$$

$$c_{0,0} = 1; \quad c_{0,r} = c_{r,0} = 0; \quad (r = 1, 2, \dots); \quad c_{r,1} = 0 (r = 2, 3, \dots) \quad (36.6.50)$$

$$\begin{aligned} jc_{j,r} = & [(m-j+1)(n_1-j+1)]c_{j-1,r-1} + [(j(2m+n_1-2j+2) + 2(r-1))/n_2]c_{j,r-1} \quad (36.6.51) \\ & + [(j+1)/n_2 - (j+1)(m-j+1)/n_2^2]c_{j+1,r-1} - [(mn_1+2(r-2)/n_2)c_{j,r-2} \\ & + (2/n_2) \sum_{i=1}^{r-2} i\omega_i (c_{j,r-i-1} - c_{j,r-i-2}) \end{aligned}$$

### 36.6.6.2 Hotelling's T2, Central distribution

Central distribution: Coefficients for Davis expansion:

$$s = -1, \quad k = -n_1, \quad a = 2n_1 + m + 1 \quad (36.6.52)$$

and follow equations (36.6.48) to (36.6.52)

## 36.7 The Product of Independent Beta Variables

### 36.7.1 Definition

Wilks (1932) introduced the concept of random variables  $U$ , which have moments of the form

$$\mathbb{E}(U^h) = \prod_{j=1}^p \frac{\Gamma(c_j)\Gamma(b_j + h)}{\Gamma(b_j)\Gamma(c_j + h)} \quad (36.7.1)$$

i.e.  $U$  is distributed as the product of  $p$  independent random variables which have a beta distribution with  $b_j$  and  $c_j - b_j$  degrees of freedom. See section 36.6.4 for examples of tests statistics which follow this type of distribution.

### 36.7.2 Density and CDF

---

Function **BetaProductDist**(*x* As mpNum, *p* As mpNum, *a* As mpNum[], *b* As mpNum[], *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function BetaProductDist returns pdf, CDF and related information for the central BetaProduct-distribution

#### Parameters:

*x*: A real number

*p*: An integer greater 0, representing the number of variates

*a*: An array of real numbers greater 0, representing the numerator degrees of freedom

*b*: An array of real numbers greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 32.2.2.1 and 32.2.2.2.

#### 36.7.2.1 Density: Meijer G function

The Product  $X$  of  $n$  independent betas  $X_i$ , i.e.  $X_i \sim \text{beta}(\gamma_i, \delta_i)$ ,  $i = 1, \dots, n$ , has its density defined on  $(0, 1)$ , expressed as a Meijer G function as follows (Mathai *et al.*, 2010; Pham-Gia, 2008):

$$h(x) = \left( \prod_{j=1}^n \frac{\Gamma(\gamma_j + \delta_j)}{\Gamma(\gamma_j)} \right) G_{2,2}^{1,1} \left( n \left| \begin{matrix} *, n \\ 0, n \end{matrix} \right. \right) \gamma_1 + \delta_1 - 1, \dots, \gamma_n + \delta_n - 1 \gamma_1 - 1, \dots, \gamma_n - 1 x \quad (36.7.2)$$

#### 36.7.2.2 Density: Beta-Series

Tang & Gupta (1984, 1986) proposes the following approximation:

$$f_{BetaProd}(x; b_{i=1 \dots p}, c_{i=1 \dots p}) = K_p \sum_{r=0}^{\infty} \sigma_{r,p} B(b_p, f_p + r) I'(x; b_p, f_p + r), \quad \text{where} \quad (36.7.3)$$

### 36.7.2.3 CDF: Beta-Series

Tang & Gupta (1984, 1986) proposes the following approximation:

$$\Pr[W_p \leq x] = F_{BetaProd}(x; b_{i=1 \dots p}, c_{i=1 \dots p}) = K_p \sum_{r=0}^{\infty} \sigma_{r,p} B(b_p, f_p + r) I(x; b_p, f_p + r), \quad \text{where} \quad (36.7.4)$$

$$K_m = \prod_{j=1}^m \frac{\Gamma(c_j)}{\Gamma(b_j)}; \quad f_m = \sum_{j=1}^m c_j - b_j \quad (36.7.5)$$

$$\sigma_{r,k} = \frac{\Gamma(f_{k-1})}{\Gamma(f_k + r)} \sum_{s=0}^r \frac{(c_k - b_{k-1})_s \cdot \sigma_{r-s, k-1}}{s!} \quad (36.7.6)$$

with initial values  $\sigma_{0,1} = 1/\Gamma(f_1)$  and  $\sigma_{r,1} = 0$ .

### 36.7.2.4 CDF: Chi-Square Series

Tang & Gupta (1987) proposes the following approximation:

$$F_{BetaProd}(x; b_{i=1 \dots p}, c_{i=1 \dots p}) = K_p \sum_{r=0}^{\infty} l_r(a) a^{-r} f_{\chi^2}(2f_p + 2r, 2ax), \quad 0 < x < 2\pi, \quad \text{where} \quad (36.7.7)$$

$$l_r(a) = \frac{1}{r} \sum_{r=0}^r k q_k(a) l_{r-k}(a), \quad l_0(a) = 1, \quad (36.7.8)$$

$$q_k(a) = \frac{(-1)^{k+1}}{k(k+1)} \sum_{j=1}^p B_{k+1}(b_j - a) - B_{k+1}(c_j - a) \quad (36.7.9)$$

and  $a$  is a positive constant.

### 36.7.2.5 CDF: Algorithm 3

Nagarsenker & Suniaga (1983) proposes the following approximation:

$$\Pr[W_p \leq \lambda] = I(x; sm + d, \nu_1 + r; \delta) + O(sm^{-3}), \quad \text{where} \quad (36.7.10)$$

$$x = \lambda^{1/s}; \quad a = \frac{\nu_1 - \nu_2}{2\nu_1}; \quad d = \frac{1 - \nu_1}{2}; \quad \nu_r = \sum_{i=1}^p c_i^r - b_i^r \quad (36.7.11)$$

$$s^2 = \frac{-2B_2((1 + \nu_1)/2)}{\sum_{i=1}^p B_3(a + b_i) - B_3(a + c_i)} \quad (36.7.12)$$

and  $\delta^2$  is the noncentrality parameter to be used in the non-central  $\chi^2$ -expansion of the criterion.

### 36.7.3 Quantiles

---

Function **BetaProductDistInv**(*Prob* As mpNum, *p* As mpNum, *a* As mpNum[], *b* As mpNum[], *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **BetaProductDistInv** returns quantiles and related information for the the central BetaProduct-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*p*: An integer greater 0, representing the number of variates

*a*: An array of real numbers greater 0, representing the numerator degrees of freedom

*b*: An array of real numbers greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

$$\Pr[W_p \leq \lambda] = I(x; sm + d, \nu_1 + r; \delta) + O(sm^{-3}), \quad \text{where} \quad (36.7.13)$$

$$x = \lambda^{1/s}; \quad a = \frac{\nu_1 - \nu_2}{2\nu_1}; \quad d = \frac{1 - \nu_1}{2}; \quad \nu_r = \sum_{i=1}^p c_i^r - b_i^r \quad (36.7.14)$$

$$s^2 = \frac{-2B_2((1 + \nu_1)/2)}{\sum_{i=1}^p B_3(a + b_i) - B_3(a + c_i)} \quad (36.7.15)$$

From these formulas an inversion in closed form can be derived, which can be used as a starting point for a Newton iteration.

### 36.7.4 Properties

---

Function **BetaProductDistInfo**(*p* As mpNum, *a* As mpNum[], *b* As mpNum[], *Output* As String)  
As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function BetaProductDistInfo returns moments and related information for the central BetaProduct-distribution

**Parameters:**

*p*: An integer greater 0, representing the number of variates

*a*: An array of real numbers greater 0, representing the numerator degrees of freedom

*b*: An array of real numbers greater 0, representing the denominator degrees of freedom

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 32.13.4.

#### 36.7.4.1 Moments: algorithms and formulas

The raw moments are given by:

$$\mathbb{E}(W^h) = \prod_{j=1}^p \frac{\Gamma(c_j)\Gamma(b_j + h)}{\Gamma(b_j)\Gamma(c_j + h)} \quad (36.7.16)$$

### 36.7.5 Random Numbers

---

Function **BetaProductDistRandom**(*Size* As mpNum, *p* As mpNum, *a* As mpNum[], *b* As mpNum[], *Generator* As String, *Output* As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function `BetaProductDistRandom` returns random numbers following a central BetaProduct-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*p*: An integer greater 0, representing the number of variates

*a*: An array of real numbers greater 0, representing the numerator degrees of freedom

*b*: An array of real numbers greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

### 36.7.5.1 Random Numbers: algorithms and formulas

A random number  $\xi$  from a BetaProduct distribution of  $p$  variates is obtained by generating  $p$  random numbers from a Beta distribution with  $b_i$  and  $c_i$  degrees of freedom, and then assigning their product to  $\xi$ .

# Chapter 37

## Examples: Continuous Distribution Functions

### 37.1 Distribution of the Sample Correlation Coefficient

#### 37.1.1 Definition

Let  $(X, Y)$  have a joint bivariate normal distribution with means  $\mu_x, \mu_y$ , standard deviations  $\sigma_x, \sigma_y$ , respectively, and correlation  $\rho$ . If  $(X_1, Y_1), \dots, (X_N, Y_N)$  denotes a random sample of size  $N$  from  $(X, Y)$ , the sample correlation coefficient  $R$  is defined by

$$R = \frac{\sum_{i=1}^N (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^N (X_i - \bar{X})^2 \sum_{j=1}^N (Y_j - \bar{Y})^2}} \quad (37.1.1)$$

where

$$\bar{X} = \frac{\sum_{i=1}^N X_i}{N}, \quad \bar{Y} = \frac{\sum_{i=1}^N Y_i}{N}. \quad (37.1.2)$$

For given  $N \geq 3$ , the distribution of  $R$  is independent of  $\mu_x, \mu_y, \sigma_x, \sigma_y$ , but depends upon  $\rho$ , where  $-1 < \rho < 1$ . For  $-1 \leq r \leq 1$ , define  $f_R(r, N; \rho)$  to be the probability density function for  $R$ . We denote the cumulative distribution function of  $R$  by (Odeh, 1986)

$$F_R(r, N; \rho) = \Pr[R \leq r] = \int_{-1}^r f_R(x, N; \rho) dx \quad \text{and define } P_N(r, \rho) = 1 - F_R(r, N; \rho) \quad (37.1.3)$$

See also [Subrahmaniam & Subrahmaniam \(1983\)](#)

#### 37.1.2 Density and CDF

---

Function **PearsonRhoDist**(*x* As mpNum, *N* As mpNum, *rho* As mpNum, *Output* As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function **PearsonRhoDist** returns pdf, CDF and related information for the distribution of the sample correlation coefficient

**Parameters:**

*x*: A real number

*N*: A real number greater 2, representing the sample size

*rho*: A real number greater 0, representing the correlation coefficient

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 37.1.2.1 and 37.1.2.3.

### 37.1.2.1 Density: Hotelling's algorithm

We define

$$x = \rho r, \quad A = \sqrt{1 - \rho^2}, \quad B = \sqrt{1 - x^2}, \quad C = \sqrt{1 - r^2}, \quad U = \frac{\arccos(-x)}{B} \quad (37.1.4)$$

The probability density function for  $R$  is then given by (Hotelling, 1953)

$$f_R(r, N; \rho) = K_1 A^{N-1} C^{N-4} (1 - x)^{\frac{3}{2} - N} {}_2F_1 \left( \frac{1}{2}, \frac{1}{2}; N - \frac{1}{2}; \frac{1}{2} + \frac{1}{2}x \right), \quad (37.1.5)$$

$$\text{where } K_1 = \frac{(N-2)\Gamma(N-1)}{\sqrt{2\pi}\Gamma(N-\frac{1}{2})}, \quad (37.1.6)$$

$${}_2F_1 \left( \frac{1}{2}, \frac{1}{2}; N - \frac{1}{2}; \frac{1}{2} + \frac{1}{2}x \right) = \sum_{i=0}^{\infty} M_i, \quad (37.1.7)$$

$$M_0 = 1, \quad M_i = M_{i-1} \frac{a_i^2}{c_i} \frac{1+x}{2i} \quad (37.1.8)$$

$a_1 = \frac{1}{2}$ ,  $c_1 = N - \frac{1}{2}$ ,  $a_i = a_{i-1} + 1$ ,  $c_i = c_{i-1} + 1$  and  ${}_2F_1(\cdot)$  is the Gaussian hypergeometric function (see section 11.4.1).

### 37.1.2.2 Density :Closed form expressions and recursions

For  $N = 3, 4$  the probability density function can be expressed in closed form (Odeh, 1986):

$$f_R(r, 3; \rho) = \frac{A^2(1+xU)}{\pi B^2 C} \quad (37.1.9)$$

$$f_R(r, 4; \rho) = \frac{AC^3(B^2U + 3x(1+xU))}{\pi B^4} \quad (37.1.10)$$

where  $x, A, B, C$  and  $U$  are defined in (37.1.4). The probability density function  $f_N(r, \rho)$  satisfies the following recurrence formula for  $N \geq 5$  (Hotelling, 1953):

$$f_R(r, N; \rho) = \frac{2N-5}{B^2(N-3)} x A C f_R(r, N-1; \rho) + \frac{N-3}{B^2(N-4)} A^2 C^2 f_R(r, N-2; \rho) \quad (37.1.11)$$

For  $0 \leq x \leq 1$ , eqn (37.1.11) can be used to find a sequence of values for  $f_5(r; \rho), f_6(r; \rho), \dots, f_N(r; \rho)$ . However, for  $-1 < x < 0$  the recurrence formula is numerically unstable, since the two terms on the right-hand of eqn are of opposite sign. In this case  $(1+x) < 1$ , so the series given by eqn (37.1.5) will converge extremely fast and can be used.

### 37.1.2.3 CDF: Hotelling's series

Hotelling (1953) defines  $Q_N(r; \rho) = \Pr[\rho < R < r]$  and develops  $Q_N(r; \rho)$  in the following uniformly convergent series for  $-1 < \rho < r < 1$ .

$$Q_N(r, \rho) = K_1 \sum_{j=0}^{\infty} \frac{(1 \cdot 3 \cdots (2j-1))^2 S_j}{j! 2^{2j} \cdot (2N+1) \cdots (2N+2j-1)}, \quad (37.1.12)$$

where  $K_1$  and  $S_j$  are defined in equations (37.1.6) and (37.1.17), respectively. From the relationships

$$P_N(r, \rho) = 1 - F_R(r, N; \rho) = Q_N(1, \rho) - Q_N(r, \rho) \quad (37.1.13)$$

$$P_N(-1, \rho) = 1 \quad (37.1.14)$$

$$F_R(r, N; \rho) = 1 - P_N(r, \rho) \quad (37.1.15)$$

$$F_R(-r, N; -\rho) = 1 - F_R(r, N; \rho) \quad (37.1.16)$$

we can compute  $F_N(r; \rho)$  for any value of  $r$  and  $\rho$  with  $-1 \leq r \leq 1$ ,  $-1 < \rho < 1$ . Hotelling shows that the error committed by truncating the series at any point is less than  $\frac{2}{1-|\rho|}$  times the last term used. However, it is important to note that the series converges very slowly for small values of  $N$ , and in this case a large number of terms must be computed.

$$S_j = \sum_{k=0}^j \binom{j}{k} (-1)^k \frac{1}{2} (1 - \rho^2)^k 2^{j-k} N_k \quad (37.1.17)$$

$$N_k = \sum_{s=0}^{\infty} \frac{\Gamma\left(\frac{3}{2} - k\right)}{\Gamma\left(\frac{3}{2} - k - s\right) s!} \cdot I\left(\frac{1}{2}(s+1), \frac{1}{2}(n-1), \frac{(r-\rho)^2}{(1-\rho r)^2}\right), \quad (37.1.18)$$

where  $I(a, b; x)$  denotes the Incomplete Beta Function (see section 7.7.6). Hotelling shows that in the evaluation of  $N_k$  for large  $s$  the absolute value of the ratio of the term of order  $(s+1)$  to the term of order  $s$  is bounded by  $|\rho(r-\rho)/(1-\rho r)|$ , so that the series converges rapidly.

### 37.1.2.4 CDF: Guenther's series

Guenther (1977) writes  $P_N(r; \rho) = \Pr[R > 0] - \Pr[0 < R < r]$  and develops  $\Pr[0 < R < r]$  in an infinite series involving the Incomplete Beta Function denoted by  $I(a, b; x)$ . The result is

$$\Pr[R > 0] = \frac{1}{2} \left[ 1 + \operatorname{sgn}(\rho) \cdot I\left(\frac{1}{2}(N-1), \frac{1}{2}; \rho^2\right) \right] \quad (37.1.19)$$

$$\begin{aligned} \Pr[0 < R < r] &= \sum_{j=0}^{\infty} K_1(j) \cdot I\left(\frac{1}{2}(N-2), \frac{1}{2}(2j+1); r^2\right) \\ &\quad + \sum_{j=0}^{\infty} K_2(j) \cdot I\left(\frac{1}{2}(N-2), j+1; r^2\right) \end{aligned} \quad (37.1.20)$$

where  $K_1(j), K_2(j)$  are defined recursively by

$$K_1(0) = \frac{1}{2} (1 - \rho^2)^{\frac{1}{2}(N-1)}, \quad K_1(j) = \frac{2j+N-3}{2j} \rho^2 K_1(j-1)$$

$$K_2(0) = \frac{\Gamma\left(\frac{1}{2}N\right)}{\sqrt{\pi} \Gamma\left(\frac{1}{2}(N-1)\right)} \rho (1 - \rho^2)^{\frac{1}{2}(N-1)}, \quad K_2(j) = \frac{2j+N-2}{2j+1} \rho^2 K_2(j-1)$$

Eqns (37.1.19) and (37.1.20) are used together to give  $P_N(r; \rho)$ . Guenther also obtains error bounds for truncating the infinite series given by (37.1.20).

### 37.1.2.5 CDF: Closed form expressions and recursions

For  $N = 3, 4, 5, 6$  the cdf can be expressed in closed form (Odeh, 1986):

$$F_3(r; \rho) = \frac{\arccos(-r)}{\pi} - \frac{\rho CU}{\pi} \quad (37.1.21)$$

$$F_4(r; \rho) = \frac{\arccos(\rho)}{\pi} - \frac{\rho AC^2}{\pi B^2} - \frac{rA^3U}{\pi B^2} \quad (37.1.22)$$

$$F_5(r; \rho) = \frac{\arccos(-r)}{\pi} - \frac{(x^2 - 3\rho^3 + 2)rA^2C}{2\pi B^4} + \frac{(\rho^2 - 3 + 2\rho^2x^2)\rho C^3U}{2\pi B^4} \quad (37.1.23)$$

$$F_6(r; \rho) = \frac{\arccos(\rho)}{\pi} - \frac{[\rho r^2(2x^2 + 13) - 2\rho(4x^4 + 6x^2 + 5) + \rho^3(11x^2 + 4)]AC^2}{6\pi B^6} + \frac{2x^2(-2r^2 + 1)rA^5U}{6\pi B^6} \quad (37.1.24)$$

where  $x, A, B, C$  and  $U$  are defined in (37.1.4). The cumulative distribution function  $F_N(r, \rho)$  satisfies the following recurrence formula for  $N \geq 7$  (Hotelling, 1953):

$$\begin{aligned} F_N(r; \rho) &= \frac{2(N-4)\rho^2 - N + 5}{(N-3)\rho^2} F_{N-2}(r; \rho) \\ &+ \frac{(N-5)A^2}{(N-3)\rho^2} F_{N-4}(r; \rho) \\ &+ \frac{(N-4)A^2C^2 - (2N-9)B^2}{(N-4)(N-3)\rho^2 AC} \rho f_{N-1}(r; \rho) \\ &+ \frac{(N-4)^2 + (3N(N-8) + 47)\rho^2}{(N-4)^2(N-3)\rho^2} r f_{N-2}(r; \rho) \end{aligned} \quad (37.1.25)$$

For  $N$  odd, the above formula can be used repeatedly to find  $F_7, F_9, \dots, F_N$  starting with values of  $F_3$  and  $F_5$ , for which exact expressions are given by eqns (37.1.21) and (37.1.23) respectively. For  $N$  even, the formula can be used repeatedly to find  $F_8, F_{10}, \dots, F_N$  starting with values of  $F_4$  and  $F_6$ , for which exact expressions are given by equations (37.1.22) and (37.1.24) respectively. However, the formula can only be applied if  $\rho^2 \geq \frac{N-5}{2(N-4)}$ , since if  $\rho^2$  is less than this bound, the first two terms on the right hand side of eqn (37.1.25) are of opposite sign and the formula is numerically unstable.

### 37.1.2.6 CDF: Fisher's Approximation

The widely used Fisher  $z$ -transformation leads to the approximation

$$F_N(r, \rho) \approx \Phi\left(\sqrt{N-3}(Z(r) - Z(\rho))\right), \quad \text{where} \quad (37.1.26)$$

$Z(\cdot)$  is defined in equation (37.1.46), and  $\Phi(\cdot)$  denotes the CDF of the normal distribution (see section 32.11.2.2). This approximation can be improved by using the first 4 cumulants of  $Z(R)$ , resulting in

$$F_N(r, \rho) \approx \Phi(x), \quad \text{where} \quad (37.1.27)$$

$$u = \frac{Z(r) - \kappa_1}{\sqrt{\kappa_2}}, \quad x = u + (u^2 - 1)\frac{\gamma_1}{6} + (u^3 - 3u)\frac{\gamma_2}{24} + (4u^3 - 7u)\frac{\gamma_1^2}{36} \quad (37.1.28)$$

and  $\kappa_1, \kappa_2, \gamma_1$  and  $\gamma_2$  are defined in equations (37.1.48) to (37.1.52).

### 37.1.2.7 CDF: Winterbottom's Approximation

The CDF approximation given by (Winterbottom, 1980) is:

$$F_N(r, \rho) \approx \Phi(\sqrt{m}Y), \quad \text{where } m = N - 1, \quad (37.1.29)$$

$$Y = -\frac{r}{2m} - \frac{3r + r^3}{12m^2} + \left(1 - \frac{1 + r^2}{4m} + \frac{3 - 11r^4}{96m^2}\right)w + \frac{3r - 4r^3}{24m}w^2 - \left(\frac{1}{12} - \frac{2 + 7r^2 - 6r^4}{48m}\right)w^3 + \frac{3}{160}w^5, \quad (37.1.30)$$

$w = Z(r) - Z(\rho)$  and  $Z(\cdot)$  is defined in equation (37.1.46).

### 37.1.3 Quantiles

---

Function **PearsonRhoDistInv**(*Prob* As *mpNum*, *N* As *mpNum*, *rho* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **PearsonRhoDistInv** returns quantiles and related information for the distribution of the sample correlation coefficient

**Parameters:**

*Prob*: A real number between 0 and 1.

*N*: A real number greater 2, representing the sample size

*rho*: A real number greater 0, representing the correlation coefficient

*Output*: A string describing the output choices

For a given value of  $\alpha$ ,  $N$  and  $\rho$  a  $100\alpha$  percentage point of  $R$  is the unique value of  $R$ , say  $r_\alpha$ , which satisfies  $F_N(r_\alpha; \rho) = \alpha$

Let  $u_\alpha = \Phi^{-1}(\alpha)$  be the lower  $100\alpha$  percentage point of the standard normal distribution (see section 32.11.3). Then an approximation to the  $100\alpha$  percentage point  $r_\alpha$  is obtained by

$$r_\alpha \approx Z^{-1} \left( Z(\rho) + \frac{u_\alpha}{\sqrt{N-3}} \right). \quad (37.1.31)$$

$Z(\cdot)$  and  $Z^{-1}(\cdot)$  are defined in equations (37.1.46) and (37.1.47). This approximation can be improved by using the first 4 cumulants of  $Z(R)$ , resulting in

$$r_\alpha \approx Z^{-1}(\kappa_1 + x\sqrt{\kappa_2}) \quad \text{where} \quad (37.1.32)$$

$$x = u_\alpha + (u_\alpha^2 - 1)\frac{\gamma_1}{6} + (u_\alpha^3 - 3u_\alpha)\frac{\gamma_2}{24} + (2u_\alpha^3 - 5u_\alpha)\frac{\gamma_1^2}{36} \quad (37.1.33)$$

and  $\kappa_1$ ,  $\kappa_2$ ,  $\gamma_1$  and  $\gamma_2$  are defined in equations (37.1.48) to (37.1.52).

#### 37.1.3.1 Winterbottom's Quantile Approximation

An approximation to the  $100\alpha$  percentage point  $r_\alpha$  is obtained by (Winterbottom, 1980)

$$r_\alpha \approx Z^{-1}(y), \quad \text{where} \quad (37.1.34)$$

$$y = Z(\rho) + \frac{u_\alpha}{\sqrt{m}} + \frac{\rho}{2m} + \frac{u_\alpha^3 + 3(3 - \rho^2)u_\alpha}{12\sqrt{m^3}} + \frac{4\rho^3u_\alpha^2 + 15\rho - \rho^3}{24m^2} \\ + \frac{u_\alpha^5 + (-60\rho^4 + 30\rho^2 + 80)u_\alpha^3 + (45\rho^4 - 21\rho^2 + 375)u_\alpha}{480\sqrt{m^5}} \quad (37.1.35)$$

$Z(\cdot)$  and  $Z^{-1}(\cdot)$  are defined in equations (37.1.46) and (37.1.47).

### 37.1.4 Properties

---

Function **PearsonRhoDistInfo**(*N* As mpNum, *rho* As mpNum, *Output* As String) As mpNumList  
 NOT YET IMPLEMENTED

---

The function **PearsonRhoDistInfo** returns moments and related information for the distribution of the sample correlation coefficient

**Parameters:**

*N*: A real number greater 2, representing the sample size

*rho*: A real number greater 0, representing the correlation coefficient

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.1.4.1 The moments of $R$

The moments of  $R$  are given by [Anderson \(2003\)](#), p. 166:

$$\mathbb{E}(R^{2h+1}) = \frac{(1 - \rho^2)^{n/2}}{\sqrt{\pi}\Gamma(n/2)} \sum_{i=0}^{\infty} \frac{(2\rho)^{2i+1}}{(2i+1)!} \frac{\Gamma^2[(n+1)/2 + i]\Gamma(h+i+3/2)}{\Gamma(n/2+h+i+1)} \quad (37.1.36)$$

$$\mathbb{E}(R^{2h}) = \frac{(1 - \rho^2)^{n/2}}{\sqrt{\pi}\Gamma(n/2)} \sum_{i=0}^{\infty} \frac{(2\rho)^{2i}}{(2i)!} \frac{\Gamma^2[(n+1)/2 + i]\Gamma(h+i+1/2)}{\Gamma(n/2+h+i)} \quad (37.1.37)$$

The first 4 moments of the distribution of  $R$  can also be expressed in terms of hypergeometric functions (see [Johnson \*et al.\* \(1995.\)](#) p.553):

$$\mu'_1 = c_n b_1 \rho \quad (37.1.38)$$

$$\mu'_2 = 1 - \frac{b_2(n-2)(1-\rho^2)}{n-1} \quad (37.1.39)$$

$$\mu'_3 = c_n(b_3 - b_1) \left( b_1 \rho - \frac{(n-1)(n-2)}{\rho} \right) \quad (37.1.40)$$

$$\mu'_4 = 1 + \frac{b_2(n-2)(n-4)(1-\rho^2)}{2(n-1)} - \frac{(b_2-1)n(n-2)(n-\rho^2)}{4\rho^2} \quad (37.1.41)$$

with

$$c_n = \frac{2}{n-1} \left[ \frac{\Gamma(n/2)}{\Gamma((n-1)/2)} \right]^2 \quad (37.1.42)$$

$$b_1 = {}_2F_1\left(\frac{1}{2}, \frac{1}{2}; \frac{1}{2}(n+1); \rho^2\right) \quad (37.1.43)$$

$$b_2 = {}_2F_1\left(1, 1; \frac{1}{2}(n+1); \rho^2\right) \quad (37.1.44)$$

$$b_3 = {}_2F_1\left(\frac{1}{2}, \frac{1}{2}; \frac{1}{2}(n-1); \rho^2\right) \quad (37.1.45)$$

where  $\Gamma(\cdot)$  is the Gamma function (see section ??), and  ${}_2F_1(\cdot)$  is the Gaussian hypergeometric function (see section 11.4.1).

### 37.1.4.2 Fisher's z-transform

The Fisher  $z$ -transform is defined by

$$Z(a) = \frac{1}{2} \log\left(\frac{1+a}{1-a}\right) = \operatorname{atanh}(a) \quad (37.1.46)$$

The inverse Fisher  $z$ -transform is defined by

$$Z^{-1}(a) = \frac{e^{2a} - 1}{e^{2a} + 1} = \tanh(a) \quad (37.1.47)$$

### 37.1.4.3 The cumulants of $Z(R)$

Let  $m = N - 1$ . Then the first 4 cumulants of  $Z(R)$  (with  $Z(\cdot)$  as defined in equation 37.1.46) are given by (Hotelling, 1953)

$$\kappa_1 = \frac{1}{2} \log\left(\frac{1+\rho}{1-\rho}\right) + \frac{\rho}{2m} + \frac{5+\rho^2}{4m^2} + \frac{11+2\rho^2+3\rho^4}{8m^3} + O(m^{-4}) \quad (37.1.48)$$

$$\kappa_2 = \frac{1}{m} + \frac{4-\rho^2}{2m^2} + \frac{22-6\rho^2-3\rho^4}{6m^3} + O(m^{-4}) \quad (37.1.49)$$

$$\kappa_3 = \frac{\rho^3}{m^3} + O(m^{-4}) \quad (37.1.50)$$

$$\kappa_4 = \frac{2}{m^3} + O(m^{-4}) \quad (37.1.51)$$

$$\gamma_1 = \frac{\kappa_3}{\sqrt{\kappa_2 \kappa_2}}, \quad \gamma_2 = \frac{\kappa_4}{\kappa_2^2} \quad (37.1.52)$$

## 37.1.5 Random Numbers

---

Function **PearsonRhoDistRan**(**Size** As mpNum, **N** As mpNum, **rho** As mpNum, **Generator** As String, **Output** As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function PearsonRhoDistRan returns random numbers following the distribution of the sample correlation coefficient

#### Parameters:

**Size**: A positive integer up to  $10^7$

**N**: A real number greater 2, representing the sample size

**rho**: A real number greater 0, representing the correlation coefficient

**Generator**: A string describing the random generator

**Output**: A string describing the output choices

See section 32.1.3.6 for the options for **Size**, **Generator** and **Output**. Algorithms and formulas are given below:

The correlation coefficient  $r$  in samples of size  $N > 2$  from a non-singular bivariate normal population with correlation coefficient  $\rho$  can be represented in the form

$$\tilde{r} = \frac{z + \tilde{\rho}\chi_{N-1}}{\chi_{N-2}} \quad (37.1.53)$$

where  $\tilde{r} = r/\sqrt{1 - r^2}$ ,  $\tilde{\rho} = \rho/\sqrt{1 - \rho^2}$ ,  $z$  is a standardized normal variate and  $z$ ,  $\chi_{N-1}$ , and  $\chi_{N-2}$  are independent (Ruben, 1966). This can be used to develop an approximation where

$$\frac{\sqrt{(2N-5)/2}\tilde{r} - \sqrt{(2N-3)/2}\tilde{\rho}}{\sqrt{1 + \frac{1}{2}(\tilde{r}^2 + \tilde{\rho}^2)}} \quad (37.1.54)$$

is distributed as a standard normal variate. See also [Akahira & Torigoe \(1998\)](#).

### 37.1.6 Confidence limits for rho

---

Function **PearsonRhoDistNoncentrality**(*alpha* As *mpNum*, *rho* As *mpNum*, *N* As *mpNum*, *Output* As *String*) As *mpNumList*

---

**NOT YET IMPLEMENTED**

---

The function **PearsonRhoDistNoncentrality** returns confidence limits for the noncentrality parameter *rhi* and related information for the distribution of the sample correlation coefficient.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*rho*: A real number between -1 and 1, representing the noncentrality parameter.

*N*: A real number greater 2, representing the sample size

*Output*: A string describing the output choices

See section [32.1.3.5](#) for the options for *alpha*, *Noncentrality* and *Output*. Algorithms and formulas are given below.

For a given value of  $\alpha_1 < \frac{1}{2}$ ,  $N$ , and an observed value of  $R$ , say  $r_0$ , a lower  $100(1-\alpha_1)\%$  confidence limit on  $\rho$  is the unique value of  $\rho$ , say  $\rho_L$ , which satisfies  $F_N(r_0; \rho) = 1 - \alpha_1$ . A one-sided lower  $100(1 - \alpha_1)\%$  confidence interval on  $\rho$  is given by  $\rho_L \leq \rho \leq 1$ .

For  $\alpha_1 < \frac{1}{2}$ , an upper  $100(1 - \alpha_2)\%$  confidence limit on  $\rho$  is the unique value of  $\rho$ , say  $\rho_U$ , which satisfies  $F_N(r_0; \rho) = 1 - \alpha_2$ . A one-sided upper  $100(1 - \alpha_2)\%$  confidence interval on  $\rho$  is given by  $-1 \leq \rho \leq \rho_U$ .

For  $\alpha_1 = 1 - \alpha_2 = \alpha_0$ , an equal-tailed two-sided  $100(1 - 2\alpha_0)\%$  confidence interval on  $\rho$  is given by  $\rho_L \leq \rho \leq \rho_U$ .

#### 37.1.6.1 Fisher's Approximation

To obtain a confidence interval for  $\rho$ , we first compute a confidence interval for  $z(r)$ . The inverse Fisher transformation bring the interval back to the correlation scale.

$$\rho_L \approx Z^{-1} \left( Z(r) - \frac{u_\alpha}{\sqrt{N-3}} \right). \quad (37.1.55)$$

For example, suppose we observe  $r = 0.3$  with a sample size of  $n=50$ , and we wish to obtain a 95% confidence interval for  $\rho$ . The transformed value is  $\text{arctanh}(0.3) = 0.30952$ , so the confidence interval on the transformed scale is  $0.30952 - 1.96/\sqrt{47}$ , or  $(0.023624, 0.595415)$ . Converting back to the correlation scale yields  $(0.024, 0.534)$ .

### 37.1.6.2 Winterbottom's Approximation

Let  $u_\alpha = \Phi^{-1}(\alpha)$  be the lower  $100\alpha$  percentage point of the standard normal distribution (see section 32.11.3) and let  $m = N - 1$ . Then an asymptotic approximation for a lower  $100(1 - \alpha)$  confidence limit on  $\rho$  is given by (Winterbottom, 1980)

$$\rho_L \approx Z^{-1}(y), \quad \text{where} \quad (37.1.56)$$

$$y = Z(r) + \frac{u_\alpha}{\sqrt{m}} - \frac{r}{2m} + \frac{u_\alpha^3 + 3(1 + r^2)u_\alpha}{12\sqrt{m^3}} - \frac{4r^3u_\alpha^2 + 5r^3 + 9r}{24m^2} + \frac{u_\alpha^5 + (60r^4 - 30r^2 + 20)u_\alpha^3 + (165r^4 + 30r^2 + 15)u_\alpha}{480\sqrt{m^5}} \quad (37.1.57)$$

$Z(\cdot)$  and  $Z^{-1}(\cdot)$  are defined in equations (37.1.46) and (37.1.47).

### 37.1.7 Sample Size Function

---

Function **PearsonRhoDistSampleSize**(*alpha* As mpNum, *beta* As mpNum, *ModifiedNoncentrality* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function PearsonRhoDistSampleSize returns sample size estimates and related information for the distribution of the sample correlation coefficient.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*beta*: A real number between 0 and 1, specifies the Type II error (or 1 - Power).

*ModifiedNoncentrality*: A real number greater 0, representing the modified noncentrality parameter.

*Output*: A string describing the output choices

See section 32.1.3.4 for the options for *alpha*, *beta*, *ModifiedNoncentrality* and *Output*. Algorithms and formulas are given below.

We denote by  $N_{Rho}(\alpha, \beta, \tilde{\rho})$  the sample size function of the noncentral  $t$ -distribution for a given confidence level  $\alpha$ , power  $\beta$  and modified noncentrality parameter  $\tilde{\rho}$ . This function determines the minimal sample size  $N$  for given noncentrality parameter, Type I error  $1 - \alpha$ , and Type II error  $1 - \beta$ , where  $\delta = \sqrt{N}\tilde{\rho}$

## 37.2 Distribution of the Sample Multiple Correlation Coefficient

### 37.2.1 Definition

Let  $(X_1, \dots, X_N)$  be a sample of independent vector observations from a  $p$ -variate normal population with mean  $\mu$  and covariance matrix  $\Sigma$ . Define

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i \quad \text{and} \quad \mathbf{A} = \sum_{i=1}^N (X_i - \bar{X})(X_i - \bar{X})'. \quad (37.2.1)$$

Partition  $\mathbf{A}$  as

$$\mathbf{A} = \begin{pmatrix} a_{11} & \mathbf{a}_{12} \\ \mathbf{a}_{21} & \mathbf{A}_{22} \end{pmatrix} \quad (37.2.2)$$

so that  $a_{11}$  is  $1 \times 1$ ,  $\mathbf{a}_{12}$  is  $1 \times (p-1)$ , and  $\mathbf{A}_{22}$  is a  $(p-1) \times (p-1)$  matrix. In terms of these submatrices, the squared sample multiple correlation coefficient is given by

$$R^2 = \frac{\mathbf{a}_{12} \mathbf{A}_{22}^{-1} \mathbf{a}_{21}}{a_{11}} \quad (37.2.3)$$

The sample multiple correlation coefficient is the positive square root of  $R^2$ . The squared population multiple correlation coefficient,  $\rho^2$ , is defined similarly in terms of the submatrices of  $\Sigma$ .

For given  $N - p \geq 1$ , the distribution of  $R^2$  is independent of  $\mu$  and  $\Sigma$ , but depends upon  $\rho^2$ , where  $0 \leq \rho^2 < 1$ . For  $0 \leq x \leq 1$ , define  $f_{R^2}(x; p, N, \rho^2)$  to be the probability density function for  $R^2$ . We denote the cumulative distribution function of  $R^2$  by

$$F_{R^2}(x; p, N, \rho^2) = \Pr[R^2 \leq x] = \int_0^x f_{R^2}(t; p, N, \rho^2) dt. \quad (37.2.4)$$

Note: The univariate version of the noncentral distribution of Wilks  $\Lambda$ : Canonical Correlation (see section 37.13.1.2) is equivalent to  $W = 1 - R^2$ .

### 37.2.2 Density and CDF

---

Function **Rho2Dist**(*x* As mpNum, *p* As mpNum, *N* As mpNum, *Rho2* As mpNum, *Output* As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function Rho2Dist returns pdf, CDF and related information for the distribution of the squared population multiple correlation coefficient

#### Parameters:

*x*: A real number

*p*: An integer greater 2, representing the number of variates

*N*: A real number greater *p*, representing the sample size

*Rho2*: A real number greater or equal 0, representing the squared sample multiple correlation coefficient

*Output*: A string describing the output choices

### 37.2.2.1 Density: Infinite series

The density function of the multiple sample correlation coefficient is given by (Benton & Krishnamoorthy, 2003; Ding, 1996)

$$f_{R^2}(x; p, N, \rho^2) = \sum_{i=0}^{\infty} f_{\text{NegBin}}\left((N-1)/2, i; 1-\rho^2\right) \times f_{\text{Beta}}\left(x; \frac{1}{2}(p-1) + i, \frac{1}{2}(N-p)\right) \quad (37.2.5)$$

where  $f_{\text{NegBin}}(\cdot)$  denotes the pmf of the negative binomial distribution (see section 32.10.1.1) and  $f_{\text{Beta}}(\cdot)$  denotes the pdf of the Beta distribution (see section 32.2.2.1)

### 37.2.2.2 Density: Representation in terms of hypergeometric functions

The density function of the multiple sample correlation coefficient is given by (Gurland, 1968; Lee, 1971c)

$$f_{R^2}(x; p, n, \rho^2) = \frac{(1-\rho^2)^{n/2} (x)^{(n_1-2)/2} (1-x)^{(n_2-2)/2} {}_2F_1\left(\frac{1}{2}n, \frac{1}{2}n, \frac{1}{2}n_1; \rho^2 x\right)}{B\left(\frac{1}{2}n_1, \frac{1}{2}n_2\right)} \quad (37.2.6)$$

$$f_{R^2}(x; p, N, \rho^2) = f_{\text{Beta}}\left(x; \frac{1}{2}(p-1), \frac{1}{2}(N-p)\right) (1-\rho^2)^{n/2} \times {}_2F_1\left(\frac{1}{2}N, \frac{1}{2}N, \frac{1}{2}p; \rho^2 x\right) \quad (37.2.7)$$

where  $f_{\text{Beta}}(\cdot)$  denotes the pdf of the Beta distribution (see section 32.2.2.1) and  ${}_2F_1(\cdot)$  is the Gaussian hypergeometric function (see section 11.4.1).

### 37.2.2.3 Density: Closed form expressions for odd $p$

For the particular cases of  $p = 3$  and  $p = 5$ , we have for the pdf (Lee, 1971c):

$$f_{R^2}(n, 3; \rho^2) = \frac{(n-3)\sqrt{1-\rho^2} [f_R(n-1, R; \rho) - f_R(n-1, -R; \rho)]}{2(n-2)\rho\sqrt{1-R^2}} \quad (37.2.8)$$

$$f_{R^2}(n, 5; \rho^2) = \frac{(n-5)(1-\rho^2)R [f_R(n-2, R; \rho) + f_R(n-2, -R; \rho)]}{2(n-2)\rho^2(1-R^2)} - \frac{2(1-\rho^2)(1-R^2)f_{R^2}(n-2, 3; \rho^2)}{2(n-2)\rho^2(1-R^2)} \quad (37.2.9)$$

where  $f_R(\cdot)$  denotes the pdf of the distribution of the sample correlation coefficient (see section 37.1.2.1). Starting with these formulas, the pdf can be calculated for all odd  $p$ , using the recurrence relation given in equation (37.2.17).

### 37.2.2.4 CDF: Infinite series

The CDF of the multiple sample correlation coefficient is given by (Benton & Krishnamoorthy, 2003; Ding, 1996)

$$F_{R^2}(x; p, N, \rho^2) = \sum_{i=0}^{\infty} f_{\text{NegBin}}\left((N-1)/2, i; 1-\rho^2\right) \times F_{\text{Beta}}\left(x; \frac{1}{2}(p-1) + i, \frac{1}{2}(N-p)\right) \quad (37.2.10)$$

where  $f_{\text{NegBin}}(\cdot)$  denotes the pmf of the negative binomial distribution and  $F_{\text{Beta}}(\cdot)$  denotes the CDF of the Beta distribution (see section 32.2.2.2) where  $f_{\text{NegBin}}(\cdot)$  denotes the pmf of the negative binomial distribution (see section 32.10.1.1) and  $F_{\text{Beta}}(\cdot)$  denotes the CDF of the Beta distribution (see section 32.2.2.2)

### 37.2.2.5 CDF: Series of Gurland

Let  $R^2$  be the sample multiple correlation coefficient between  $X$  (consisting of  $p - 1$  variates) and  $Y$ , based on  $N$  observations. If both  $X$  and  $Y$  are random variates, then distribution of  $F$  is given by (Gurland & Milton, 1970; Lee, 1972b),

$$\Pr[F \leq x] = F_{R^2}(x; a, b, \rho^2) = \frac{b^m}{a^{n/2}} \sum_{j=0}^{\infty} c_j I(m+j, k, y) \quad \text{where} \quad (37.2.11)$$

$$\begin{aligned} z &= F(p-1), \quad y = \frac{z}{z + (N-p)}, \quad a = \frac{1}{1 - \rho^2}, \quad n = (N-p)/2, \quad m = (p-1)/2 \\ c_0 &= 1, \quad c_j = \frac{c_{j-1}(-n/2 - j + 1)\rho^2}{j} \end{aligned}$$

### 37.2.2.6 CDF: Closed form expressions for odd $p$

For the particular cases of  $p = 3$  and  $p = 5$ , we have for the CDF (Lee, 1971c):

$$F_{R^2}(n, 3; \rho^2) = F_{R^2}(n, 1; \rho^2) - \frac{\sqrt{(1 - \rho^2)(1 - R^2)} [f_R(n-1, R; \rho) - f_R(n-1, -R; \rho)]}{(n-2)\rho} \quad (37.2.12)$$

$$\begin{aligned} F_{R^2}(n, 5; \rho^2) &= F_{R^2}(n, 3; \rho^2) - \frac{(1 - \rho^2)R [f_R(n-2, R; \rho) - f_R(n-2, -R; \rho)]}{(n-2)\rho} \\ &\quad - \frac{(1 - \rho^2) [F_{R^2}(n-2, 3; \rho^2) - F_{R^2}(n-2, 1; \rho^2)]}{(n-2)\rho} \end{aligned} \quad (37.2.13)$$

where  $F_R(\cdot)$  denotes the CDF of the sample correlation coefficient (see section 37.1.2.3). Starting with these formulas, the CDF can be calculated for all odd  $p$ , using the recurrence relation given in equation (37.2.18).

### 37.2.2.7 CDF: Approximation by non-central F

Lee (1971c) suggests the following approximation, based on the noncentral  $F$  distribution:

$$F_{R^2}(x; n_1, n_2, \rho^2) \approx F_{F'}(y; \nu, n_2, \lambda), \quad \text{where} \quad (37.2.14)$$

$$\begin{aligned} \gamma &= 1/(1 - \rho^2), \\ A_j &= (n_1 + n_2)(\gamma^j - 1) + n_1, \quad j = 1, 2, 3, \\ G &= (A_2 - \sqrt{A_2^2 - A_1 A_3})/A_1, \\ \lambda &= \rho^2 \gamma \sqrt{\gamma(n_1 + n_2)n_2}/G^2, \\ \nu &= (A_2/G^2) - 2\lambda, \\ y &= x/(1 - x) \times n_2/(\nu \cdot G), \end{aligned}$$

and  $F_{F'}(\cdot; \nu, n_2, \lambda)$  denotes the CDF of the noncentral  $F$  distribution with  $\nu$  and  $n_2$  degrees of freedom and noncentrality parameter  $\lambda$ . For performance reasons, the noncentral  $F$  distribution is evaluated using the (fast and highly accurate) saddlepoint approximation given in section 37.10.2.7, rather than any of the "exact" algorithms.

### 37.2.2.8 CDF: 2-moment approximation by central F

$$F_{F'}(x; n_1, n_2, \rho^2) \approx F_F(x/c; m_1, n_2), \quad (37.2.15)$$

where  $c = A_1/n_1$ ,  $m_1 = A_1^2/A_2$ ,  $A_1 = (n_1 + n_2)(\gamma - 1) + n_1$ ,  $A_2 = (n_1 + n_2)(\gamma^2 - 1) + n_1$ ,  $\gamma = 1/(1 - \rho^2)$ , and  $F_F(\cdot; m_1, n_2)$  denotes the CDF of a central  $F$  distribution with  $m_1$  and  $n_2$  degrees of freedom (see 32.6.2.2).

### 37.2.3 Quantiles

---

Function **Rho2DistInv**(*Prob* As mpNum, *p* As mpNum, *N* As mpNum, *Rho2* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **Rho2DistInv** returns quantiles and related information for the distribution of the squared population multiple correlation coefficient

**Parameters:**

*Prob*: A real number between 0 and 1.

*p*: An integer greater 2, representing the number of variates

*N*: A real number greater *p*, representing the sample size

*Rho2*: A real number greater or equal 0, representing the squared sample multiple correlation coefficient

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

An approximation to  $Rho_{\alpha, n_1, n_2, \lambda}^2$ , the  $\alpha$ -quantile of the distribution of the squared population multiple correlation coefficient with *p* variates, sample size *N* the distribution of the squared population multiple correlation coefficient and noncentrality parameter  $\rho^2$ , is obtained as

$$F_{\alpha, n_1, n_2, \lambda} \approx c \cdot F_{\alpha, m_1, n_2}, \quad (37.2.16)$$

where  $c = A_1/n_1$ ,  $m_1 = A_1^2/A_2$ ,  $A_1 = (n_1 + n_2)(\gamma - 1)$ ,  $A_2 = (n_1 + n_2)(\gamma^2 - 1)$ ,  $\gamma = 1/(1 - \rho^2)$ , and  $F_{\alpha, m_1, n_2}$  denotes the  $\alpha$ -quantile of a central  $F$ -distribution with  $m_1$  and  $n_2$  degrees of freedom (see section ??).

### 37.2.4 Properties

---

Function **Rho2DistInfo**(*p* As mpNum, *N* As mpNum, *Rho2* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **Rho2DistInfo** returns moments and related information for the distribution of the squared population multiple correlation coefficient

**Parameters:**

*p*: An integer greater 2, representing the number of variates

*N*: A real number greater *p*, representing the sample size

*Rho2*: A real number greater or equal 0, representing the squared sample multiple correlation coefficient

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

### 37.2.4.1 Recurrence relations

Lee (1971c) gives the following recurrence relations for the pdf and CDF, valid for all integer values of  $p \geq 6$ :

$$(n-2)\rho^2(1-R^2)f_{R^2}(n,p;\rho^2) = (n-p)R^2(1-\rho^2)f_{R^2}(n-2,p-4;\rho^2) - (p-4)(1-\rho^2)(1-R^2)f_{R^2}(n-2,p-2;\rho^2) \quad (37.2.17)$$

$$(n-2)\rho^2(1-R^2)F_{R^2}(n,p;\rho^2) = (n-p)\rho^2F_{R^2}(n,p-2;\rho^2) - (p-4)(1-\rho^2)[F_{R^2}(n-2,p-2;\rho^2) - F_{R^2}(n-2,p-4;\rho^2)] - 2R^2(1-\rho^2)f_{R^2}(n-2,p-4;\rho^2) \quad (37.2.18)$$

### 37.2.4.2 The moments of $R^2$

Muirhead (1982), p. 178, gives the following formula (see also Johnson *et al.* (1995.), p. 621)

$$\mathbb{E}[(1-R^2)^h] = \frac{\left[\frac{1}{2}(n-m+1)\right]_h}{\left(\frac{1}{2}n\right)_h} (1-\bar{R}^2)^h \times {}_2F_1(h, h, \frac{1}{2}n+h; \bar{R}^2). \quad (37.2.19)$$

where  $(a)_k$  is the Pochammer symbol (see section 7.3) and  ${}_2F_1(\cdot)$  is the Gaussian hypergeometric function (see section 11.4.1).

The lower order moments are given by

$$\mu'_1 = 1 - \frac{(n-m+1)}{n} (1-\bar{R}^2)^h \times {}_2F_1(1, 1, \frac{1}{2}n+1; \bar{R}^2). \quad (37.2.20)$$

$$\mathbb{E}[(1-R^2)^2] = \frac{\left[\frac{1}{2}(n-m+1)\right]_2}{\left(\frac{1}{2}n\right)_2} (1-\bar{R}^2)^2 \times {}_2F_1(2, 2, \frac{1}{2}n+2; \bar{R}^2). \quad (37.2.21)$$

## 37.2.5 Random Numbers

---

Function **Rho2DistRan**(*Size* As *mpNum*, *p* As *mpNum*, *N* As *mpNum*, *Rho2* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **Rho2DistRan** returns random numbers following the distribution of the squared population multiple correlation coefficient

### Parameters:

*Size*: A positive integer up to  $10^7$

*p*: An integer greater 2, representing the number of variates

*N*: A real number greater *p*, representing the sample size

*Rho2*: A real number greater or equal 0, representing the squared sample multiple correlation coefficient

*Generator*: A string describing the random generator

*Output:* A string describing the output choices

A notable result concerning the distribution of  $R^2$  is that  $\tilde{R}^2 = R^2/(1-R^2)$  has the representation

$$\tilde{R}^2 = \frac{(\tilde{\rho}\chi_n + z)^2 + \chi_{p-1}^2}{\chi_{n-p}^2} \quad (37.2.22)$$

where  $\tilde{\rho}^2 = \rho^2/(1-\rho^2)$ ,  $n$  is the sample size less one,  $z$  is a standard normal variate,  $\chi_f$  and  $\chi_f^2$  are chi and chi-square variates on  $f$  degrees of freedom; the variates figuring in this relation are independently distributed and  $\tilde{\rho}$  is taken to be positive [Lee \(1971c\)](#).

### 37.2.6 Confidence Limits for the Noncentrality Parameter

---

Function **Rho2DistNoncentrality**(*alpha* As mpNum, *Rho2* As mpNum, *p* As mpNum, *N* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **Rho2DistNoncentrality** returns confidence limits for the noncentrality parameter  $\rho^2$  and related information for the distribution of the squared population multiple correlation coefficient.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*Rho2*: A real number greater or equal 0, representing the squared sample multiple correlation coefficient

*p*: An integer greater 2, representing the number of variates

*N*: A real number greater *p*, representing the sample size

*Output*: A string describing the output choices

See section [32.1.3.5](#) for the options for *alpha*, *Noncentrality* and *Output*. Algorithms and formulas are given below.

Let  $T$  be a statistic according to the non-central  $t$ -distribution with  $n$  degrees of freedom and a non-centrality parameter  $\delta$ . Then the lower confidence limit  $\hat{\delta}$  of level  $1 - \alpha$  and the two-sided confidence interval  $[\underline{\delta}, \bar{\delta}]$  of the non-centrality parameter  $\delta$  of level  $1 - \alpha$  are given by [Akahira et al. \(1995\)](#):

$$\hat{\delta} = bT - z_\alpha \sqrt{k} + hT^3(z_\alpha^2 - 1)/k, \quad (37.2.23)$$

$$\underline{\delta} = bT - z_{\alpha/2} \sqrt{k} + hT^3(z_{\alpha/2}^2 - 1)/k, \quad (37.2.24)$$

$$\bar{\delta} = bT + z_{\alpha/2} \sqrt{k} - hT^3(z_{\alpha/2}^2 - 1)/k, \quad (37.2.25)$$

where  $k = 1 + (1 - b^2)T^2$ ,  $h$  and  $b$  are defined in equation [\(37.8.19\)](#), and  $z_\alpha$  denotes the  $\alpha$ -quantile of the normal distribution (see section [32.11.3](#)).

### 37.2.7 Sample Size

---

Function **Rho2DistSampleSize**(*alpha* As mpNum, *beta* As mpNum, *p* As mpNum, **Modified-Noncentrality** As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function `Rho2DistSampleSize` returns sample size estimates and related information for the distribution of the squared population multiple correlation coefficient

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*beta*: A real number between 0 and 1, specifies the Type II error (or 1 - Power).

*p*: An integer greater 2, representing the number of variates

*ModifiedNoncentrality*: A real number greater 0, representing the modified noncentrality parameter.

*Output*: A string describing the output choices

See section [32.1.3.4](#) for the options for *alpha*, *beta*, *ModifiedNoncentrality* and *Output*. Algorithms and formulas are given below.

## 37.3 Skew Normal Distribution

### 37.3.1 Definition

In probability theory and statistics, the skew normal distribution is a continuous probability distribution that generalises the normal distribution to allow for non-zero skewness.

See also [http://en.wikipedia.org/wiki/Skew\\_normal\\_distribution](http://en.wikipedia.org/wiki/Skew_normal_distribution).

### 37.3.2 Density and CDF

---

Function **SkewNormalDistBoost**(*x* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **SkewNormalDistBoost** returns returns pdf, CDF and related information for the skew normal distribution

**Parameters:**

- x*: A real number.
- a*: The location parameter.
- b*: The scale parameter
- c*: The shape parameter

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 37.3.2.1 and 37.3.2.2.

#### 37.3.2.1 Density

The probability density function with location parameter *a*, scale parameter *b*, and shape parameter *c* is

$$f(x; a, b, c) = \frac{2}{b} \phi\left(\frac{x-a}{b}\right) \Phi\left(c\left(\frac{x-a}{b}\right)\right) \quad (37.3.1)$$

#### 37.3.2.2 CDF

$$F(a, b, c) = \Phi\left(\frac{x-a}{b}\right) - 2T\left(\frac{x-a}{b}, c\right) \quad (37.3.2)$$

where  $T_{\text{Owen}}(\cdot, \cdot)$  denotes Owen's *T* function (see section 37.3.6).

### 37.3.3 Quantiles

---

Function **SkewNormalDistInvBoost**(*Prob* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, *c* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **SkewNormalDistInvBoost** returns returns quantiles and related information for the the skew normal distribution

**Parameters:**

- Prob*: A real number between 0 and 1.

*a*: The location parameter.

*b*: The scale parameter

*c*: The shape parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). The quantile is determined using an iterative algorithm.

### 37.3.4 Properties

---

Function **SkewNormalDistInfoBoost**(*a* As mpNum, *b* As mpNum, *c* As mpNum, **Output** As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function SkewNormalDistInfoBoost returns returns moments and related information for the skew normal distribution

**Parameters:**

*a*: The location parameter.

*b*: The scale parameter

*c*: The shape parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.3.4.1 Moments

$$\mu_1 = a + bd\sqrt{\frac{2}{\pi}}, \quad \text{where } d = \frac{c}{\sqrt{1+c^2}} \quad (37.3.3)$$

$$\mu_2 = b^2 \left(1 - \frac{2d^2}{\pi}\right) \quad (37.3.4)$$

$$\gamma_1 = \frac{4-\pi}{2} \frac{\left(d\sqrt{2/\pi}\right)^3}{(1-2d^2/\pi)^{3/2}} \quad (37.3.5)$$

$$\gamma_2 = 2(\pi-3) \frac{\left(d\sqrt{2/\pi}\right)^4}{(1-2d^2/\pi)^2} \quad (37.3.6)$$

### 37.3.5 Random Numbers

---

Function **SkewNormalDistRanBoost**(**Size** As mpNum, *a* As mpNum, *b* As mpNum, *c* As mpNum, **Generator** As String, **Output** As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function SkewNormalDistRanBoost returns returns random numbers following a skew normal distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*a*: The location parameter.

*b*: The scale parameter

*c*: The shape parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section [32.1.3.6](#) for the options for *Size*, *Generator* and *Output*.

### 37.3.6 Owen's T-Function

---

Function **TOwenBoost**(*h* As mpNum, *a* As mpNum) As mpNum

---

The function **TOwenBoost** returns Owen's T-Function

**Parameters:**

*h*: A real number.

*a*: A real number.

Owen's T-Function is defined as ([Owen, 1956](#)):

$$T(h, a) = \frac{1}{2\pi} \int_0^a \frac{\exp\left[-\frac{1}{2}h^2(1+x^2)\right]}{1+x^2} dx \quad (37.3.7)$$

The implementation uses the algorithm described in [Patefield & Tand \(2000\)](#).

## 37.4 Multivariate Normal Distribution

### 37.4.1 Definition

An  $n$ -dimensional random variable  $\mathbf{X}$  with mean vector  $\boldsymbol{\mu}$  and covariance matrix  $\boldsymbol{\Sigma}$  is said to have a nonsingular multivariate normal distribution, in symbols  $\mathbf{X} \sim \mathcal{N}_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ , if  $\boldsymbol{\Sigma}$  is positive definite, and the density function of  $\mathbf{X}$  is of the form given in equation 37.4.1.

See also [Steck & Owen \(1962\)](#) and [Owen & Steck \(1962\)](#)

See [Lai \(2006\)](#)

### 37.4.2 Density of the multivariate normal distribution

The density of the multivariate normal distribution is given by ([Tong, 1990](#)):

$$f(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2} |\boldsymbol{\Sigma}|^{1/2}} e^{-Q_n(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})/2}, \quad \mathbf{x} \in \mathbb{R}^n \quad (37.4.1)$$

where

$$Q_n(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (\mathbf{x} - \boldsymbol{\mu})' \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}). \quad (37.4.2)$$

### 37.4.3 Bivariate Normal

The bivariate normal distribution has the following density:

$$g(x, y; \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} e^{\frac{-(x^2-2\rho xy+y^2)}{2(1-\rho^2)}} \quad (37.4.3)$$

The method developed by [Owen \(1956\)](#) was for a long time the most widely used approach to calculate bivariate normal (BVN) probabilities. Owen showed that

$$\Phi_2(-\infty, b1, b2; \rho) = \frac{\Phi(b1) - \Phi(b2)}{2} - T_{\text{Owen}}(b1, \hat{b}_1) - T_{\text{Owen}}(b2, \hat{b}_2) - c, \quad (37.4.4)$$

where  $\rho$  is the correlation coefficient,  $T_{\text{Owen}}(\cdot, \cdot)$  denotes Owen's  $T$  function (see section 37.3.6), and

$$c = \begin{cases} 0, & \text{if } b_1 b_2 > 0 \text{ or } b_1 b_2 = 0, b_1 + b_2 \geq 0 \\ \frac{1}{2} & \text{otherwise,} \end{cases} \quad (37.4.5)$$

$$\hat{b}_1 = \frac{b_2 - b_1 \rho}{b_1 \sqrt{1 - \rho^2}}, \quad \hat{b}_2 = \frac{b_2 - b_2 \rho}{b_2 \sqrt{1 - \rho^2}}. \quad (37.4.6)$$

[Abramowitz & Stegun. \(1970\)](#) gives the following representation:

$$B(h, k; \rho) = \int_{-\infty}^k \int_{-\infty}^h g(x, y; \rho) dx dy \quad (37.4.7)$$

$$L(h, k; \rho) = \int_k^{\infty} \int_h^{\infty} g(x, y; \rho) dx dy \quad (37.4.8)$$

$$L(-h, -k; \rho) = B(h, k; \rho) \quad (37.4.9)$$

Integral representation [Tong \(1990\)](#):

$$B(h, k; \rho) = \int_{-\infty}^{\infty} \prod_{j=1}^2 \left( \frac{d_j \sqrt{|\rho|} z + a_j}{\sqrt{1-\rho}} \right) \phi(z) dz \quad (37.4.10)$$

where  $d_j = -1$  if  $\rho < 0$  and  $j = 2$  and  $d_j = 1$  otherwise.

### 37.4.4 Special structure: zero correlation

This case covers the normal maximum

$$F_{MM}(x, k, \mu) = \prod_{i=1}^k [\Phi(x - \mu_i)] \quad (37.4.11)$$

$$F_M(x, k) = [\Phi(x)]^k \quad (37.4.12)$$

This case covers the normal maximum modulus

$$F_{MM}(x, k, \mu) = \prod_{i=1}^k [\Phi(x - \mu_i) - \Phi(-x - \mu_i)] \quad (37.4.13)$$

$$F_{MM}(x, k, 0) = [\Phi(x) - \Phi(-x)]^k = [2\Phi(x) - 1]^k, \quad x > 0 \quad (37.4.14)$$

See [Stoline & Ury \(1979\)](#)

See [Narula \(1978\)](#)

### 37.4.5 Special structure: Equal-correlated case

With  $\rho_{ij} = \rho$ , we have for a one-sided test [Tong \(1990\)](#):

$$F_n(h; \rho) = \int_{-\infty}^{\infty} \left[ \Phi \left( \frac{h + \sqrt{|\rho|} y}{\sqrt{1-\rho}} \right) \right]^n \phi(y) dy = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} g(h; \rho) dx_1 \cdots dx_n. \quad (37.4.15)$$

and for a two-sided test:

$$F_n(h; \rho) = \int_{-\infty}^{\infty} \left[ \Phi \left( \frac{h + \sqrt{|\rho|} y}{\sqrt{1-\rho}} \right) - \Phi \left( \frac{-h + \sqrt{|\rho|} y}{\sqrt{1-\rho}} \right) \right]^n \phi(y) dy = \int_{-h}^{\infty} \cdots \int_{-h}^{\infty} g(h; \rho) dx_1 \cdots dx_n. \quad (37.4.16)$$

Note: The distribution of the twosided version can get bi-modal for large arguments, if  $P(x)$  is calculated. This requires modification of the search algorithm.

### 37.4.6 Special structure: Dunnett test

See [Dunnett \(1955\)](#)

See also [Cheng & Meng \(1995\)](#)

For  $\lambda_i = \sqrt{\rho} \geq 0$  for all  $i$ , this reduces to the equicorrelated case.

With  $\rho_{ij} = \lambda_i \lambda_j$ , we have for a one-sided test:

$$F_n(h; \rho_{ij}) = \int_{-\infty}^{\infty} \prod_{i=1}^n \left[ \Phi \left( \frac{(a_i - \mu_i)/\sigma_i + \lambda_i z}{\sqrt{1 - \lambda_i^2}} \right) \right] \phi(y) dz \quad (37.4.17)$$

and for a two-sided test:

$$F_n(h; \rho_{ij}) = \int_{-\infty}^{\infty} \prod_{i=1}^n \left[ \Phi \left( \frac{(a_i - \mu_i)/\sigma_i + \lambda_i z}{\sqrt{1 - \lambda_i^2}} \right) - \Phi \left( \frac{(b_i - \mu_i)/\sigma_i + \lambda_i z}{\sqrt{1 - \lambda_i^2}} \right) \right] \phi(y) dz \quad (37.4.18)$$

Note: The distribution of the twosided version can get bi-modal for large arguments, if  $P(x)$  is calculated. This requires modification of the search algorithm.

### 37.4.7 Special structure: normal range

Let  $X_1, \dots, X_k$  be a random sample of size  $k$  from a  $N(\mu, \sigma^2)$  distribution. Let  $s^2$  be an independent mean square estimate of  $\sigma^2$  with  $n$  degrees of freedom.

Then  $Q_n = \max|X_i - X_j|, 1 < i < j < k$ , has a Normal Range distribution with  $k$  degrees of freedom, and  $Q_t = Q_n/s$  has a Studentized Range distribution with  $k$  and  $n$  degrees of freedom [Hochberg & Tamhane \(1987\)](#).

See also [Stoline \(1978\)](#) and [Harter \(1960\)](#) and [David et al. \(1972\)](#)

#### 37.4.7.1 Density

The density in the central case is given by

$$p(x) = k(k-1) \int_{-\infty}^{\infty} (\Phi(y) - \Phi(y-x))^{k-2} \phi(y) \phi(y-x) dy \quad (37.4.19)$$

#### 37.4.7.2 Normal Range

$$Q(x) = \sum_{i=1}^k \int_{-\infty}^{\infty} \phi(y_i - \mu_i) \left( \prod_{j=1, j \neq i}^k (\Phi(y_j - \mu_j) - \Phi(y_j - \mu_j - x)) \right) dy_i \quad (37.4.20)$$

for  $\mu = 0$  this simplifies to

$$Q(x) = k \int_{-\infty}^{\infty} \phi(y) (\Phi(y) - \Phi(y-x))^{k-1} dy \quad (37.4.21)$$

$$P(x) = 1 - Q(x) = k \int_{-\infty}^{\infty} \phi(y) (L_1^k - [L_1 - L_2]^{k-1}) dy, \quad \text{where} \quad (37.4.22)$$

$$L_1 = \Phi(y), \quad L_2 = \Phi(y-x), \quad L_2 \rightarrow 0 \text{ for } x \rightarrow \infty \quad (37.4.23)$$

$$L_1^k - (L_1 - L_2)^k = L_1^k \left( 1 - \left( 1 - \frac{L_2}{L_1} \right)^k \right) \quad (37.4.24)$$

### 37.4.8 Normal Orthant Probabilities

Let  $X = (X_1, X_2, \dots, X_n)$  be a random vector distributed as  $N(0, R_n)$ , where  $R_n$  is positive definite. The normal orthant probability is defined as

$$P_n = \int_0^{\infty} N(0, R_n) d^n x. \quad (37.4.25)$$

Based on the method of [Sun \(1988a,b\)](#), this probability can be expressed as

$$P_{2k} = \frac{1}{2^{2k}} + \frac{1}{2^{2k-1}\pi} \sum_{i < j=1}^{2k} \arcsin(r_{ij}) + \sum_{j=2}^k \frac{1}{2^{2k-j}\pi^j} \sum_{i_1 < i_2 < \dots < i_{2j}} I_{2j}(R^{(i_1 i_2 \dots i_{2j})}), \quad (37.4.26)$$

$$P_{2k+1} = \frac{1}{2^{2k+1}} + \frac{1}{2^{2k}\pi} \sum_{i=j=1}^{2k+1} \arcsin(r_{ij}) + \sum_{j=2}^k \frac{1}{2^{2k+1-j}\pi^j} \sum_{i_1 < i_2 < \dots < i_{2j}} I_{2j}(R^{(i_1 i_2 \dots i_{2j})}), \quad (37.4.27)$$

where  $R^{(i_1 i_2 \dots i_{2j})}$  denotes the submatrix consisting of the  $(i_1 i_2 \dots i_{2j})^{th}$  rows and columns of  $R_n$ ,

$$I_n(\Lambda_n) = (-2\pi)^{-k} \int_{-\infty}^{\infty} \prod_{i=1}^n \frac{1}{\omega_i} \exp(-\frac{1}{2}\omega^t \Lambda_n \omega) d^n \omega, \quad n = 2k, \quad (37.4.28)$$

and  $\Lambda_n = (\lambda_{ij})$  is a covariance matrix of  $n$  variates.

See [Genz & Bretz \(2002, 2009\)](#) for alternative methods

### 37.4.9 Normal Rank Order Probabilities

Consider the situation where random variables  $X_1, \dots, X_m$  and  $X_1, \dots, Y_n$  are normally distributed with means  $\mu_X$  and  $\mu_Y$ , respectively, and common variance  $\sigma^2$ , all  $m+n$  random variables being mutually independent and  $d = (\mu_Y - \mu_X)/\sigma$ .

Let  $\mathbf{U} = (U_1, \dots, U_{m+n})$ ,  $U_1 < \dots < U_{m+n}$ , denote the order statistics of the random variables  $(X_1, \dots, X_m)$ ,  $(Y_1, \dots, Y_n)$ , and let  $\mathbf{Z} = (Z_1, \dots, Z_{m+n})$  denote a random vector of zeros and ones, where the  $i$ th component  $Z_i$  is 0 (or 1) if  $U_i$  is an  $X$  (or  $Y$ ). Denote by  $\phi(x - \theta)$  the normal density with mean  $\theta$  and variance 1 (see section [32.11.2.1](#)). If  $\mathbf{z} = (z_1, \dots, z_{m+n})$  is a fixed vector of zeros and ones, the probability of the rank order  $z$ ,  $\Pr[\mathbf{Z} = \mathbf{z}]$ , is given by

$$P_{m,n}(\mathbf{z}|d) = m!n! \int_R \dots \int \prod_{i=1}^{m+n} \phi(t_i - z_i d) dt_i, \quad (37.4.29)$$

where the region of integration  $R$  is  $-\infty < t_1 \leq t_2 \leq \dots \leq t_{m+n} < \infty$ . [Milton \(1970\)](#) describes a  $p$ -dimensional midpoint algorithm that economizes the number of arithmetic operations required to evaluate equation (37.4.29), corrects for effects of the edges of the region of integration and involves no high-degree quadrature formulas.

### 37.4.10 Random Numbers

#### 37.4.10.1 Generation of Exchangeable Normal Variates

Consider the situation in which we are interested in generating  $N$  (pseudo) independent  $n$ -dimensional variates  $\mathbf{X}_1, \dots, \mathbf{X}_N$ , such that  $\mathbf{X} = (X_{1t}, \dots, X_{nt})'$  ( $t = 1, \dots, N$ ) has an  $\mathcal{N}_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$  distribution with a common mean  $\boldsymbol{\mu}$ , a common variance  $\sigma^2$ , and a common correlation coefficient  $\rho \in [0, 1]$ . Then a corresponding algorithm for generating  $\mathbf{X}_1, \dots, \mathbf{X}_N$  is:

Input  $\boldsymbol{\mu}$ ,  $\sigma^2$ ,  $\rho$ ,  $n$ , and  $N$ .

For  $t = 1, \dots, N$  compute

$$X_{it} = \mu + \sigma \left( \sqrt{1 - \rho} Z_{it} + \sqrt{\rho} Z_{0t} \right) \quad (37.4.30)$$

and form  $\mathbf{X} = (X_{1t}, \dots, X_{nt})'$ .

#### 37.4.10.2 Generation of Multivariate Normal Variates with a Special Correlation Structure

In certain statistical applications the covariance matrix  $\boldsymbol{\Sigma} = (\sigma_{ij})$  may be of the form

$$\sigma_{ij} = \begin{cases} \sigma_i & \text{for } i = j \\ \sigma_i \sigma_j \lambda_i \lambda_j & \text{for } i \neq j, \end{cases} \quad (37.4.31)$$

where  $\lambda_i \in [-1, 1](i = 1, \dots, n)$ . In this case, the correlation coefficients are  $\rho_{ij} = \lambda_i \lambda_j$  for all  $i \neq j$ . To generate  $\mathbf{X}_1, \dots, \mathbf{X}_N$  according to an  $\mathcal{N}_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$  distribution when  $\boldsymbol{\Sigma}$  has such a structure and  $\boldsymbol{\mu} = (\mu_1, \dots, \mu_n)'$ , we replace equation 37.4.30 by

$$X_{it} = \mu_i + \sigma_i \left( \sqrt{1 - \lambda_i^2} Z_{it} + \lambda_i Z_{0t} \right) \quad (37.4.32)$$

and otherwise follow the algorithm in section 37.4.10.1. Note that here the  $\mu_i$  are not necessarily the same and the correlation coefficients are not necessarily all nonnegative. If  $\mu_i = \mu$ ,  $\sigma_i = \sigma$ , and  $\lambda_i = \sqrt{\rho} \geq 0(i = 1, \dots, n)$ , then this algorithm reduces to the one in section 37.4.10.1 as a special case.

### 37.4.10.3 Generation of Multivariate Normal Variates with an Arbitrary Nonsingular Multivariate Normal Distribution

We are interested in generating  $\mathbf{X}_1, \dots, \mathbf{X}_N$  from an  $\mathcal{N}_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$  distribution with arbitrary but fixed  $\boldsymbol{\mu}$  and  $\boldsymbol{\Sigma}$  (which is positive definite).

Let  $\mathbf{T} = (\tau_{ij})$  be the unique lower triangular  $n \times n$  matrix (obtained by Cholesky decomposition) such that  $\mathbf{T}\mathbf{T}' = \boldsymbol{\Sigma}$ . If  $\mathbf{Z} \sim \mathcal{N}_n(\mathbf{0}, \mathbf{I}_n)$ , then  $\mathbf{X} = \mathbf{T}\mathbf{Z} + \boldsymbol{\mu}$  has an  $\mathcal{N}_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$  distribution.

Consequently, to generate independent random variates  $\mathbf{X}_1, \dots, \mathbf{X}_N$  according to this distribution, the following algorithm may be used:

Compute  $\mathbf{T} = (\tau_{ij})$

Input  $N, n, \boldsymbol{\mu} = (\mu_1, \dots, \mu_n)'$  and  $\mathbf{T} = (\tau_{ij})$ .

Generate  $Z_{1t}, \dots, Z_{nt}$  (which are (pseudo) independent  $\mathcal{N}(0, 1)$  variates) and apply the transformation  $\mathbf{X}_t = \mathbf{T}\mathbf{Z}_t + \boldsymbol{\mu}$ , i.e. compute

$$X_{it} = \sum_{j=1}^i \tau_{ij} Z_j + \mu_i, \quad \text{for } i = 1, \dots, n, \quad (37.4.33)$$

and then form  $\mathbf{X} = (X_{1t}, \dots, X_{nt})'$ .

Repeat this step for  $t = 1, \dots, N$ .

## 37.5 Multivariate t-Distribution

### 37.5.1 Definition

Let  $\mathbf{R} = (\rho_{ij})$  be an  $n \times n$  symmetric matrix such that it is either positive definite or positive semidefinite and  $\rho_{ii} = 1$  ( $i = 1, \dots, n$ ). Let  $\mathbf{Z} = (Z_1, \dots, Z_n)'$  have an  $\mathcal{N}_n(\mathbf{0}, \mathbf{R})$  distribution, and let the univariate variable  $S$  be such that  $S$  is independent of  $\mathbf{Z}$ , and  $\nu S^2$  has a  $\chi^2(\nu)$  distribution. Then a natural generalization of the Student's  $t$  variable is

$$\mathbf{t} = (t_1, \dots, t_n)' = \left( \frac{Z_1}{S}, \dots, \frac{Z_n}{S} \right)'. \quad (37.5.1)$$

If  $\mathbf{R}$  is positive definite, then the density of  $\mathbf{t}$  (with correlation matrix  $\mathbf{R}$  and degrees of freedom  $\nu$ ) is given by (Tong, 1990):

$$h(\mathbf{t}; \mathbf{R}, \nu) = \frac{\Gamma((n + \nu)/2)}{(\nu\pi)^{n/2}\Gamma(\nu/2)|\mathbf{R}|^{1/2}} \left( 1 + \frac{1}{\nu} \mathbf{t}' \mathbf{R}^{-1} \mathbf{t} \right)^{-(n+\nu)/2}, \quad \mathbf{t} \in \mathbb{R}^n. \quad (37.5.2)$$

### 37.5.2 Studentized Maximum and Maximum Modulus

$$\frac{\partial}{\partial c} (F(cx)g(x)) = xg(x)F'(cx) \quad (37.5.3)$$

#### 37.5.2.1 Density and CDF

Let  $X_1, \dots, X_k$  be a random sample of size  $k$  from a  $\mathcal{N}(0, \sigma^2)$  distribution. Let  $s^2$  be an independent mean square estimate of  $\sigma$  with  $n$  degrees of freedom. Then

$$Q = \frac{\max X_j}{s}, \quad j = 1, \dots, k \quad (37.5.4)$$

has a Studentized Maximum distribution with  $k$  and  $n$  degrees of freedom, and

$$Q = \frac{\max |X_j|}{s}, \quad j = 1, \dots, k \quad (37.5.5)$$

has a Studentized Maximum Modulus distribution with  $k$  and  $n$  degrees of freedom.

#### 37.5.2.2 Quantiles

Studentized Maximum Critical Values  $M_{k,\nu,\alpha}$  (One-sided Multivariate  $t$  Critical Values  $t_{k,\nu,\rho=0,\alpha}$ )  
 Studentized Maximum Modulus Critical Values  $|M|_{k,\nu,\alpha}$  (Two-sided Multivariate  $t$  Critical Values  $|t|_{k,\nu,\rho=0,\alpha}$ )

### 37.5.3 Dunnett's t

#### 37.5.3.1 Density and CDF

Let  $X_1, \dots, X_k$  be a random sample of size  $k$  from a  $\mathcal{N}(0, \sigma^2)$  distribution. Let  $s^2$  be an independent mean square estimate of  $\sigma$  with  $n$  degrees of freedom. Then

$$Q = \frac{\max X_1 - X_j}{s}, \quad 1 < j < k \quad (37.5.6)$$

has a onesided Dunnett's  $t$ -distribution with  $k$  and  $n$  degrees of freedom, and

$$Q = \frac{\max_s |X_1 - X_j|}{s}, \quad 1 < j < k \quad (37.5.7)$$

has a twosided Dunnett's  $|t|$ -distribution with  $k$  and  $n$  degrees of freedom.

For Dunnett's test:

$$\lambda_i = \frac{1}{\sqrt{1 + n_0/n_i}} \quad (37.5.8)$$

### 37.5.3.2 Quantiles

One-Sided Multivariate t Critical Values  $t_{k,\nu,\rho=\lambda_i\lambda_j,\alpha}$  Common Correlation  $\rho = 0.5$

Two-Sided Multivariate t Critical Values  $|t|_{k,\nu,\rho=\lambda_i\lambda_j,\alpha}$  Common Correlation  $\rho = 0.5$

### 37.5.4 Studentized Range

Let  $X_1, \dots, X_k$  be a random sample of size  $k$  from a  $\mathcal{N}(0, \sigma^2)$  distribution. Let  $s^2$  be an independent mean square estimate of  $\sigma$  with  $n$  degrees of freedom. Then

$$Q = \frac{\max_s |X_i - X_j|}{s}, \quad 1 < i < j < k \quad (37.5.9)$$

has a Studentized Range distribution with  $k$  and  $n$  degrees of freedom.

#### 37.5.4.1 Density and CDF

$$Q_t(c; n) = \int_0^\infty Q_n(cx)g(x, n)dx, \quad \text{where} \quad (37.5.10)$$

$$g(x; n) = \frac{n^{n/2}}{2^{(n-1)/2}\Gamma(n/2)} x^{n-1} e^{-nx^2/2} \quad \text{is the pdf of } \frac{x}{\sigma} \quad (37.5.11)$$

$$SR(c) = \int_0^\infty Q(cx)g(x)dx, \quad \text{and for } n = \infty, \quad SR(c) = Q(c). \quad (37.5.12)$$

#### 37.5.4.2 Quantiles

Critical Values  $q_{k,\nu,\alpha}$  for the Studentized Range Distribution

See [Hirotzu \(1979\)](#)

### 37.5.5 Special case Owen

Owen considers a special case of the bivariate noncentral t-distribution which is relevant in quality control [Owen \(1965\)](#)

## 37.6 Noncentral Chi-Square Distribution

### 37.6.1 Definition

Let  $X_1, X_2, \dots, X_n$  be independent and identically distributed random variables each following a normal distribution with mean  $\mu_j$  and unit variance. Then  $\chi^2 = \sum_{j=1}^n X_j$  is said to follow a non-central  $\chi^2$ -distribution with  $n$  degrees of freedom and noncentrality parameter  $\lambda = \sum_{j=1}^n (\mu_j - \mu)$ .

### 37.6.2 Density and CDF

---

Function **NoncentralCDistEx**(*x* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **NoncentralCDistEx** returns pdf, CDF and related information for the noncentral  $\chi^2$ -distribution

**Parameters:**

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 37.6.2 and 37.6.2.3.

#### 37.6.2.1 Density: Closed form representations

The density of a noncentral chi-square variable with  $n$  degrees of freedom and  $\lambda$  is given by (Wang & Gray, 1993)

$$f_{\chi^2}(n, x; \lambda) = e^{-\lambda/2} f_{\chi^2}(n, x) {}_0F_1\left(-; \frac{n}{2}; \frac{x\lambda}{4}\right) \quad (37.6.1)$$

$$f_{\chi^2}(n, x; \lambda) = \frac{1}{2} e^{-(\lambda+x)/2} \left(\frac{x}{\lambda}\right)^{(n-2)/2} I_{(n-2)/2}(\sqrt{\lambda x}) \quad (37.6.2)$$

where  $f_{\chi^2}(n, \cdot)$  is the pdf of the (central)  $\chi^2$  distribution (see section 32.4.2.1),  ${}_0F_1(\cdot)$  is the confluent hypergeometric limit function (see section 11.1), and  $I_\nu(\cdot)$  is the modified Bessel function of the first kind of order  $\nu$  (see section ??).

#### 37.6.2.2 Density and CDF: Finite series for odd degrees of freedom

For odd degrees of freedom, the pdf and cdf can be expressed as a finite sum, using the recurrence relations given in section 37.6.4.2, and defining  $h(n, x, \lambda) = e^{(1/2)(x+\lambda)} f_{\chi^2}(n, x; \lambda)$  (András & Baricz, 2008):

$$F_{\chi^2}(1, x; \delta^2) = \Phi(x - \delta) - \Phi(-x - \delta) \quad (37.6.3)$$

$$h(1, x; \lambda) = \frac{\cosh(\sqrt{x\lambda})}{\sqrt{2\pi x}}, \quad h(3, x; \lambda) = \frac{\sinh(\sqrt{x\lambda})}{\sqrt{2\pi\lambda}} \quad (37.6.4)$$

where  $\Phi(\cdot)$  denotes the cdf of the normal distribution (see section 32.11.2.2).

### 37.6.2.3 CDF: General formulas

The cdf of a noncentral chi-square variable with  $n$  degrees of freedom and  $\lambda$  is given by

$$\Pr [\chi^2 \leq x] = F_{\chi^2}(n, x; \lambda) = \int_0^x f_{\chi^2}(n, t; \lambda) dt \quad (37.6.5)$$

### 37.6.2.4 CDF: Infinite series in terms of the central cdf

The cdf of a noncentral chi-square variable with  $n$  degrees of freedom and  $\lambda$  is given by

$$F_{\chi^2}(n, x; \lambda) = e^{-\lambda/2} \sum_{j=0}^{\infty} \frac{(\lambda/2)^j}{j!} F_{\chi^2}(n + 2 + j, x) \quad (37.6.6)$$

where  $F_{\chi^2}(n, \cdot)$  is the cdf of the (central)  $\chi^2$  distribution (see section 32.4.2.2).

### 37.6.2.5 CDF: Infinite series in terms of the central pdf

Ding (1992) gives the following representation (this is used by Boost for small lambda):

$$F_{\chi^2}(n, x; \lambda) = 2e^{-\lambda/2} \sum_{i=0}^{\infty} f_{\chi^2}(n + 2 + 2i, x) \left( \sum_{k=0}^i \frac{(\lambda/2)^k}{k!} \right) \quad (37.6.7)$$

where  $f_{\chi^2}(n, \cdot)$  is the pdf of the (central)  $\chi^2$  distribution (see section 32.4.2.1).

### 37.6.2.6 CDF: Integral representation

Chou (1985) gives the following representation for  $n \geq 2$  and  $\lambda \geq 0$ :

$$F_{\chi^2}(n, x; \lambda) = \frac{2^{(1-n)/2} \sqrt{2\pi}}{\Gamma((n-1)/2)} \int_0^x y^{(n-3)/2} \phi(\sqrt{y}) [\Phi(\sqrt{x-y} - \sqrt{\lambda}) - \Phi(-\sqrt{x-y} - \sqrt{\lambda})] dy \quad (37.6.8)$$

where  $\phi(\cdot)$  denotes the pdf of the normal distribution (see section 32.11.2.1) and  $\Phi(\cdot)$  denotes the cdf of the normal distribution (see section 32.11.2.2).

### 37.6.2.7 CDF: 2 moment approximation

Patnaik (1949) gives the following 2-moment approximation based on the central  $\chi^2$ -distribution:

$$F_{\chi^2}(n, x; \lambda) \approx F_{\chi^2}(n_1, x_1; ), \quad \text{where } n_1 = \frac{(n + \lambda)^2}{n + 2\lambda}, \quad x_1 = \frac{x(n + \lambda)}{n + 2\lambda} \quad (37.6.9)$$

where  $F_{\chi^2}(n, \cdot)$  is the cdf of the (central)  $\chi^2$  distribution (see section 32.4.2.2).

### 37.6.2.8 CDF: 3 moment approximation

Pearson (1959) gives the following 3-moment approximation based on the central  $\chi^2$ -distribution:

$$F_{\chi^2}(n, x; \lambda) \approx F_{\chi^2}(n_1, x_1; ), \quad \text{where } n_1 = \frac{(n + 2\lambda)^3}{2(n + 3\lambda)^2}, \quad x_1 = \frac{x + \lambda^2/(n + 3\lambda)}{2(n + 3\lambda)/n + 2\lambda} \quad (37.6.10)$$

where  $F_{\chi^2}(n, \cdot)$  is the cdf of the (central)  $\chi^2$  distribution (see section 32.4.2.2).

### 37.6.2.9 CDF: Saddlepoint approximation

Butler (2007) suggests the following approximation:

$$K(t) = -\frac{n}{2} \log(1 - 2t) + \frac{\lambda t}{1 - 2t}, \quad t \in (-\infty, \frac{1}{2}) \quad (37.6.11)$$

$$K^{(j)}(t) = \frac{2^{j-1}(j-1)!}{(1-2t)^j} \left[ n + \frac{\lambda j}{1-2t} \right] \quad (37.6.12)$$

$$s(x) = -\frac{1}{4x} \left[ n - 2x + \sqrt{n^2 + 4x\lambda} \right], \quad x > 0 \quad (37.6.13)$$

### 37.6.2.10 CDF: Wiener Germ approximation

Penev & Raykov (2000) give the following first and second order Wiener germ approximation:

$$F_{\chi^2}(n, x; \lambda) \approx \Phi \left( \text{sgn}(s) \sqrt{n(s-1)^2(1/(2s) + m - h(1-s)/s) - \ln(A(s)) + 2B(s)/n} \right) \quad (37.6.14)$$

$$\text{where } m = \lambda/n; \quad h(y) = \frac{(1-y)\ln(1-y) + y - \frac{1}{2}y^2}{y^2}; \quad s = \frac{\sqrt{1+4xm/n} - 1}{2m} \quad (37.6.15)$$

$$A(s) = \frac{1}{s} - \frac{2}{s} \cdot \frac{h(1-s)}{1+2ms}; \quad B(s) = \frac{(1+3m)^2}{9(1+2m)^3} \quad (37.6.16)$$

where  $\Phi(\cdot)$  denotes the cdf of the normal distribution (see section 32.11.2.2).

## 37.6.3 Quantiles

---

Function **NoncentralCDistInvEx**(*Prob* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function NoncentralCDistInvEx returns quantiles and related information for the noncentral  $\chi^2$ -distribution

### Parameters:

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below

The quantile is approximated as

$$\chi_{n,\lambda,\alpha}^2 \approx (1+b)\chi_{n_1,\lambda,\alpha}^2, \quad \text{where } n_1 = \frac{(n+\lambda)^2}{n+2\lambda}, \quad b = \frac{\lambda}{n+\lambda} \quad (37.6.17)$$

### 37.6.4 Properties

---

Function **NoncentralCDistInfoEx**(*n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **NoncentralCDistInfoEx** returns moments and related information for the noncentral  $\chi^2$ -distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.6.4.1 Moments and Cumulants

The cumulants of noncentral  $\chi^2$  are given by

$$\kappa_r(n, \lambda) = 2^{r-1}(r-1)!(n+r\lambda) \quad (37.6.18)$$

The first 4 cumulants of cube root noncentral  $\chi^2$ ,  $\chi^{2/3}$ , are given by  $\kappa_i^*(n, \lambda) = T_i(1, n, \lambda)$ , with  $r = n + \lambda$ ,  $b = \lambda/r$  (see (Abdel-Aty, 1954)):

$$\begin{aligned} T_1(c, n, \lambda) &= \left(\frac{cr}{n}\right)^{1/3} \left(1 - \frac{2(b+1)}{9r} - \frac{40b^2}{3^4 r^2} + \frac{80(1+3b+33b^2-77b^3)}{3^5 r^3}\right) \\ &+ \left(\frac{cr}{n}\right)^{1/3} \left(\frac{176(1+4b-210b^2+2380b^3-2975b^4)}{3^9 r^4}\right) + o(r^{-4}) \end{aligned} \quad (37.6.19)$$

$$\begin{aligned} T_2(c, n, \lambda) &= \left(\frac{cr}{n}\right)^{2/3} \left(\frac{2(b+1)}{9r} - \frac{16b^2}{3^4 r^2} - \frac{8(13+39b+405b^2-1025b^3)}{3^7 r^3}\right) \\ &+ \left(\frac{cr}{n}\right)^{2/3} \left(\frac{160(1+4b-87b^2+1168b^3-1544b^4)}{3^8 r^4}\right) + o(r^{-4}) \end{aligned} \quad (37.6.20)$$

$$\begin{aligned} T_3(c, n, \lambda) &= \left(\frac{-cr}{n}\right) \left(\frac{8b^2}{3^3 r^2} - \frac{32(1+3b+21b^2-62b^3)}{3^6 r^3}\right) \\ &+ \left(\frac{-cr}{n}\right) \left(\frac{32(8+32b-177b^2+4550b^3-6625b^4)}{3^8 r^4}\right) + o(r^{-4}) \end{aligned} \quad (37.6.21)$$

$$\begin{aligned} T_4(c, n, \lambda) &= -\left(\frac{cr}{n}\right)^{4/3} \left(\frac{16(1+3b+12b^2-44b^3)}{3^6 r^3}\right) \\ &- \left(\frac{cr}{n}\right)^{4/3} \left(\frac{256(1+4b-6b^2+274b^3-458b^4)}{3^8 r^4}\right) + o(r^{-4}) \end{aligned} \quad (37.6.22)$$

### 37.6.4.2 Recurrence Relations

The following recurrence relations hold for the pdf and CDF (Cohen, 1988):

$$f_{\chi^2}(n+4, x; \lambda) = \frac{x \cdot f_{\chi^2}(n, x; \lambda) - n \cdot f_{\chi^2}(n+2, x; \lambda)}{\lambda} \quad (37.6.23)$$

$$F_{\chi^2}(n, x; \lambda) - F_{\chi^2}(n+2, x; \lambda) = 2f_{\chi^2}(n+2, x; \lambda) \quad (37.6.24)$$

$$F_{\chi^2}(n, x; \lambda) - F_{\chi^2}(n-2, x; \lambda) = 2 \frac{\partial}{\partial \lambda} F_{\chi^2}(n-2, x; \lambda) \quad (37.6.25)$$

### 37.6.5 Confidence Limits for the Noncentrality Parameter

---

Function **NoncentralCDistNoncentralityEx**(*alpha* As mpNum, *lambda* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **NoncentralCDistNoncentralityEx** returns confidence limits for the noncentrality parameter lambda and related information for the noncentral  $\chi^2$ -distribution.

#### Parameters:

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*lambda*: A real number greater 0, representing the noncentrality parameter.

*n*: A real number greater 0, representing the degrees of freedom.

*Output*: A string describing the output choices

See section 32.1.3.5 for the options for *alpha*, *Noncentrality* and *Output*. Algorithms and formulas are given below.

(Winterbottom, 1979) gives the following formula to determine the parameter  $\lambda$  of a noncentral  $\chi^2$  distribution with  $n$  degrees of freedom, so that  $F_{\chi^2}(n, x; 0) = 1 - \alpha$  and  $F_{\chi^2}(n, x; \lambda) = 1 - \beta$ . Let  $c$  be the  $1 - \alpha$  percentage point of a  $\chi^2$ -distribution with  $n$  degrees of freedom, let  $x$  be the  $1 - \beta$  percentage point of a  $N(0, 1)$  distribution, and  $T = (c - n)/n$ ,  $Y = 2T + 1$ . Then

$$\begin{aligned} \lambda \approx nT + \sqrt{2nY}x + \frac{2((3T+2)x^2 + 3T+1)}{3Y} - \frac{(6T+5)x^3 - (36T^2 + 42T + 17)x}{18\sqrt{nY^5/2}} \\ + \frac{(324T^2 + 594T + 276)x^4}{405nY^4} - \frac{(1080T^3 + 2484T^2 + 976)x^2}{405nY^4} \\ + \frac{1080T^3 + 1512T^2 + 612T + 148}{405nY^4} - \frac{(10368T^3 + 30780T^2 + 30564T + 10143)x^5}{9720\sqrt{n^3Y^{11}/2}} \\ + \frac{(25920T^4 + 98928T^3 + 163080T^2 + 137544T + 47188)x^3}{9720\sqrt{n^3Y^{11}/2}} \\ + \frac{(45360T^4 + 106704T^3 + 80460T^2 + 31092T + 13489)x}{9720\sqrt{n^3Y^{11}/2}} \end{aligned} \quad (37.6.26)$$

### 37.6.6 Sample Size Calculation

---

Function **NoncentralCDistSampleSizeEx**(*alpha* As mpNum, *beta* As mpNum, *n* As mpNum, *ModifiedNoncentrality* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralCDistSampleSizeEx returns sample size estimates and related information for the noncentral  $\chi^2$ -distribution.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*beta*: A real number between 0 and 1, specifies the Type II error (or 1 - Power).

*n*: A real number greater 0, representing the denominator degrees of freedom.

*ModifiedNoncentrality*: A real number greater 0, representing the modified noncentrality parameter.

*Output*: A string describing the output choices

See section 32.1.3.4 for the options for *alpha*, *beta*, *ModifiedNoncentrality* and *Output*. Algorithms and formulas are given below.

### 37.6.7 Random Numbers

#### 37.6.7.1 Noncentral Chi-Square

---

Function **NoncentralCDistRanEx**(*Size* As mpNum, *n* As mpNum, *lambda* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralCDistRanEx returns random numbers following a noncentral  $\chi^2$ -distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below

Random numbers from a non-central chi-square distribution is easily obtained using the definition above by e.g.

1. Put  $\mu = \sqrt{\lambda/n}$
2. Sum  $n$  random numbers from a normal distribution with mean  $\mu$  and variance unity. Note that this is not a unique choice. The only requirement is that  $\lambda = \sum \mu_i^2$ .
3. Return the sum as a random number from a non-central chi-square distribution with  $n$  degrees of freedom and non-central parameter  $\lambda$ .

### 37.6.7.2 Central Wishart distribution

To simulate a standard central Wishart matrix, in symbols  $\mathbf{V} \sim \mathcal{W}_s(k, \mathbf{I}_s, \mathbf{0})$ , we use the method proposed by Odell and Feiveson:

Generate  $s(s-1)/2$  i.i.d  $\mathcal{N}(0, 1)$  random variables  $N_{ij}$ ,  $1 \leq i \leq j \leq s$ .

Independently generate  $s$  independent random variables  $L_1, L_2, \dots, L_s$ , where  $L_j \sim \chi^2_{k-j+1}$ ,  $j = 1, 2, \dots, s$ . Then  $\mathbf{V} = (V_{ij}) \sim \mathcal{W}_s(k, \mathbf{I}_s, \mathbf{0})$ , with

$$V_{jj} = L_j + \sum_{i=1}^{j-1} N_{ij}^2, \quad j = 1, 2, \dots, s, \quad (37.6.27)$$

$$V_{ij} = N_{ij} \sqrt{L_i} + \sum_{\nu=1}^{j-1} N_{\nu i} N_{\nu j} = V_{ji}, \quad i < j. \quad (37.6.28)$$

### 37.6.7.3 Noncentral Wishart Distribution: Definition

The  $p \times p$  random Matrix is said to have a noncentral Wishart distribution, , in symbols  $\mathbf{A} \sim \mathcal{W}_p(n, \Sigma, \mathbf{M})$ , with noncentrality parameter  $\mathbf{M} : p \times p$  and degrees of freedom  $n$  if there exist independent  $p \times 1$  random vectors  $\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_n$  and constant  $p \times 1$  vectors  $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_n$  such that  $\mathbf{Y}_i \sim \mathcal{N}(0, 1)$ ,  $i = 1, 2, \dots, n$ , and

$$\mathbf{A} = \sum_{i=1}^n \mathbf{Y}_i \mathbf{Y}_i', \quad \mathbf{M} = \Sigma^{\frac{1}{2}} \left[ \sum_{i=1}^p \boldsymbol{\mu}_i \boldsymbol{\mu}_i' \right] \left( \Sigma^{\frac{1}{2}} \right)' \quad (37.6.29)$$

Here,  $\Sigma^{\frac{1}{2}}$  is any square root of  $\Sigma$ .

### 37.6.7.4 Noncentral Wishart distribution: Full Rank

To simulate a non-central Wishart matrix, in symbols  $\mathbf{A} \sim \mathcal{W}_p(n, \Sigma, \mathbf{M})$ , we use the method proposed by Gleser (1976):

$$\mathbf{A} = \Sigma^{\frac{1}{2}} \left[ \sum_{i=1}^p (\mathbf{Z}_i + \mathbf{M}_i)(\mathbf{Z}_i + \mathbf{M}_i)' + \mathbf{V} \right] \left( \Sigma^{\frac{1}{2}} \right)' \quad (37.6.30)$$

where  $\mathbf{Z}_1, \mathbf{Z}_2, \dots, \mathbf{Z}_p$  and  $\mathbf{V}$  are mutually independent,  $\mathbf{M} = (\mathbf{M}_1, \mathbf{M}_2, \dots, \mathbf{M}_p)$ ,  $\mathbf{Z}_i \sim \mathcal{N}_p(\mathbf{0}, \mathbf{I}_p)$ ,  $i = 1, 2, \dots, p$ , and  $\mathbf{V} \sim \mathcal{W}_p(n-p, \mathbf{I}_p, \mathbf{0})$ .

Generating  $\mathbf{Z}_1, \mathbf{Z}_2, \dots, \mathbf{Z}_p$  requires  $p^2$  i.i.d.  $\mathcal{N}(0, 1)$  variables, while  $p(p-1)/2$  additional i.i.d.  $\mathcal{N}(0, 1)$  variables and  $p$  independent  $\chi^2$  variables are needed to generate  $\mathbf{V}$  by means of the Odell-Feiveson method already described.

### 37.6.7.5 Noncentral Wishart distribution: Less Than Full Rank

As an alternative, the following representation can be used:

$$\mathbf{A} = \Sigma^{\frac{1}{2}} \Gamma \begin{pmatrix} \mathbf{Q} & \mathbf{Q}^{\frac{1}{2}} \mathbf{R} \\ \mathbf{Q}^{\frac{1}{2}} \mathbf{R} & \mathbf{W} + \mathbf{R}' \mathbf{R} \end{pmatrix} \Gamma' \left( \Sigma^{\frac{1}{2}} \right)', \quad \text{where} \quad (37.6.31)$$

$$\mathbf{M} = \Gamma \begin{pmatrix} \mathbf{D}_\delta & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \Gamma', \quad (37.6.32)$$

$\mathbf{Q}$ ,  $\mathbf{R}$  and  $\mathbf{W}$  are mutually independent,  $\mathbf{Q} \sim \mathcal{W}_t(n, \mathbf{I}_t, \mathbf{D}_\delta)$ ,  $\mathbf{W} \sim \mathcal{W}_{p-t}(n-t, \mathbf{I}_{p-t}, \mathbf{0})$ , the scalar elements  $r_{ij}$  of  $\mathbf{R}$  are i.i.d.  $\mathcal{N}(0, 1)$  and  $\mathbf{Q}^{\frac{1}{2}}$  is any square root of  $\mathbf{Q}$

Assuming that  $\Sigma^{\frac{1}{2}}$  and  $\Gamma$  are already calculated, we proceed as follows:

Generate  $\mathbf{W} \sim \mathcal{W}_{p-t}(n-t, \mathbf{I}_{p-t}, \mathbf{0})$  by the Odell-Feiveson method.

Generate  $t(p-t)$  i.i.d.  $\mathcal{N}(0, 1)$  variables to serve as the elements  $r_{ij}$  of  $\mathbf{R}$ .

Generate  $\mathbf{Q} \sim \mathcal{W}_t(n, \mathbf{I}_t, \mathbf{D}_\delta)$  by the standard method (based on equation 37.6.30) outlined above.

Take the square root of  $Q$  using the Cholesky decomposition, yielding a lower triangular matrix.

Form the matrix product  $\mathbf{Q}^{\frac{1}{2}} \mathbf{R}$ , taking advantage of the fact that  $\mathbf{Q}^{\frac{1}{2}}$  is lower triangular. Also, calculate  $\mathbf{R}' \mathbf{R}$ , and then  $\mathbf{W} + \mathbf{R}' \mathbf{R}$ .

Finally, calculate the matrix in equation 37.6.31.

## 37.7 Noncentral Beta-Distribution

### 37.7.1 Definition

If  $X_1$  and  $X_2$  are independent random variables,  $X_1$  following a non-central  $\chi^2$ -distribution with noncentrality parameter  $\lambda$  and  $2a$  degrees of freedom, and  $X_2$  following a  $\chi^2$ -distribution with  $2b$  degrees of freedom, then the distribution of the ratio  $B = \frac{X_1}{X_1+X_2}$  is said to follow a non-central Beta-distribution with  $a$  and  $b$  degrees of freedom.

Note: The univariate version of the noncentral distribution of Wilks Lambda: GLM (see section 37.13.1.1) is equivalent to  $W = 1 - B$

See [Tiwari & Yang \(1997\)](#)

### 37.7.2 Density and CDF

---

Function **NoncentralBetaDistBoost**(*x* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

**NOT YET IMPLEMENTED**

---

The function NoncentralBetaDistBoost returns pdf, CDF and related information for the central Beta-distribution

#### Parameters:

*x*: A real number

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

#### 37.7.2.1 Density

The density function of the noncentral beta-Distribution is given by ([Wang & Gray, 1993](#)): this needs to be checked, see Paolella 2007:

$$f_{\text{Beta}'}(x; n_1, n_2, \lambda) = e^{-\lambda/2} f_B(x; n_1, n_2) {}_1F_1\left(\frac{1}{2}(m+n), \frac{1}{2}n, \frac{nx\lambda}{2(m+nx)}\right) \quad (37.7.1)$$

where  $f_B(x; m, n)$  is the pdf of the (central) beta-distribution (see section 32.2.2.1) and  ${}_1F_1(\cdot)$  is the confluent hypergeometric function (see section 11.2.1).

#### 37.7.2.2 CDF: Infinite Series

The cdf can be calculated using the following infinite series [Benton & Krishnamoorthy \(2003\)](#):

$$\Pr[F \leq x] = F_{B'}(x; a, b, \lambda) = e^{-\lambda/2} \sum_{j=0}^{\infty} \frac{(\lambda/2)^j}{j!} I(a+j, b, x) \quad (37.7.2)$$

### 37.7.3 Quantiles

---

Function **NoncentralBetaDistInvBoost**(*Prob* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

**NOT YET IMPLEMENTED**

---

The function `NoncentralBetaDistInvBoost` returns quantiles and related information for the the noncentral Beta-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

### 37.7.4 Properties

---

Function **NoncentralBetaDistInfoBoost**(*m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function `NoncentralBetaDistInfoBoost` returns moments and related information for the non-central Beta-distribution

**Parameters:**

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.7.4.1 Moments of the non-central Beta-distribution

Algebraic moments of the non-central Beta-distribution are given as (see Walck (2007), p. 109)

$$\mathbb{E}(x^k) = e^{-\lambda/2} \sum_{r=0}^{\infty} \frac{(\lambda/2)^r}{r!} \frac{B(p+r+k, q)}{B(p+r, q)} \quad (37.7.3)$$

$$\mathbb{E}(x^k) = \frac{\Gamma(p+k)\Gamma(p+q)}{\Gamma(p)\Gamma(p+q+k)} \times e^{-\lambda/2} {}_2F_1(p+q, p+k; p, p+q+k; \lambda/2) \quad (37.7.4)$$

Lower order moments are given by

$$\mathbb{E}(Y) = \frac{a}{a+b} \times e^{-\lambda/2} {}_2F_1(a+1, a+b; a, a+b+1; \lambda/2) \quad (37.7.5)$$

$$\mathbb{E}(Y^2) = \frac{a(a+1)}{a+b(a+b+1)} \times e^{-\lambda/2} {}_2F_1(a+1, a+b; a, a+b+1; \lambda/2) \quad (37.7.6)$$

Let  $W = 1 - B = |E|/|T + E|$ . The noncentral moments of  $W$  are specified in Theorem 10.5.1 of Muirhead (1982) as

$$E(W^s) = \frac{\Gamma(n/2 + s)\Gamma((n+m)/2)}{\Gamma(n/2)\Gamma((n+m)/2 + s)} {}_1F_1\left(s; \frac{n+m}{2} + s; -\frac{1}{2}\lambda\right) \quad (37.7.7)$$

where  $\Gamma(\cdot)$  is the gamma function (see section ??) and  ${}_1F_1(\cdot)$  is the confluent hypergeometric function (see section 11.2.1). From this the moments of  $B$  can derived in the same way as the moments of  $R^2$  are derived from the moments of  $1 - R^2$  (see section 37.2.4.2).

### 37.7.5 Random Numbers

---

Function **NoncentralBetaDistRanBoost**(*Size* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function NoncentralBetaDistRanBoost returns random numbers following a noncentral Beta-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

Random numbers from a non-central Beta-distribution with integer or half-integer p- and q-values is easily obtained using the definition above i.e. by using a random number from a non-central chi-square distribution and another from a (central) chi-square distribution.

## 37.8 Noncentral Student's t-Distribution

### 37.8.1 Definition

If  $X$  is a random variable following a normal distribution with mean  $\delta$  and variance unity and  $\chi^2$  is a random variable following an independent  $\chi^2$ -distribution with  $n$  degrees of freedom, then the distribution of the ratio  $\frac{X}{\sqrt{\chi^2/n}}$  is called noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$ .

### 37.8.2 Density and CDF

---

Function **NoncentralTDistBoost**(*x* As mpNum, *n* As mpNum, *delta* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **NoncentralTDistBoost** returns pdf, CDF and related information for the noncentral *t*-distribution

**Parameters:**

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 37.8.2 and 37.8.2.3.

#### 37.8.2.1 Density: Closed form representation

The density of a variable following a noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$  is given by (Wang & Gray, 1993)

$$f_{t'}(n, x, \delta) = K_1 \left[ \frac{\sqrt{2}\delta x \cdot {}_1F_1(m + 1; \frac{3}{2}; y^2)}{a\Gamma(m + \frac{1}{2})} - \frac{{}_1F_1(m + \frac{1}{2}; \frac{1}{2}; y^2)}{\sqrt{a}\Gamma(m + 1)} \right] \quad (37.8.1)$$

$$K_1 = \frac{n^m \Gamma(n + 1) \exp(-\frac{1}{2}\delta^2)}{2^n a^m \Gamma(m)}; \quad m = \frac{n}{2}; \quad a = n + x^2 \quad y^2 = \frac{\delta^2 x^2}{2a}, \quad (37.8.2)$$

where  $\Gamma(\cdot)$  is the Gamma function (see section ??), and  ${}_1F_1(\cdot)$  is the confluent hypergeometric function (see section 11.2.1). For  $x = 0$  this simplifies to (Witkovsky, 2013):

$$f_{t'}(n, 0, \delta) = \exp(-\frac{1}{2}\delta^2) \frac{\Gamma(m + \frac{1}{2})}{\sqrt{\pi n} \Gamma(m)} \quad (37.8.3)$$

#### 37.8.2.2 Density: Integral representation

The pdf of a variable following a (singly) noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$  is given by (Witkovsky, 2013)

$$f_{t'}(n, x, \delta) = \int_0^\infty \phi \left( x \sqrt{\frac{q}{n}} - \delta \right) f_{\chi^2}(n, q) \sqrt{\frac{q}{n}} dq. \quad (37.8.4)$$

where  $\phi(\cdot)$  denotes the pdf of the normal distribution (see section 32.11.2.1) and  $f_{\chi^2}(n, \cdot)$  denotes the pdf of the (central)  $\chi^2$  distribution with  $n$  degrees of freedom (see section 32.4.2.1).

### 37.8.2.3 Density: Infinite series

The pdf of a variable following a noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$  is given by (Bristow *et al.*, 2013)

$$f_{t'}(n, x, \delta) = \frac{nt}{n^2 + 2nt^2 + t^4} + \frac{1}{2} \sum_{i=0}^{\infty} P_i I'_x \left( i + \frac{1}{2}, \frac{n}{2} \right) + \frac{\delta}{\sqrt{2}} Q_i I'_x \left( i + 1, \frac{n}{2} \right), \quad \text{and} \quad (37.8.5)$$

$I'_x(\cdot, \cdot)$  denotes the density of the beta distribution (see section 32.2.2.1), and  $P_i$  and  $Q_i$  are defined in equation 37.8.15.

### 37.8.2.4 CDF: Integral representation

The cdf of a variable following a noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$  is given by (Witkovsky, 2013)

$$\Pr[X \leq x] = F_{t'}(n, x, \delta) = \int_0^{\infty} \Phi \left( x \sqrt{\frac{q}{n}} - \delta \right) f_{\chi^2}(n, q) dq. \quad (37.8.6)$$

where  $\Phi(\cdot)$  denotes the cdf of the normal distribution (see section 32.11.2.2) and  $f_{\chi^2}(n, \cdot)$  denotes the pdf of the  $\chi^2$  distribution with  $n$  degrees of freedom (see section 32.4.2.1).

### 37.8.2.5 CDF: Finite series for integer degrees of freedom

Owen (1968) gives the following algorithm:

$$F_{t'}(n, x, \delta) = \Phi(-\delta\sqrt{B}) + 2(M_1 + M_3 + \dots + M_n) \quad \text{for odd degrees of freedom} \quad (37.8.7)$$

$$F_{t'}(n, x, \delta) = \Phi(-\delta) + \sqrt{2\pi}(M_2 + M_4 + \dots + M_n) \quad \text{for even degrees of freedom} \quad (37.8.8)$$

$$A = \frac{t}{\sqrt{n}}, \quad B = \frac{n}{n + t^2} \quad (37.8.9)$$

$$M_1 = T_{\text{Owen}}(\delta\sqrt{B}, A), \quad M_2 = A\sqrt{B}\phi(\delta\sqrt{B}) - \Phi(\delta A\sqrt{B}) \quad (37.8.10)$$

$$M_3 = B(\delta A M_2 + A\phi(\delta)/\sqrt{2\pi}), \quad M_4 = \frac{1}{2}B(\delta A M_3 + M_2) \quad (37.8.11)$$

$$M_k = \frac{k-3}{k-2}B(a_k \delta A M_{k-1} + M_{k-2}), \quad a_k = \frac{1}{(k-4)a_{k-1}} \quad \text{for } k \geq 5, \quad (37.8.12)$$

where  $T_{\text{Owen}}(\cdot, \cdot)$  denotes Owen's T function (see section 37.3.6), and  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the cdf (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively.

### 37.8.2.6 CDF: Infinite series

The cdf of a variable following a noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$  is given by (Benton & Krishnamoorthy, 2003; Bristow *et al.*, 2013)

$$F_{t'}(n, x, \delta) = \Phi(-\delta) + \frac{1}{2} \sum_{i=0}^{\infty} P_i I_x\left(i + \frac{1}{2}, \frac{n}{2}\right) + \frac{\delta}{\sqrt{2}} Q_i I_x\left(i + 1, \frac{n}{2}\right), \quad \text{and} \quad (37.8.13)$$

$$1 - F_{t'}(n, x, \delta) = \frac{1}{2} \sum_{i=0}^{\infty} P_i I_y\left(\frac{n}{2}, i + \frac{1}{2}\right) + \frac{\delta}{\sqrt{2}} Q_i I_y\left(\frac{n}{2}, i + 1\right), \quad \text{where} \quad (37.8.14)$$

$$\lambda = \frac{1}{2}\delta^2; \quad P_i = \frac{e^{-\lambda}\lambda^i}{i!}; \quad Q_i = \frac{e^{-\lambda}\lambda^i}{\Gamma(i + 3/2)}; \quad x = \frac{t^2}{n + t^2}; \quad y = 1 - x, \quad (37.8.15)$$

$I_x(\cdot, \cdot)$  denotes the (normalized) incomplete beta function (see section 32.2.2.2), and  $\Phi(\cdot)$  denotes the cdf of the normal distribution (see section 32.11.2.2).

### 37.8.2.7 CDF: Saddlepoint approximation

Broda & Paolella (2007) give the following saddlepoint approximation:

$$F_{t'}(y_1; n, \mu) \approx \Phi(w) + \phi(w) \left( \frac{1}{w} - \frac{d}{u} \right) \quad (37.8.16)$$

$$f_{t'}(y_1; n, \mu) \approx \phi(w)/u, \quad \text{where} \quad (37.8.17)$$

$$y_2 = [\mu y_1 + \sqrt{4n(y_1^2 + n) + \mu^2 y_1^2}] / (2y_1^2 + 2n),$$

$$t_1 = -\mu + y_1 y_2,$$

$$t_2 = -y_1 t_1 / (2ny_2),$$

$$d = 1/(t_1 y_2),$$

$$u = \sqrt{(\mu y_1 y_2 + 2n) / (2n)} / y_2,$$

$$w = \text{sgn}(y_1 - \mu) \sqrt{-\mu t_1 - 2n \ln(y_2)},$$

and  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the CDF (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively.

### 37.8.2.8 CDF: Edgeworth expansion

Akahira *et al.* (1995) gives the following approximation:

$$F_{t'}(n, x, \delta) \approx \Phi\left(\frac{\sqrt{k - 4ac} - \sqrt{k}}{2a}\right), \quad \text{where} \quad (37.8.18)$$

$$h = \frac{1}{24} \left( \frac{1}{n^2} + \frac{1}{4n^3} \right) \quad b = \sqrt{\frac{2}{n} \frac{\Gamma(\frac{1}{2}n + \frac{1}{2})}{\Gamma(\frac{1}{2}n)}} \quad (37.8.19)$$

$c = bt - a + \delta$ ,  $a = t^3 h / k$ ,  $k = 1 + (1 - b^2)t^2$ , and  $\Phi(\cdot)$  denotes the CDF of the normal distribution (see section 32.11.2.2).

### 37.8.3 Quantiles

---

Function **NoncentralTDistInvBoost**(*Prob* As mpNum, *n* As mpNum, *delta* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralTDistInvBoost returns quantiles and related information for the noncentral *t*-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below:

Let  $t_{n,\delta;\alpha}$  denote the  $\alpha$ -quantile of the noncentral *t*-distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$ . The quantile is calculated starting with one of the approximation below, followed by a Newton procedure using 37.8.2.7 until a relative error of  $10^{-4}$ . The final value is then obtained from this value by a Newton procedure using the "exact" formulas for the CDF.

#### 37.8.3.1 Quantiles: Akahira's approximation

Akahira *et al.* (1995) gives the following approximation:

$$t_{n,\delta;\alpha} \approx \frac{\delta b + u_1 \sqrt{b + c(\delta^2 - u_1^2)}}{b^2 - cu_1^2}, \quad \text{where} \quad (37.8.20)$$

$$u_1 = z_\alpha - \frac{at_1^3}{(1 + ct_1^2)^{3/2}}, \quad t_1 = \frac{\delta b + u_0 \sqrt{b + c(\delta^2 - u_0^2)}}{b^2 - cu_0^2}, \quad u_0 = z_\alpha - \frac{at_0^3}{(1 + ct_0^2)^{3/2}},$$

$t_0 = \delta + z_\alpha$ ,  $c = 1 - b^2$ ,  $g = c^{-1/2}$ ,  $a = (z_\alpha^2 - 1)h$ ,  $h$  and  $b$  are defined in equation (37.8.19), and  $z_\alpha$  denotes the  $\alpha$ -quantile of the normal distribution (see section 32.11.3). This approximation produces reliable results for  $\delta > 0$ , if either  $-bg < z_\alpha \leq -1$  or  $1 \leq z_\alpha < bg + |z_\alpha^2 - 1|hg$ .

#### 37.8.3.2 Quantiles: van Eeden's approximation

van Eeden (1961) gives the following approximation to the upper percentage point:

$$t_{n,\delta;\alpha} \approx z + \frac{z^3 + z}{4n} + \frac{5z^5 + 16z^3 + 3z}{96n^2} + \delta + \frac{\delta(2z^2 + 1) + \delta^2 z}{4n} + \delta \left( \frac{4z^4 + 12z^2 + 1}{32n^2} + \frac{\delta(z^3 + 4z + 4z)}{16n^2} - \frac{\delta^2(z^2 - 1)}{24n^2} - \frac{\delta^3 z}{32n^2} \right) \quad (37.8.21)$$

where  $z_\alpha$  denotes the  $\alpha$ -quantile of the normal distribution (see section 32.11.3). This approximation produces reliable results for  $\delta > 0$ , if  $z_\alpha \leq 1$ .

### 37.8.4 Properties

---

Function **NoncentralTDistInfoBoost**(*n* As mpNum, *delta* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralTDistInfoBoost returns moments and related information for the noncentral *t*-distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.8.4.1 Moments (singly noncentral)

The algebraic moments (defined for  $n > r$ ) are given by

$$\mu'_r = \left(\frac{1}{2}n\right)^{r/2} \frac{\Gamma\left(\frac{1}{2}(n-r)\right)}{\Gamma\left(\frac{1}{2}n\right)} \sum_{i=0}^{[r/2]_G} \binom{r}{2i} \frac{(2i)!}{2^i i!} \delta^{r-2i}. \quad (37.8.22)$$

The first four raw moments are given by

$$\mathbb{E}(t) = \delta \sqrt{\frac{1}{2}n} \frac{\Gamma\left(\frac{1}{2}(n-1)\right)}{\Gamma\left(\frac{1}{2}n\right)} \quad (37.8.23)$$

$$\mathbb{E}(t^2) = (\delta^2 + 1) \frac{n}{n-2} \quad (37.8.24)$$

$$\mathbb{E}(t^3) = \delta(\delta^2 + 3) \sqrt{\frac{1}{8}n^3} \frac{\Gamma\left(\frac{1}{2}(n-3)\right)}{\Gamma\left(\frac{1}{2}n\right)} \quad (37.8.25)$$

$$\mathbb{E}(t^4) = (\delta^4 + 6\delta^2 + 3) \frac{n^2}{(n-2)(n-4)} \quad (37.8.26)$$

#### 37.8.4.2 Recurrence relations

( ) The following relation holds (Witkovsky, 2013):

$$f_{t'}(n, x, \delta) = \frac{n}{x} \left[ F_{t'}\left(n+2, x\sqrt{1+2/n}, \delta\right) - F_{t'}(n, x, \delta) \right] \quad (37.8.27)$$

See Wang & Gray (1993) for additional recurrence relations for the density.

### 37.8.5 Random Numbers

---

Function **NoncentralTDistRanBoost**(*Size* As mpNum, *n* As mpNum, *delta* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function `NoncentralTDistRanBoost` returns random numbers following a noncentral  $t$ -distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below:

Random numbers from a non-central  $t$ -distribution is easily obtained using the definition in terms of the ratio between two independent random numbers from a normal and a central chi-square distribution. This ought to be sufficient for most applications but if needed more efficient techniques may easily be developed e.g. using more general techniques.

### 37.8.6 Confidence Limits for the Noncentrality Parameter

---

Function **NoncentralTDistNoncentrality**(*alpha* As `mpNum`, *delta* As `mpNum`, *theta* As `mpNum`, *n* As `mpNum`, *Output* As `String`) As `mpNumList`

---

**NOT YET IMPLEMENTED**

---

The function `NoncentralTDistNoncentrality` returns confidence limits for the doubly noncentrality parameter *delta* and related information for the noncentral  $t$ -distribution.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*delta*: A real number greater 0, representing the numerator noncentrality parameter

*theta*: A real number greater 0, representing the denominator noncentrality parameter

*n*: A real number greater 0, representing the degrees of freedom.

*Output*: A string describing the output choices

See section 32.1.3.5 for the options for *alpha*, *Noncentrality* and *Output*. Algorithms and formulas are given below.

Let  $T$  be a statistic according to the non-central  $t$ -distribution with  $n$  degrees of freedom and a non-centrality parameter  $\delta$ . Then the lower confidence limit  $\hat{\delta}$  of level  $1 - \alpha$  and the two-sided confidence interval  $[\underline{\delta}, \bar{\delta}]$  of the non-centrality parameter  $\delta$  of level  $1 - \alpha$  are given by [Akahira et al. \(1995\)](#):

$$\hat{\delta} = bT - z_\alpha \sqrt{k} + hT^3(z_\alpha^2 - 1)/k, \quad (37.8.28)$$

$$\underline{\delta} = bT - z_{\alpha/2} \sqrt{k} + hT^3(z_{\alpha/2}^2 - 1)/k, \quad (37.8.29)$$

$$\bar{\delta} = bT + z_{\alpha/2} \sqrt{k} - hT^3(z_{\alpha/2}^2 - 1)/k, \quad (37.8.30)$$

where  $k = 1 + (1 - b^2)T^2$ ,  $h$  and  $b$  are defined in equation (37.8.19), and  $z_\alpha$  denotes the  $\alpha$ -quantile of the normal distribution (see section 32.11.3).

### 37.8.7 Sample Size Function

---

Function **NoncentralTDistSampleSize**(*alpha* As mpNum, *beta* As mpNum, **ModifiedNoncentrality1** As mpNum, **ModifiedNoncentrality2** As mpNum, **Output** As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralTDistSampleSize returns sample size estimates and related information for the doubly noncentral *t*-distribution.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*beta*: A real number between 0 and 1, specifies the Type II error (or 1 - Power).

**ModifiedNoncentrality1**: A real number greater 0, representing the modified numerator noncentrality parameter.

**ModifiedNoncentrality2**: A real number greater 0, representing the modified denominator noncentrality parameter.

**Output**: A string describing the output choices

See section 32.1.3.4 for the options for *alpha*, *beta*, **ModifiedNoncentrality** and *Output*. Algorithms and formulas are given below.

We denote by  $N_{t'}(\alpha, \beta, \tilde{\rho})$  the sample size function of the noncentral *t*-distribution for a given confidence level  $\alpha$ , power  $\beta$  and modified noncentrality parameter  $\tilde{\rho}$ . This function determines the minimal sample size  $N$  for given noncentrality parameter, Type I error  $1 - \alpha$ , and Type II error  $1 - \beta$ , where  $\delta = \sqrt{N}\tilde{\rho}$

## 37.9 Doubly Noncentral Student's t-Distribution

### 37.9.1 Definition

Let  $T = X/\sqrt{Y/n}$ , where  $X$  and  $Y$  are independent,  $X$  has a normal distribution with mean  $\mu$  and unit variance, and  $Y$  has a noncentral  $\chi^2$  distribution with  $n$  degrees of freedom and noncentrality parameter  $\theta$ .

### 37.9.2 Density and CDF

---

Function **DoublyNoncentralTDist**(*x* As *mpNum*, *n* As *mpNum*, *delta* As *mpNum*, *theta* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **DoublyNoncentralTDist** returns pdf, CDF and related information for the doubly noncentral  $t$ -distribution

#### Parameters:

*x*: A real number

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the numerator noncentrality parameter

*theta*: A real number greater 0, representing the denominator noncentrality parameter

*Output*: A string describing the output choices

#### 37.9.2.1 Density (doubly noncentral): Integral representation

The pdf of a variable following a doubly noncentral  $t$ -distribution with  $n$  degrees of freedom and noncentrality parameters  $\delta$  and  $\theta$  is given by ([Witkovsky, 2013](#))

$$f_{t''}(t; n; \delta, \theta) = \int_0^\infty \phi\left(x\sqrt{\frac{q}{n}} - \delta\right) f_{\chi^2}(n, q; \theta) \sqrt{\frac{q}{n}} dq. \quad (37.9.1)$$

where  $\phi(\cdot)$  denotes the pdf of the normal distribution (see section [32.11.2.1](#)) and  $f_{\chi_n^2}(n, \cdot; \theta)$  denotes the pdf of the noncentral  $\chi^2$  distribution with  $n$  degrees of freedom and noncentrality parameter  $\theta$  (see section [37.6.2](#)).

#### 37.9.2.2 Density: Infinite series

[Kocherlakota & Kocherlakota \(1991\)](#) show that

$$f_{t''}(t; n; \mu, \theta) = \sum_{i=0}^{\infty} \omega_{i,\theta} s_{i,n} f_{t'}(s_{i,n} t; n + 2i, \mu), \quad \text{where} \quad (37.9.2)$$

$$\omega_{i,\theta} = \frac{\exp(-\theta/2)(\theta/2)^i}{i!} \quad \text{and} \quad s_{i,n} = \sqrt{\frac{n+2i}{n}} \quad (37.9.3)$$

and  $f_{t'}(\cdot)$  denotes the pdf of the singly noncentral  $t$ -distribution (see section [37.8.2](#)). Note that the terms  $f_{t'}(s_{i,n} t; n + 2i, \mu)$  can be calculated by using the recurrence relation in equation [\(37.8.4.2\)](#) once the first 2 terms have been calculated explicitly using any of the methods given in section [37.8.2](#).

### 37.9.2.3 Density (doubly noncentral): Saddlepoint approximation

Broda & Paoletta (2007) give the following saddlepoint approximation:

$$f_{t''}(t; n; \mu, \theta) \approx \phi(w)/u, \quad \text{where} \quad (37.9.4)$$

where  $w$  and  $u$  are defined in (37.9.9).

### 37.9.2.4 CDF: Integral representation

The cdf of a variable following a doubly noncentral t-distribution with  $n$  degrees of freedom and noncentrality parameters  $\delta$  and  $\theta$  is given by (Witkovsky, 2013)

$$\Pr[X \leq x] = F_{t''}(t; n; \delta, \theta) = \int_0^\infty \Phi\left(x\sqrt{\frac{q}{n}} - \delta\right) f_{\chi^2}(n, q; \theta) dq. \quad (37.9.5)$$

where  $\Phi(\cdot)$  denotes the cdf of the normal distribution (see section 32.11.2.2) and  $f_{\chi^2}(n, \cdot; \theta)$  denotes the pdf of the noncentral  $\chi^2$  distribution with  $n$  degrees of freedom and noncentrality parameter  $\theta$  (see section 37.6.2).

### 37.9.2.5 CDF: Infinite series

Kocherlakota & Kocherlakota (1991) show that

$$F_{t''}(t; n; \mu, \theta) = \sum_{i=0}^{\infty} \omega_{i,\theta} s_{i,n} F_{t'}(s_{i,n}t; n+2i, \mu), \quad \text{where} \quad (37.9.6)$$

where  $\omega_{i,\theta}$  and  $s_{i,n}$  are defined in (37.9.3) and  $F_{t'}(\cdot)$  denotes the cdf of the singly noncentral t-distribution (see section 37.8.2.3). Note that the terms  $F_{t'}(s_{i,n}t; n+2i, \mu)$  can be calculated by using the recurrence relation in equation (37.8.4.2) once the first 2 terms have been calculated explicitly using any of the methods given in section 37.8.2.3.

### 37.9.2.6 CDF: Approximation by singly noncentral t

Broda & Paoletta (2007) proposes the following approximation

$$F_{t''}(t; n; \mu, \theta) \approx F_{t'}(t/c; f; \mu), \quad \text{where} \quad (37.9.7)$$

$$f = \frac{7}{2} \left( -1 + \sqrt{15 - 7g^2/h} \right)^{-1}, \quad c = \sqrt{h(1 - 2/f)}, \quad g = m_1/\sqrt{\mu^2/2}, \quad h = m_2/(1 + \mu^2), \quad (37.9.8)$$

and  $m_i$  denotes the  $i$ th raw moment of  $t''$ , given in section 37.9.4.1.

### 37.9.2.7 CDF: Saddlepoint approximation CDF

Broda & Paoletta (2007) give the following saddlepoint approximation:

$$F_{t''}(y_1; n; \mu, \theta) \approx \Phi(w) + \phi(w) \left( \frac{1}{w} - \frac{d}{u} \right), \quad \text{where} \quad (37.9.9)$$

$$\begin{aligned}
a &= y_1^4 + 2ny_1^2 + n^2 \\
c_2 &= (-2y_1^3\mu - 2y_1n\mu)/a \\
c_1 &= (y_1^2\mu^2 - ny_1^2 - n^2 - \theta n)/a \\
c_0 &= (y_1n\mu)/a \\
q &= \frac{1}{3}c_1 - \frac{1}{9}c_2^2, \quad r = \frac{1}{6}(c_1c_2 - 3c_0) - \frac{1}{27}c_2^3 \\
y_2 &= \sqrt{-4q} \cos \left( \frac{1}{3} \arccos \left( r/\sqrt{-q^3} \right) \right) - \frac{1}{3}c_2 \\
t_1 &= -\mu + y_1y_2 \\
t_2 &= -y_1t_1/(2ny_2) \\
d &= 1/(t_1y_2) \\
\nu &= 1/(1 - 2t_2) \\
\alpha &= \mu/\sqrt{1 + \theta/n} \\
u &= \sqrt{(y_1^2 + 2nt_2)(2n\nu^2 + 4\theta\nu^3) + 4n^2y_2^2}/(2ny_2^2) \\
w &= \text{sgn}(y_1 - \alpha)\sqrt{-\mu t_1 - n \ln(\nu) - 2\theta\nu t_2}
\end{aligned}$$

, and  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the cdf (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively.

### 37.9.3 Quantiles

---

Function **DoublyNoncentralTDistInv**(*Prob* As mpNum, *n* As mpNum, *delta* As mpNum, *theta* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **DoublyNoncentralTDistInv** returns quantiles and related information for the doubly noncentral *t*-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the numerator noncentrality parameter

*theta*: A real number greater 0, representing the denominator noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below:

The quantiles are calculated by a Newton iteration, using an first approximation by the singly noncentral distribution as initial estimate in an iteration based on equation 37.9.9, until a relative error of  $10^{-4}$  has been achieved. This new approximation is then used as initial estimate for a Newton iteration based on equation 37.9.6.

### 37.9.4 Properties

---

Function **DoublyNoncentralTDistInfo**(*n* As mpNum, *delta* As mpNum, *theta* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **DoublyNoncentralTDistInfo** returns moments and related information for the doubly noncentral *t*-distribution

**Parameters:**

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the numerator noncentrality parameter

*theta*: A real number greater 0, representing the denominator noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.9.4.1 Moments

The algebraic moments (defined for  $n > r$ ) are given by

$$\mu'_r = \left(\frac{1}{2}n\right)^{r/2} \frac{\Gamma\left(\frac{1}{2}(n-r)\right)}{\Gamma\left(\frac{1}{2}n\right)} \times \sum_{i=0}^{\lfloor r/2 \rfloor} \binom{r}{2i} \frac{(2i)!}{2^i i!} \delta^{r-2i} \times e^{-\lambda/2} M\left(\frac{1}{2}(n-r), \frac{1}{2}n, \frac{1}{2}\lambda\right). \quad (37.9.10)$$

The first four raw moments are given by

$$\mathbb{E}(t) = \delta \sqrt{\frac{1}{2}n} \frac{\Gamma\left(\frac{1}{2}(n-1)\right)}{\Gamma\left(\frac{1}{2}n\right)} e^{-\lambda/2} M\left(\frac{1}{2}(n-1), \frac{1}{2}n, \frac{1}{2}\lambda\right) \quad (37.9.11)$$

$$\mathbb{E}(t^2) = (\delta^2 + 1) \frac{n}{n-2} e^{-\lambda/2} M\left(\frac{1}{2}(n-2), \frac{1}{2}n, \frac{1}{2}\lambda\right) \quad (37.9.12)$$

$$\mathbb{E}(t^3) = \delta(\delta^2 + 3) \sqrt{\frac{1}{8}n^3} \frac{\Gamma\left(\frac{1}{2}(n-3)\right)}{\Gamma\left(\frac{1}{2}n\right)} e^{-\lambda/2} M\left(\frac{1}{2}(n-3), \frac{1}{2}n, \frac{1}{2}\lambda\right) \quad (37.9.13)$$

$$\mathbb{E}(t^4) = (\delta^4 + 6\delta^2 + 3) \frac{n^2}{(n-2)(n-4)} e^{-\lambda/2} M\left(\frac{1}{2}(n-4), \frac{1}{2}n, \frac{1}{2}\lambda\right) \quad (37.9.14)$$

### 37.9.5 Random Numbers

---

Function **DoublyNoncentralTDistRan**(*Size* As mpNum, *n* As mpNum, *delta* As mpNum, *theta* As mpNum, *Generator* As String, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **DoublyNoncentralTDistRan** returns random numbers following a doubly noncentral *t*-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*n*: A real number greater 0, representing the degrees of freedom

*delta*: A real number greater 0, representing the numerator noncentrality parameter

*theta*: A real number greater 0, representing the denominator noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below:

Random numbers from a doubly non-central t-distribution is easily obtained using the definition in terms of the ratio between two independent random numbers from a normal and a non-central chi-square distribution. This ought to be sufficient for most applications but if needed more efficient techniques may easily be developed e.g. using more general techniques.

### 37.9.6 Confidence Limits for the Noncentrality Parameter

---

Function **DoublyNoncentralTDistNoncentrality**(*alpha* As mpNum, *delta* As mpNum, *theta* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **DoublyNoncentralTDistNoncentrality** returns confidence limits for the doubly noncentrality parameter *delta* and related information for the noncentral *t*-distribution.

#### Parameters:

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*delta*: A real number greater 0, representing the numerator noncentrality parameter

*theta*: A real number greater 0, representing the denominator noncentrality parameter

*n*: A real number greater 0, representing the degrees of freedom.

*Output*: A string describing the output choices

See section 32.1.3.5 for the options for *alpha*, *Noncentrality* and *Output*. Algorithms and formulas are given below.

Let  $T$  be a statistic according to the non-central  $t$ -distribution with  $n$  degrees of freedom and a non-centrality parameter  $\delta$ . Then the lower confidence limit  $\hat{\delta}$  of level  $1 - \alpha$  and the two-sided confidence interval  $[\underline{\delta}, \bar{\delta}]$  of the non-centrality parameter  $\delta$  of level  $1 - \alpha$  are given by [Akahira et al. \(1995\)](#):

$$\hat{\delta} = bT - z_\alpha \sqrt{k} + hT^3(z_\alpha^2 - 1)/k, \quad (37.9.15)$$

$$\underline{\delta} = bT - z_{\alpha/2} \sqrt{k} + hT^3(z_{\alpha/2}^2 - 1)/k, \quad (37.9.16)$$

$$\bar{\delta} = bT + z_{\alpha/2} \sqrt{k} - hT^3(z_{\alpha/2}^2 - 1)/k, \quad (37.9.17)$$

where  $k = 1 + (1 - b^2)T^2$ ,  $h$  and  $b$  are defined in equation (37.8.19), and  $z_\alpha$  denotes the  $\alpha$ -quantile of the normal distribution (see section 32.11.3).

### 37.9.7 Sample Size Function

---

Function **DoublyNoncentralTDistSampleSize**(*alpha* As mpNum, *beta* As mpNum, **ModifiedNoncentrality1** As mpNum, **ModifiedNoncentrality2** As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function `DoublyNoncentralTDistSampleSize` returns sample size estimates and related information for the doubly noncentral  $t$ -distribution.

**Parameters:**

`alpha`: A real number between 0 and 1, specifies the confidence level (or Type I error).

`beta`: A real number between 0 and 1, specifies the Type II error (or 1 - Power).

`ModifiedNoncentrality1`: A real number greater 0, representing the modified numerator noncentrality parameter.

`ModifiedNoncentrality2`: A real number greater 0, representing the modified denominator noncentrality parameter.

`Output`: A string describing the output choices

See section 32.1.3.4 for the options for `alpha`, `beta`, `ModifiedNoncentrality` and `Output`. Algorithms and formulas are given below.

We denote by  $N_t'(\alpha, \beta, \tilde{\rho})$  the sample size function of the noncentral  $t$ -distribution for a given confidence level  $\alpha$ , power  $\beta$  and modified noncentrality parameter  $\tilde{\rho}$ . This function determines the minimal sample size  $N$  for given noncentrality parameter, Type I error  $1 - \alpha$ , and Type II error  $1 - \beta$ , where  $\delta = \sqrt{N}\tilde{\rho}$

## 37.10 NonCentral F-Distribution

### 37.10.1 Definition

If  $X_1$  and  $X_2$  are independent random variables,  $X_1$  following a non-central  $\chi^2$ -distribution with noncentrality parameter  $\lambda$  and  $m$  degrees of freedom, and  $X_2$  following a  $\chi^2$ -distribution with  $n$  degrees of freedom, then the distribution of the ratio  $F = \frac{X_1/m}{X_2/n}$  is said to follow a non-central F-distribution with noncentrality parameter  $\lambda$  and  $m$  and  $n$  degrees of freedom.

### 37.10.2 Density and CDF

---

Function **NoncentralFDistBoost**(*x* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function NoncentralFDistBoost returns pdf, CDF and related information for the noncentral F-distribution

#### Parameters:

*x*: A real number

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

#### 37.10.2.1 Density

#### 37.10.2.2 Density: Closed form representation

The density function of the noncentral F-Distribution is given by (Wang & Gray, 1993)

$$f_{F'}(x; n_1, n_2, \lambda) = e^{-\lambda/2} f_F(x; n_1, n_2) {}_1F_1\left(\frac{1}{2}(m+n), \frac{1}{2}n, \frac{nx\lambda}{2(m+nx)}\right) \quad (37.10.1)$$

where  $f_F(x; m, n)$  is the pdf of the (central) F distribution (see section 32.6.2.1) and  ${}_1F_1(\cdot)$  is the confluent hypergeometric function (see section 11.2.1).

#### 37.10.2.3 CDF: Integral representation

The cdf of a variable following a (singly) noncentral F-distribution with  $n$  and  $m$  degrees of freedom and noncentrality parameter  $\lambda_1$  and is given by (Chou, 1985)

$$\begin{aligned} \Pr[X \leq c] &= F_{F'}(x; m, n; \lambda_1) = \int_0^\infty F_{\chi^2}(n, ncy/m; \lambda_1) f_{\chi^2}(n, y) dy \\ &= \int_0^\infty [1 - F_{\chi^2}(m, mx/(nc))] f_{\chi^2}(n, x; \lambda_1) dx \\ &= 1 - \frac{cn}{m} \int_0^\infty F_{\chi^2}(m, y; \lambda_1) f_{\chi^2}(n, ncy/m) dy \end{aligned} \quad (37.10.2)$$

where  $F_{\chi^2}(n, \cdot; \lambda_1)$  denotes the cdf of the noncentral  $\chi^2$  distribution with  $n$  degrees of freedom and noncentrality parameter  $\lambda_1$  (see section 37.6.2.3) and  $f_{\chi^2}(m, \cdot; \theta)$  denotes the pdf of the noncentral  $\chi^2$  distribution with  $m$  degrees of freedom and noncentrality parameter  $\lambda_2$  (see section 37.6.2).

### 37.10.2.4 CDF: Finite Series for even error degrees of freedom

This is code for noncentral beta and needs to be adapted for noncentral F. The cdf can be calculated using the following finite series, if  $b$  is an integer:

$$I(x; a, b, \lambda) = e^{-\lambda(1-x)} \sum_{n=0}^{b-1} L_n, \quad \text{where} \quad (37.10.3)$$

$$L_0 = 1, \quad L_1 = (1-x)(a + \lambda x),$$

$$L_n = \frac{1-x}{n} ((2n-2+a+\lambda x)L_{n-1} - (n+a-2)(1-x)L_{n-2})$$

### 37.10.2.5 CDF: Infinite Series

The cdf of a variable following a (singly) noncentral F-distribution with  $n$  and  $m$  degrees of freedom and noncentrality parameter  $\lambda_1$  and is given by

$$\Pr[F \leq x] = F_{F'}(x; m, n, \lambda) = e^{-\lambda} \sum_{j=0}^{\infty} \frac{(\lambda/2)^j}{j!} F(m+2j, n, x) \quad (37.10.4)$$

where  $F_F(\cdot)$  denotes the cdf of the central F-distribution (see section 32.6.2.2).

### 37.10.2.6 CDF: 2-moment approximation by central F

Patnaik (1949) suggests the following approximation (see also Tiku (1966)):

$$F_{F'}(x; n_1, n_2, \lambda) \approx F_F(y; m_1, n_2), \quad \text{where} \quad (37.10.5)$$

$$A_1 = (n_1 + \lambda),$$

$$B_1 = (n_1 + 2\lambda),$$

$$m_1 = A_1^2/B_1,$$

$$y = n_1/A_1,$$

and  $F_F(\cdot; m_1, n_2)$  denotes the CDF of a central F distribution with  $m_1$  and  $n_2$  degrees of freedom (see section 32.6.2.2).

### 37.10.2.7 CDF: Saddlepoint approximation

Butler & Paolella (2002) suggest the following second order saddlepoint approximation:

$$F_{F'}(x; n_1, n_2, \lambda) \approx \Phi(w) + \phi(w) \left( \frac{1}{w} - \frac{1}{u} \left( 1 + \frac{\kappa_4}{8} - \frac{5}{24} \kappa_3^2 \right) - \frac{1}{u^3} - \frac{\kappa_3}{2u^2} + \frac{1}{w^3} \right) \quad (37.10.6)$$

$$f_{F'}(x; n_1, n_2, \lambda) \approx \frac{1}{\sqrt{2\pi\kappa_2}} \exp(k - sx) \left( 1 + \frac{\kappa_4}{8} - \frac{5}{24} \kappa_3^2 \right), \quad \text{where} \quad (37.10.7)$$

$$a_1 = 4n_2x(n_1 + n_2)$$

$$a_2 = x^2n_1^3 + 2x^2n_1^2\lambda + 2n_1^2xn_2 + 4x^2n_1n_2\lambda + n_1\lambda^2x^2 + 2n_1\lambda xn_2 + n_2^2n_1 + 4xn_2^2\lambda$$

$$s = [xn_1(n_1 + 2n_2 + \lambda) - n_1n_2 - \sqrt{n_1a_2}]/a_1$$

$$l_1 = n_2/n_1, \quad l_2 = -x$$

$$v_1 = 1/(1 - 2sl_1), v_2 = 1/(1 - 2sl_2)$$

$$g_1 = l_1 v_1, g_2 = l_2 v_2$$

$$k = \frac{1}{2}(n_1 \ln(v_1) + n_2 \ln(v_2)) + s\lambda g_1$$

$$k_2 = 2(g_1^2(n_1 + 2\lambda v_1) + g_2^2 n_2)$$

$$k_3 = 8(g_1^3(n_1 + 3\lambda v_1) + g_2^3 n_2)$$

$$k_4 = 48(g_1^4(n_1 + 4\lambda v_1) + g_2^4 n_2)$$

$$\kappa_3 = k_3/k_2^{3/2}, \kappa_4 = k_4/k_2^2$$

$$w = \text{sgn}(s)\sqrt{2(sx - k)}$$

$$u = s\sqrt{k_2}$$

and  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the cdf (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively.

### 37.10.3 Quantiles

---

Function **NoncentralFDistInvBoost**(*Prob* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function NoncentralFDistInvBoost returns quantiles and related information for the the non-central *F*-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

#### 37.10.3.1 Quantiles: Central F approximation

An approximation to  $F_{\alpha, n_1, n_2, \lambda}$ , the  $\alpha$ -quantile of a non-central F-distribution with  $n_1$  and  $n_2$  degrees of freedom and noncentrality parameter  $\lambda$ , is obtained as

$$F_{\alpha, n_1, n_2, \lambda} \approx c \cdot F_{\alpha, m_1, n_2}, \quad (37.10.8)$$

$$A_1 = (n_1 + \lambda),$$

$$B_1 = (n_1 + 2\lambda),$$

$$m_1 = A_1^2/B_1,$$

$$c = A_1/n_1,$$

and  $F_{\alpha, m_1, n_2}$ , denotes the  $\alpha$ -quantile of a central  $F$ -distribution with  $m_1$  and  $n_2$  degrees of freedom (see section ??).

See also [Mudholkar et al. \(1975\)](#)

### 37.10.4 Properties

---

Function **NoncentralFDistInfoBoost**(*m* As mpNum, *n* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function NoncentralFDistInfoBoost returns moments and related information for the noncentral  $F$ -distribution

**Parameters:**

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Output*: A string describing the output choices

See section [32.1.3.3](#) for the options for *Output*. Algorithms and formulas are given below.

#### 37.10.4.1 Moments

The algebraic moments (defined for  $f_2 > 2r$ ) are given by

$$\mu'_r = \frac{\Gamma(\frac{1}{2}f_1 + r) - \Gamma(\frac{1}{2}f_2 - r)}{\Gamma(\frac{1}{2}f_2)} \sum_{j=0}^r \binom{r}{j} \frac{\frac{1}{2}\lambda f_1)^j}{\Gamma(\frac{1}{2}f_1 + j)}, \quad \text{for } f_2 > 2r. \quad (37.10.9)$$

The first four raw moments (defined for  $n > 2k$ ) are given by

$$\begin{aligned} \mu'_1 &= \frac{n}{m} \frac{m + \lambda}{n - 2} \\ \mu'_2 &= \left(\frac{n}{m}\right)^2 \frac{\lambda^2 + (2\lambda + m)(m + 2)}{(n - 2)(n - 4)} \\ \mu'_3 &= \left(\frac{n}{m}\right)^3 \frac{\lambda^3 + 3(m + 4)\lambda^2 + (3\lambda + m)(m + 4)(m + 2)}{(n - 2)(n - 4)(n - 6)} \\ \mu'_4 &= \left(\frac{n}{m}\right)^4 \frac{\lambda^3 + 4(m + 6)\lambda^3 + 6(m + 6)(m + 4)\lambda^2 + (4\lambda + m)(m + 6)(m + 4)(m + 2)}{(n - 2)(n - 4)(n - 6)(n - 8)} \end{aligned}$$

#### 37.10.4.2 Recurrence relations

Let the density  $g_{m,n}$  be that of  $m/n$  times an  $F_{m,n}$  random variable. Let  $G_{m,n}(y)$  be its distribution function, and let  $g_{m,n}^\lambda$  and  $G_{m,n}^\lambda(y)$  be the density and distribution function of its (singly) noncentral version (the distribution of  $\chi_m^2(\lambda)/\chi_n^2(0)$ ). Then the following recurrence relations hold ([Chattamvelli & Jones, 1995](#))

$$n \left[ G_{m,n}^\lambda(y) - G_{m,n}^\lambda(y) \right] = -2g_{m,n}^\lambda(y) \quad (37.10.10)$$

$$\lambda(1 + y)g_{m+4,n}^\lambda(y) = [\lambda y - m(1 + y)]g_{m+2,n}^\lambda(y) + y(m + n)g_{m,n}^\lambda(y). \quad (37.10.11)$$

$$n(1+y)g_{m,n+2}^\lambda(y) = (m+n)g_{m,n}^\lambda(y) + \lambda g_{m+2,n}^\lambda(y). \quad (37.10.12)$$

$$\lambda g_{m+4,n-2}^\lambda(y) + mg_{m+2,n-2}^\lambda(y) = (n-2)yg_{m,n}^\lambda(y). \quad (37.10.13)$$

From equations 37.10.10 to 37.10.13 we obtain

$$\begin{aligned} \lambda(1+y)G_{m+6,n}^\lambda(y) &= [\lambda y - (m+2-\lambda)(1+y)]G_{m+4,n}^\lambda(y) \\ &\quad + [(m+2)(1+y) + y(m+n-\lambda)]G_{m+2,n}^\lambda(y) \\ &\quad - y(m+n)G_{m,n}^\lambda(y) \end{aligned} \quad (37.10.14)$$

$$\begin{aligned} n(1+y)[G_{m,n+2}^\lambda(y) - G_{m+2,n+2}^\lambda(y)] &= (m+n)G_{m,n}^\lambda(y) \\ &\quad + (\lambda - m - n)G_{m+2,n}^\lambda(y) \\ &\quad - \lambda G_{m+4,n}^\lambda(y) \end{aligned} \quad (37.10.15)$$

$$\begin{aligned} (n-2)y[G_{m,n}^\lambda(y) - G_{m+2,n}^\lambda(y)] &= (m+2)G_{m+2,n-2}^\lambda(y) \\ &\quad + (\lambda - m - 2)G_{m+4,n-2}^\lambda(y) \\ &\quad - \lambda G_{m+6,n-2}^\lambda(y) \end{aligned} \quad (37.10.16)$$

See sections ?? for the corresponding expressions for the central F distribution.

#### 37.10.4.3 Relationships to other distributions

$$F_{F'}(x; m, n, \lambda) = F_B\left(\frac{1}{2}n, \frac{1}{2}m, \frac{mx}{mx+n}; \lambda\right) \quad (37.10.17)$$

#### 37.10.5 Random Numbers

---

Function **NoncentralFDistRanBoost**(*Size* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function NoncentralFDistRanBoost returns random numbers following a noncentral *F*-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda*: A real number greater 0, representing the noncentrality parameter

*Generator*: A string describing the random generator

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

Random numbers from a non-central F-distribution is easily obtained using the definition in terms of the ratio between two independent random numbers from central and non-central chi-square distributions. This ought to be sufficient for most applications but if needed more efficient techniques may easily be developed e.g. using more general techniques.

### 37.10.6 Confidence Limits for the Noncentrality Parameter

---

Function **NoncentralFDistNoncentralityEx**(*alpha* As mpNum, *lambda1* As mpNum, *lambda2* As mpNum, *m* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralFDistNoncentralityEx returns confidence limits for the noncentrality parameter lambda and related information for the noncentral *F*-distribution.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).  
*lambda1*: A real number greater 0, representing the numerator noncentrality parameter  
*lambda2*: A real number greater 0, representing the denominator noncentrality parameter  
*m*: A real number greater 0, representing the numerator degrees of freedom.  
*n*: A real number greater 0, representing the denominator degrees of freedom.  
*Output*: A string describing the output choices

See section 32.1.3.5 for the options for *alpha*, *Noncentrality* and *Output*. Algorithms and formulas are given below.

### 37.10.7 Sample Size

---

Function **NoncentralFDistSampleSizeEx**(*alpha* As mpNum, *beta* As mpNum, *m* As mpNum, *ModifiedNoncentrality1* As mpNum, *ModifiedNoncentrality1* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralFDistSampleSizeEx returns sample size estimates and related information for the noncentral *F*-distribution.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).  
*beta*: A real number between 0 and 1, specifies the Type II error (or 1 - Power).  
*m*: A real number greater 0, representing the numerator degrees of freedom.  
*ModifiedNoncentrality1*: A real number greater 0, representing the modified numerator noncentrality parameter.  
*ModifiedNoncentrality1*: A real number greater 0, representing the modified denominator noncentrality parameter.  
*Output*: A string describing the output choices

See section 32.1.3.4 for the options for *alpha*, *beta*, *ModifiedNoncentrality* and *Output*. Algorithms and formulas are given below.

## 37.11 Doubly NonCentral F-Distribution

### 37.11.1 Definition

The doubly noncentral F distribution is determined as

$$F^{(2)} = \frac{U_1/n_1}{U_2/n_2} \quad (37.11.1)$$

where  $U_1$  and  $U_2$  are independent with  $U_i = \chi^2(n_i, \theta_i)$ , where  $n_i$  is the degrees of freedom, and  $\theta_i$  is the noncentrality parameter of the noncentral  $\chi^2$  distribution. We denote the doubly noncentral distribution as  $F^{(2)} = F_{n_1, n_2}(\theta_1, \theta_2)$ . Taking  $\theta_2 = 0$  gives the singly noncentral F and the additional requirement  $\theta_1 = 0$  leads to the central F.

See [Paoletta \(2007\)](#) and [Paoletta \(2006\)](#).

### 37.11.2 Density and CDF

---

Function **DoublyNoncentralFDistEx**(*x* As mpNum, *m* As mpNum, *n* As mpNum, *lambda1* As mpNum, *lambda2* As mpNum, *Output* As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function DoublyNoncentralFDistEx returns pdf, CDF and related information for the noncentral F-distribution

#### Parameters:

*x*: A real number

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda1*: A real number greater 0, representing the numerator noncentrality parameter

*lambda2*: A real number greater 0, representing the denominator noncentrality parameter

*Output*: A string describing the output choices

#### 37.11.2.1 Density: Infinite series

[Butler & Paoletta \(2002\)](#) show that

$$f_{F''}(t; n; \mu, \theta) = \sum_{i=0}^{\infty} \omega_{i,\theta} s_{i,n} f_{F'}(s_{i,n} t; n + 2i, \mu), \quad \text{where} \quad (37.11.2)$$

$$\omega_{i,\theta} = \frac{\exp(-\theta/2)(\theta/2)^i}{i!} \quad \text{and} \quad s_{i,n} = \sqrt{\frac{n+2i}{n}} \quad (37.11.3)$$

and  $f_{F'}(\cdot)$  denotes the pdf of the singly noncentral F-distribution (see section [37.10.2.1](#) ). Note that the terms  $f_{F'}(s_{i,n} x; n + 2i, \mu)$  can be calculated by using the recurrence relation in equation [\(37.10.11\)](#) once the first 2 terms have been calculated explicitly using any of the methods given in section [37.10.2.1](#).

### 37.11.2.2 CDF: Integral representation

The cdf of a variable following a doubly noncentral F-distribution with  $n$  and  $m$  degrees of freedom and noncentrality parameters  $\lambda_1$  and  $\lambda_2$  is given by (Chou, 1985)

$$\begin{aligned}\Pr[X \leq c] &= F_{F''}(x; m, n; \lambda_1, \lambda_2) = \int_0^\infty F_{\chi^2}(n, ncy/m; \lambda_1) f_{\chi^2}(n, y; \lambda_2) dy \quad (37.11.4) \\ &= \int_0^\infty [1 - F_{\chi^2}(m, mx/(nc); \lambda_2)] f_{\chi^2}(n, x; \lambda_1) dx \\ &= 1 - \frac{cn}{m} \int_0^\infty F_{\chi^2}(m, y; \lambda_1) f_{\chi^2}(n, ncy/m; \lambda_2) dy\end{aligned}$$

where  $F_{\chi^2}(n, \cdot; \lambda_1)$  denotes the cdf of the noncentral  $\chi^2$  distribution with  $n$  degrees of freedom and noncentrality parameter  $\lambda_1$  (see section 37.6.2.3) and  $f_{\chi^2}(m, \cdot; \theta)$  denotes the pdf of the noncentral  $\chi^2$  distribution with  $m$  degrees of freedom and noncentrality parameter  $\lambda_2$  (see section 37.6.2).

### 37.11.2.3 CDF: Infinite series

Butler & Paoletta (2002) show that

$$F_{F''}(x; m, n; \lambda_1, \lambda_2) = \sum_{i=0}^{\infty} \omega_{i,\theta} s_{i,n} F_{F'}(s_{i,n}t; m + 2i, n; \lambda_1), \quad \text{where} \quad (37.11.5)$$

where  $\omega_{i,\lambda_1}$  and  $s_{i,n}$  are defined in (37.11.3) and  $F_{F'}(\cdot)$  denotes the cdf of the singly noncentral F-distribution (see section 37.10.2). Note that the terms  $F_{F'}(s_{i,n}t; n + 2i, \mu)$  can be calculated by using the recurrence relation in equation (37.10.14) once the first 3 terms have been calculated explicitly using any of the methods given in section 37.10.2.

### 37.11.2.4 CDF: Central F approximation

Mudholkar *et al.* (1976) suggests an approximation by noncentral  $F$ , which can be converted into an approximation by central  $F$  as follows:

$$\begin{aligned}F_{F'}(x; n_1, n_2, \lambda_1, \lambda_2) &\approx F_F(y; m_1, m_2), \quad \text{where} \quad (37.11.6) \\ A_1 &= (n_1 + \lambda_1), A_2 = (n_1 + \lambda_2), \\ B_1 &= (n_1 + 2\lambda_1), B_2 = (n_1 + 2\lambda_2), \\ m_1 &= A_1^2/B_1, m_2 = A_2^2/B_2, \\ y &= x(n_1 A_2)/(n_2 A_1),\end{aligned}$$

and  $F_F(\cdot; m_1, m_2)$  denotes the CDF of a central  $F$  distribution with  $m_1$  and  $m_2$  degrees of freedom (see section 32.6.2.2).

### 37.11.2.5 CDF: Edgeworth expansion

Mudholkar *et al.* (1976) suggest an Edgeworth-series expansion, which can be converted into an Cornish-Fisher expansion as follows:

$$F_{F'}(x; n_1, n_2, \lambda_1, \lambda_2) \approx \Phi(z), \quad \text{where} \quad (37.11.7)$$

$$u = \frac{-\kappa_1}{\sqrt{\kappa_2}}, \quad z = u + (u^2 - 1) \frac{\gamma_1}{6} + (u^3 - 3u) \frac{\gamma_2}{24} + (4u^3 - 7u) \frac{\gamma_1^2}{36},$$

$$\kappa_i = T_i(1, n_1, \lambda_1) + (-1)^i T_i(x, n_2, \lambda_2), \quad i = 1, 2, 3, 4,$$

$$\gamma_1 = \frac{\kappa_3}{\sqrt{\kappa_2} \kappa_2}, \quad \gamma_2 = \frac{\kappa_4}{\kappa_2^2},$$

$T_i(\cdot)$  is defined in section 37.6.4, and  $\Phi(\cdot)$  denotes the CDF of the normal distribution (see section 32.11.2.2).

### 37.11.2.6 CDF and pdf: Saddlepoint approximation

Butler & Paolella (2002) suggest the following second order saddlepoint approximation::

$$F_{F''}(x; n_1, n_2, \lambda_1, \lambda_2) \approx \Phi(w) + \phi(w) \left( \frac{1}{w} - \frac{1}{u} \left( 1 + \frac{\kappa_4}{8} - \frac{5}{24} \kappa_3^2 \right) - \frac{1}{u^3} - \frac{\kappa_3}{2u^2} + \frac{1}{w^3} \right) \quad (37.11.8)$$

$$f_{F''}(x; n_1, n_2, \lambda_1, \lambda_2) \approx \frac{1}{\sqrt{2\pi k_2}} \exp(k - sx) \left( 1 + \frac{\kappa_4}{8} - \frac{5}{24} \kappa_3^2 \right), \quad \text{where} \quad (37.11.9)$$

$$a = 8x^2 n_2^2 (n_1 + n_2)$$

$$a_0 = (x \lambda_2 n_1^2 - (1 - x) n_1^2 n_2 - n_1 n_2 \lambda_1) / a$$

$$a_1 = (2(n_2^2 n_1 + n_1^2 n_2 x^2) - 4x n_1 n_2 (n_1 + n_2 + \lambda_1 + \lambda_2)) / a$$

$$a_2 = (8x(1 - x)n_1 n_2^2 + 4x(n_2^3 + \lambda_2 n_2^2 - n_1^2 n_2 x - n_1 n_2 \lambda_1 x)) / (3a)$$

$$p = \sqrt{|3a_2^2 - a_1| / 3}$$

$$q = a_2(2a_2^2 - a_1) + a_0$$

$$s = -2p \cos \left( \frac{1}{3} \arccos(-\frac{1}{2}qp^{-3}) + \frac{1}{3}\pi \right) - a_2$$

$$l_1 = n_2 / n_1, \quad l_2 = -x$$

$$v_1 = 1 / (1 - 2s l_1), \quad v_2 = 1 / (1 - 2s l_2)$$

$$g_1 = l_1 v_1, \quad g_2 = l_2 v_2$$

$$K(s) = \frac{1}{2}(n_1 \ln(v_1) + n_2 \ln(v_2)) + s(\lambda_1 g_1 + \lambda_2 g_2)$$

$$K''(s) = 2(g_1^2(n_1 + 2\lambda_1 v_1) + g_2^2(n_2 + 2\lambda_2 v_2))$$

$$K^{(3)}(s) = 8(g_1^3(n_1 + 3\lambda_1 v_1) + g_2^3(n_2 + 3\lambda_2 v_2))$$

$$K^{(4)}(s) = 48(g_1^4(n_1 + 4\lambda_1 v_1) + g_2^4(n_2 + 4\lambda_2 v_2))$$

$$\kappa_3 = K^{(3)}(s) / K''(s)^{3/2}, \quad \kappa_4 = K^{(4)}(s) / K''(s)^2$$

$$w = \text{sgn}(s) \sqrt{2(sx - K(s))}$$

$$u = s \sqrt{K''(s)}$$

and  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the cdf (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively.

### 37.11.3 Quantiles

---

Function **DoublyNoncentralFDistInvEx**(*Prob* As mpNum, *m* As mpNum, *n* As mpNum, *lambda1* As mpNum, *lambda2* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **DoublyNoncentralFDistInvEx** returns quantiles and related information for the the noncentral *F*-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda1*: A real number greater 0, representing the numerator noncentrality parameter

*lambda2*: A real number greater 0, representing the denominator noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

#### 37.11.3.1 Quantiles: Central F approximation

An approximation to  $F_{\alpha, n_1, n_2, \lambda_1, \lambda_2}$ , the  $\alpha$ -quantile of a doubly non-central *F*-distribution with  $n_1$  and  $n_2$  degress of freedom and noncentrality parameters  $\lambda_1$  and  $\lambda_2$ , is obtained as

$$F_{\alpha, n_1, n_2, \lambda_1, \lambda_2} \approx c \cdot F_{\alpha, m_1, m_2}, \quad (37.11.10)$$

$$A_1 = (n_1 + \lambda_1), A_2 = (n_1 + \lambda_2),$$

$$B_1 = (n_1 + 2\lambda_1), B_2 = (n_1 + 2\lambda_2),$$

$$m_1 = A_1^2/B_1, m_2 = A_2^2/B_2,$$

$$c = (n_2 A_1) / (n_1 A_2),$$

and denotes the  $\alpha$ -quantile of a central *F*-distribution with  $m_1$  and  $m_2$  degress of freedom (see section ??).

### 37.11.4 Properties

---

Function **DoublyNoncentralFDistInfoEx**(*m* As mpNum, *n* As mpNum, *lambda1* As mpNum, *lambda2* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **DoublyNoncentralFDistInfoEx** returns moments and related information for the non-central *F*-distribution

**Parameters:**

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda1*: A real number greater 0, representing the numerator noncentrality parameter

*lambda2*: A real number greater 0, representing the denominator noncentrality parameter

*Output:* A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

### 37.11.4.1 Moments

The algebraic moments (defined for  $f_2 > 2r$ ) are given by

$$\mu'_r = \frac{\Gamma(\frac{1}{2}f_1 + r) - \Gamma(\frac{1}{2}f_2 - r)}{\Gamma(\frac{1}{2}f_2)} \times \sum_{j=0}^r \binom{r}{j} \frac{\frac{1}{2}\lambda f_1)^j}{\Gamma(\frac{1}{2}f_1 + j)} \times e^{\lambda_2/2} M\left(\frac{1}{2}(n-2r), \frac{1}{2}n, \frac{1}{2}\lambda_2\right) \quad (37.11.11)$$

The first four raw moments are given by

$$\begin{aligned} \mu'_1 &= \frac{n}{m} \frac{m+\lambda}{n-2} \times e^{\lambda_2/2} M\left(\frac{1}{2}(n-2), \frac{1}{2}n, \frac{1}{2}\lambda_2\right) \\ \mu'_2 &= \left(\frac{n}{m}\right)^2 \frac{\lambda^2 + (2\lambda + m)(m+2)}{(n-2)(n-4)} \times e^{\lambda_2/2} M\left(\frac{1}{2}(n-4), \frac{1}{2}n, \frac{1}{2}\lambda_2\right) \\ \mu'_3 &= \left(\frac{n}{m}\right)^3 \frac{\lambda^3 + 3(m+4)\lambda^2 + (3\lambda+m)(m+4)(m+2)}{(n-2)(n-4)(n-6)} \times e^{\lambda_2/2} M\left(\frac{1}{2}(n-6), \frac{1}{2}n, \frac{1}{2}\lambda_2\right) \\ \mu'_4 &= \left(\frac{n}{m}\right)^4 \frac{\lambda^3 + 4(m+6)\lambda^3 + 6(m+6)(m+4)\lambda^2 + (4\lambda+m)(m+6)(m+4)(m+2)}{(n-2)(n-4)(n-6)(n-8)} \\ &\quad \times e^{\lambda_2/2} M\left(\frac{1}{2}(n-8), \frac{1}{2}n, \frac{1}{2}\lambda_2\right) \end{aligned}$$

### 37.11.4.2 Recurrence relations

Let the density  $g_{m,n}$  be that of  $m/n$  times an  $F_{m,n}$  random variable. Let  $G_{m,n}(y)$  be its distribution function, and let  $g_{m,n}^{\lambda_1, \lambda_2}$  and  $G_{m,n}^{\lambda_1, \lambda_2}(y)$  be the density and distribution function of its doubly noncentral version (the distribution of  $\chi_m^2(\lambda_1)/\chi_n^2(\lambda_2)$ ). Then the following recurrence relations hold (Chattamvelli & Jones, 1995)

$$n \left[ G_{m,n}^{\lambda}(y) - G_{m,n}^{\lambda}(y) \right] = -2g_{m,n}^{\lambda}(y) \quad (37.11.12)$$

$$\lambda_1(1+y)g_{m+4,n}^{\lambda_1, \lambda_2}(y) = [\lambda_1y - m(1+y)]g_{m+2,n}^{\lambda_1, \lambda_2}(y) + y(m+n)g_{m,n}^{\lambda_1, \lambda_2}(y) + \lambda_2g_{m,n+2}^{\lambda_1, \lambda_2}(y). \quad (37.11.13)$$

$$\lambda_2(1+y)g_{m,n+4}^{\lambda_1, \lambda_2}(y) = [\lambda_2y - n(1+y)]g_{m+2,n}^{\lambda_1, \lambda_2}(y) + (m+n)g_{m,n}^{\lambda_1, \lambda_2}(y) + \lambda_1g_{m+2,n}^{\lambda_1, \lambda_2}(y). \quad (37.11.14)$$

$$\lambda_1g_{m+4,n-2}^{\lambda_1, \lambda_2}(y) = -mg_{m+2,n}^{\lambda_1, \lambda_2}(y) + nyg_{m,n+2}^{\lambda_1, \lambda_2}(y) + \lambda_2g_{m+2,n}^{\lambda_1, \lambda_2}(y). \quad (37.11.15)$$

From equations 37.11.12 to 37.11.15 we obtain

$$\begin{aligned} \lambda_1(1+y)G_{m+6,n}^{\lambda_1, \lambda_2}(y) &= [\lambda_1y - (m+2-\lambda_1)(1+y)]G_{m+4,n}^{\lambda_1, \lambda_2}(y) \\ &\quad + [(m+2)(1+y) + y(m+n-\lambda_1)]G_{m+2,n}^{\lambda_1, \lambda_2}(y) \\ &\quad - y(m+n)G_{m,n}^{\lambda_1, \lambda_2}(y) + \lambda_2y[G_{m+2,n+2}^{\lambda_1, \lambda_2}(y) - G_{m,n+2}^{\lambda_1, \lambda_2}(y)] \end{aligned} \quad (37.11.16)$$

$$\begin{aligned} [n(1+y) - \lambda_2][G_{m,n+2}^{\lambda_1, \lambda_2}(y) - G_{m+2,n+2}^{\lambda_1, \lambda_2}(y)] &= (m+n)[G_{m+2,n}^{\lambda_1, \lambda_2}(y) - G_{m,n}^{\lambda_1, \lambda_2}(y)] \\ &\quad + \lambda_2(1+y)[G_{m+2,n}^{\lambda_1, \lambda_2}(y) - G_{m+2,n}^{\lambda_1, \lambda_2}(y)] \\ &\quad - \lambda_1[G_{m+2,n}^{\lambda_1, \lambda_2}(y) - G_{m+4,n}^{\lambda_1, \lambda_2}(y)] \end{aligned} \quad (37.11.17)$$

$$(m+2)[G_{m+2,n}^{\lambda_1,\lambda_2}(y) = (\lambda_1 - m - 2)G_{m+4,n}^{\lambda_1,\lambda_2}(y) - \lambda_1 G_{m+6,n}^{\lambda_1,\lambda_2}(y) \\ + y[nG_{m,n}^{\lambda_1,\lambda_2}(y) - nG_{m+2,n}^{\lambda_1,\lambda_2}(y) \\ - \lambda_2 G_{m,n}^{\lambda_1,\lambda_2}(y) - \lambda_2 G_{m+2,n}^{\lambda_1,\lambda_2}(y)] \quad (37.11.18)$$

See sections ?? and 37.10.4.2 for the corresponding expressions for the central and singly noncentral F distribution.

### 37.11.5 Random Numbers

---

Function **DoublyNoncentralFDistRanEx**(*Size* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *lambda1* As *mpNum*, *lambda2* As *mpNum*, *Generator* As *String*) As *mpNumList*

---

**NOT YET IMPLEMENTED**

---

The function DoublyNoncentralFDistRanEx returns random numbers following a noncentral *F*-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*lambda1*: A real number greater 0, representing the numerator noncentrality parameter

*lambda2*: A real number greater 0, representing the denominator noncentrality parameter

*Generator*: A string describing the random generatorOutput? String? A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

Random numbers from a doubly non-central F-distribution are easily obtained using the definition in terms of the ratio between two independent random numbers from non-central chi-square distributions. This ought to be sufficient for most applications but if needed more efficient techniques may easily be developed e.g. using more general techniques.

### 37.11.6 Confidence Limits for the Noncentrality Parameters

---

Function **DoublyNoncentralFDistNoncentralityEx**(*alpha* As *mpNum*, *lambda1* As *mpNum*, *lambda2* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *Output* As *String*) As *mpNumList*

---

**NOT YET IMPLEMENTED**

---

The function DoublyNoncentralFDistNoncentralityEx returns confidence limits for the noncentrality parameter lambda and related information for the noncentral *F*-distribution.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*lambda1*: A real number greater 0, representing the numerator noncentrality parameter

*lambda2*: A real number greater 0, representing the denominator noncentrality parameter

*m*: A real number greater 0, representing the numerator degrees of freedom.

*n*: A real number greater 0, representing the denominator degrees of freedom.

*Output*: A string describing the output choices

See section 32.1.3.5 for the options for *alpha*, *Noncentrality* and *Output*. Algorithms and formulas are given below.

### 37.11.7 Sample Size

---

Function **DoublyNoncentralFDistSampleSizeEx**(*alpha* As *mpNum*, *beta* As *mpNum*, *m* As *mpNum*, *ModifiedNoncentrality1* As *mpNum*, *ModifiedNoncentrality1* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function *DoublyNoncentralFDistSampleSizeEx* returns sample size estimates and related information for the noncentral *F*-distribution.

**Parameters:**

*alpha*: A real number between 0 and 1, specifies the confidence level (or Type I error).

*beta*: A real number between 0 and 1, specifies the Type II error (or 1 - Power).

*m*: A real number greater 0, representing the numerator degrees of freedom.

*ModifiedNoncentrality1*: A real number greater 0, representing the modified numerator noncentrality parameter.

*ModifiedNoncentrality1*: A real number greater 0, representing the modified denominator noncentrality parameter.

*Output*: A string describing the output choices

See section 32.1.3.4 for the options for *alpha*, *beta*, *ModifiedNoncentrality* and *Output*. Algorithms and formulas are given below.

## 37.12 Noncentral Distribution of Roy's Largest Root

### 37.12.1 Definition

Let  $X, Y$  denote two independent real Gaussian  $p \times n_1$  and  $p \times n_2$  matrices with  $n_1, n_2 \geq p$ , each constituted by zero mean independent, identically distributed columns with common covariance. The Roy's largest root criterion, used in multivariate analysis of variance (MANOVA), is based on the statistic of the largest eigenvalue,  $\Theta_1$ , of  $(A + B)^{-1}B$ , where  $A$  and  $B$  are independent central Wishart matrices. Throughout this section, we define

$$s = p, \quad m = (n_2 - p - 1)/2, \quad n = (n_1 - p - 1)/2 \quad (37.12.1)$$

See [Chiani \(2012\)](#), and [Johnstone & Nadler \(2013\)](#),

### 37.12.2 The exact distribution (null case).

[Chiani \(2014\)](#) proposes the following algorithm:

The CDF of the largest eigenvalue  $\Theta_1$  for a multivariate beta matrix in the null case is:

$$F_{\Theta_1}(\theta_1) = \Pr[\Theta_1 \leq \theta_1] = C \sqrt{|A(\theta_1)|}, \quad \text{where} \quad (37.12.2)$$

$$C = \pi^{s/2} \prod_{i=1}^s \frac{\Gamma\left(\frac{1}{2}(i+2m+2n+s+2)\right)}{\Gamma\left(\frac{i}{2}\right) \Gamma\left(\frac{1}{2}(i+2m+1)\right) \Gamma\left(\frac{1}{2}(i+2n+1)\right)} \quad (37.12.3)$$

When  $s$  is even, we have  $n_{mat} = s$  and the elements of the  $s \times s$  skew-symmetric matrix  $A(\theta_1)$  are:

$$a_{i,j}(\theta_1) = E(\theta_1; m+j, m+i) - E(\theta_1; m+i, m+j) \quad i, j = 1, \dots, s \quad (37.12.4)$$

where

$$E(x; a, b) = \int_0^x t^{a-1} (1-t)^n I(t; b, n+1) dt. \quad (37.12.5)$$

When  $s$  is odd, we have  $n_{mat} = s+1$  and the elements of the  $(s+1) \times (s+1)$  skew-symmetric matrix  $A(\theta_1)$  are as in equation (37.12.4), with the additional elements

$$a_{i,s+1}(\theta_1) = I(\theta_1; m+i, n+1) \quad i = 1, \dots, s \quad (37.12.6)$$

$$a_{s+1,j}(\theta_1) = -a_{j,s+1}(\theta_1) \quad j = 1, \dots, s \quad (37.12.7)$$

$$a_{s+1,s+1}(\theta_1) = 0. \quad (37.12.8)$$

Note that  $a_{i,j}(\theta_1) = -a_{j,i}(\theta_1)$  and  $a_{i,i}(\theta_1) = 0$ . To avoid the numerical integration in equation (37.12.5), we first observe that the incomplete beta functions can be computed iteratively by the relation

$$I(x; a+1, b) = I(x; a, b) - \frac{x^a (1-x)^b}{a+b}. \quad (37.12.9)$$

Then, from equations (37.12.4) and (37.12.5) the following identities can be easily verified:

$$E(x; a, a) = \frac{1}{2} I(x; a, n+1)^2 \quad (37.12.10)$$

$$E(x; a, b+1) = b \frac{E(x; a, b)}{b+n+1} - \frac{I(x; a+b, 2n+2)}{b+n+1} \quad (37.12.11)$$

$$E(x; b, a) = I(x; a, n+1) I(x; b, n+1) - E(x; a, b). \quad (37.12.12)$$

### 37.12.3 Approximation of the CDF(null case)

Chiani (2014) proposes the following algorithm:

$$F_{\Theta_1}(\theta_1) \approx P\left(k, \frac{\log(\theta_1/(1-\theta_1)) - \mu + \sigma\alpha}{\delta}\right) \quad (37.12.13)$$

and for its inverse, useful for evaluating the percentiles,

$$F_{\Theta_1}^{-1}(\theta_1) \approx \frac{\exp[\sigma(\delta P^{-1}(k, y) - \alpha)] + \mu}{1 + \exp[\sigma(\delta P^{-1}(k, y) - \alpha)] + \mu}, \quad (37.12.14)$$

where

$$\mu = 2 \log \tan\left(\frac{\gamma + \phi}{2}\right), \quad \sigma^3 = \frac{16}{(m+n+1)^2} \frac{1}{\sin^2(\gamma + \phi) \sin \gamma \sin \phi} \quad (37.12.15)$$

$$\gamma = \arccos\left(\frac{m+n-2p}{m+n-1}\right), \quad \phi = \arccos\left(\frac{m-n}{m+n-1}\right), \quad (37.12.16)$$

and the constants  $k = 46.446$ ,  $\delta = 0.186054$ ,  $\alpha = 9.84801$  have been chosen to match the moments of the approximation to that of the Tracy-Widom.

## 37.13 Noncentral Distribution of Wilks' Lambda

### 37.13.1 Definitions

Both Wilks  $U$  and Wilks  $\Lambda$  are in use, with  $\Lambda = U^{N/2}$ ,  $U = \Lambda^{2N}$ , and  $z = -2\rho(\Lambda)$ .

Literature: Krishnaiah et al (1971), Walster (1980), Fujikoshi, Pillai (1965), p. 408.

Edgeworth expansion: [Wakaki \(2006\)](#)

#### 37.13.1.1 General Linear Model

Suppose  $E$  is a  $p \times p$  matrix error sums of squares with a central Wishart <sub>$p$</sub> ( $n, \Sigma$ ) distribution. Let  $T = V^T V$  be the treatment sum of squares with a noncentral Wishart <sub>$p$</sub> ( $m, \Omega, \Sigma$ ) distribution with  $m$  degrees of freedom and noncentrality matrix  $\Omega = \Sigma^{-1} M^T M$ . This results when  $V$  is  $(m \times p)$  with a Normal <sub>$m \times p$</sub> ( $M, \Sigma$ ) distribution, with mean  $e(V) = M$  and columns that are independent with common covariance  $\Sigma$ .

Let  $W_{GLM} = |E|/|T + E|$ .

See [O'Brien & Shieh \(1992\)](#)

See [Lee \(1972a\)](#)

See [Lee \(1971a\)](#)

See [Kulp & Nagarsenker \(1984\)](#)

Noncentral F approximations : [Muller et al. \(1992\)](#) , confidence bounds for power: [Taylor & Muller \(1995\)](#)

#### 37.13.1.2 Block independence

Suppose blocks of variables with dimensions  $p_1$  and  $p_2$  where  $p_1 + p_2 = p$ . Let  $A$  be the matrix of sample covariances with a Wishart <sub>$p$</sub> ( $n, \Sigma$ ) distribution. Specify  $A$  in block form as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad (37.13.1)$$

where  $A_{ij}$  is  $p_i \times p_j$ . The likelihood ratio test for block independence rejects for small values of  $W_{CORR} = |A|/(|A_{11}||A_{22}|)$ .

See [Lee \(1971b\)](#)

See also [Butler & Wood \(2005\)](#)

### 37.13.2 Density and CDF (Overview)

---

Function **NoncentralWilksLambdaDist**(**x** As mpNum, **p** As mpNum, **m** As mpNum, **n** As mpNum, **Omega** As mpNum, **Output** As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function NoncentralWilksLambdaDist returns pdf, CDF and related information for the non-central WilksLambda-distribution

#### Parameters:

**x**: A real number

**p**: An integer greater 0, representing the number of variates

**m**: A real number greater 0, representing the numerator degrees of freedom

**n**: A real number greater 0, representing the denominator degrees of freedom

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.1 for the options for *Output*. Algorithms and formulas are given below.

### 37.13.3 Density: Details

#### 37.13.3.1 Exact null distribution

$$\Pr[x \leq x_0] = f_{BetaProd}(x; b_{i=1 \dots p}, c_{i=1 \dots p}), \quad \text{where} \quad (37.13.2)$$

$$b_i = \frac{1}{2}(n_2 - i + 1); \quad c_i = \frac{1}{2}(n_1 + n_2 - i + 1); \quad i = 1 \dots p. \quad (37.13.3)$$

#### 37.13.3.2 Exact distribution for the noncentral linear case (General Linear Model)

Let  $\delta$  denote the noncentrality parameter in the linear case. Then (Tretter & Walster, 1975), Walster & Tretter (1980)

$$f_{\Lambda,GLM}(p, m, n; x; \delta) = e^{-\delta} \sum_{k=0}^{\infty} \frac{\delta_k}{k!} f_{BetaProd}(x_0; b_{i=1 \dots p}, c_{i=1 \dots p}), \quad \text{where} \quad (37.13.4)$$

$$b_i = \frac{1}{2}(n_2 - i + 1); \quad i = 1 \dots p. \quad (37.13.5)$$

$$c_1 = \frac{1}{2}(n_1 + n_2 + 2k); \quad c_i = \frac{1}{2}(n_1 + n_2 - i + 1); \quad i = 2 \dots p. \quad (37.13.6)$$

#### 37.13.3.3 Exact distribution for the noncentral linear case (Block Independence)

Let  $\delta$  denote the noncentrality parameter in the linear case. Then

$$f_{\Lambda,CORR}(p, m, n; x; \rho^2) = e^{-\delta} \sum_{k=0}^{\infty} \frac{\delta_k}{k!} f_{BetaProd}(x_0; b_{i=1 \dots p}, c_{i=1 \dots p}), \quad \text{where} \quad (37.13.7)$$

$$b_i = \frac{1}{2}(n_2 - i + 1); \quad i = 1 \dots p. \quad (37.13.8)$$

$$c_1 = \frac{1}{2}(n_1 + n_2 + 2k); \quad c_i = \frac{1}{2}(n_1 + n_2 - i + 1); \quad i = 2 \dots p. \quad (37.13.9)$$

### 37.13.4 CDF: Details

#### 37.13.4.1 Exact null distribution

$$\Pr[x \leq x_0] = F_{BetaProd}(x; b_{i=1 \dots p}, c_{i=1 \dots p}), \quad \text{where} \quad (37.13.10)$$

$$b_i = \frac{1}{2}(n_2 - i + 1); \quad c_i = \frac{1}{2}(n_1 + n_2 - i + 1); \quad i = 1 \dots p. \quad (37.13.11)$$

#### 37.13.4.2 Box Expansion (null distribution)

See section 36.6.4.3 for definition of the coefficients of the Box Expansion for Wilks U.

#### 37.13.4.3 Exact distribution for the noncentral linear case (General Linear Model)

Let  $\delta$  denote the noncentrality parameter in the linear case. Then

$$F_{\Lambda,GLM}(p, m, n; x; \delta) = e^{-\delta} \sum_{k=0}^{\infty} \frac{\delta_k}{k!} F_{BetaProd}(x_0; b_{i=1 \dots p}, c_{i=1 \dots p}), \quad \text{where} \quad (37.13.12)$$

$$b_i = \frac{1}{2}(n_2 - i + 1); \quad i = 1 \dots p. \quad (37.13.13)$$

$$c_1 = \frac{1}{2}(n_1 + n_2 + 2k); \quad c_i = \frac{1}{2}(n_1 + n_2 - i + 1); \quad i = 2 \dots p. \quad (37.13.14)$$

### 37.13.4.4 Exact distribution for the noncentral linear case (Block Independence)

Let  $\delta$  denote the noncentrality parameter in the linear case. Then

$$F_{\Lambda, CORR}(p, m, n; x; \rho^2) = e^{-\delta} \sum_{k=0}^{\infty} \frac{\delta_k}{k!} F_{BetaProd}(x_0; b_{i=1 \dots p}, c_{i=1 \dots p}), \quad \text{where} \quad (37.13.15)$$

$$b_i = \frac{1}{2}(n_2 - i + 1); \quad i = 1 \dots p. \quad (37.13.16)$$

$$c_1 = \frac{1}{2}(n_1 + n_2 + 2k); \quad c_i = \frac{1}{2}(n_1 + n_2 - i + 1); \quad i = 2 \dots p. \quad (37.13.17)$$

### 37.13.4.5 Approximation by non-central Chi-Square (General Linear Model)

Fujikoshi (1973) proposes the following approximation:

$$\begin{aligned} F_{\Lambda, GLM}(p, m, n; x; \Omega) &= F_{\chi^2}(pm, x; \omega_1) + \frac{1}{4n} \sum_{k=1}^3 a_k F_{\chi^2}(pm + 2k, x; \omega_1) \quad (37.13.18) \\ &+ \frac{1}{96n^2} \sum_{k=1}^6 b_k F_{\chi^2}(pm + 2k, x; \omega_1) \\ &+ \frac{1}{96n^3} \sum_{k=1}^9 c_k F_{\chi^2}(pm + 2k, x; \omega_1) + O(n^{-4}) \end{aligned}$$

$$\Omega = \Lambda^{-1} M M'$$

$$\omega_j = \text{tr } \Omega^j$$

$$m = n + (pq - 1)/2$$

$$x = m \log(x)$$

$$s = (p + q + 1)/4$$

$$r = f(p^2 + q^2 - 5)/48$$

$$a_1 = 2s\omega_1$$

$$a_2 = (2s\omega_1 - \omega_2)$$

$$a_3 = \omega_2$$

$$b_0 = r$$

$$b_1 = 0$$

$$b_2 = r4s^2\omega_1 + 2s^2\omega_1^2 + 2s\omega_2$$

$$b_3 = 4s^2\omega_1 - (1 + 4s^2)\omega_1^2 - (1 + 8s)\omega_2 + 2s\omega_1\omega_2 + (4/3)\omega_3$$

$$b_4 = (1 + 2s^2)\omega_1^2 + (1 + 6s)\omega_2 - 4s\omega_1\omega_2 - 4\omega_3 + \omega_2^2/2$$

$$b_5 = 2s\omega_1\omega_2 + (8/3)\omega_3 - \omega_2^2$$

$$b_6 = \omega_2^2/2$$

$$\begin{aligned}
c_1 &= 2rs\omega_1 & (37.13.19) \\
c_2 &= r(2s\omega_1\omega_2) \\
c_3 &= 2s(r+4s^2)\omega_1 + 2s(1+4s^2)\omega_1^2 + (-r+2s+12s^2)\omega_2 - \frac{4}{3}s^3\omega_3^3 - 4s^2\omega_1\omega_2 - \frac{8}{3}s\omega_3 \\
c_4 &= 2s(r+4s^2)\omega_1 - (1+10s+16s^3)\omega_1^2 - (3+r+10s+36s^2)\omega_2 + 2s(1+2s^2)\omega_1^3 \\
&\quad + 2(2+s+12s^2)\omega_1\omega_2 + 4(1+6s)\omega_3 - 2s^2\omega_1^2\omega_2 - 2s\omega_2^2 - \frac{8}{3}s\omega_1\omega_3 - 2\omega_4 \\
c_5 &= (1+8s+8s^3)\omega_1^2 + (3+r+8s+24s^2)\omega_2 - 4s(1+s^2)\omega_1^3 - 4(3+s+9s^2)\omega_1\omega_2 - 12(1+4s)\omega_3 \\
&\quad + (1+6s^2)\omega_1^2\omega_2 + (1+10s)\omega_2^2 + \frac{32}{3}s\omega_1\omega_3 + 12\omega_4 - \frac{4}{3}\omega_2\omega_3 - s\omega_1\omega_2^2 \\
c_6 &= s(2+\frac{4}{3}s^2)\omega_1^3 + 2(4+s+8s^2)\omega_1\omega_2 + 8(1+\frac{10}{3}s)\omega_3 - 2(1+3s^2)\omega_1^2\omega_2 - 2(1+7s)\omega_2^2 \\
&\quad - \frac{40}{3}s\omega_1\omega_3 - 20\omega_4 + \frac{16}{3}\omega_2\omega_3 + 3s\omega_1\omega_2^2 - \frac{1}{6}\omega_2^3 \\
c_7 &= (1+2s^2)\omega_1^2\omega_2 + (1+6s)\omega_2^2 + \frac{16}{3}s\omega_1\omega_3 + 10\omega_4 - \frac{20}{3}\omega_2\omega_3 - 3s\omega_1\omega_2^2 + \frac{1}{2}\omega_2^3 \\
c_8 &= \frac{8}{3}\omega_2\omega_3 + s\omega_1\omega_2^2 - \frac{1}{2}\omega_2^3 \\
c_9 &= \frac{1}{6}\omega_2^3
\end{aligned}$$

### 37.13.4.6 Approximation by non-central Chi-Square (Block Independence)

Lee (1971b) proposes the following approximation:

$$\begin{aligned}
F_{\Lambda, CORR}(p, m, n; x; P) &= F_{\chi^2}(pm, x; s1) + \frac{1}{4n} \sum_{k=1}^3 a_k F_{\chi^2}(pm + 2k, x; s1) \quad (37.13.20) \\
&\quad + \frac{1}{96n^2} \sum_{k=1}^6 b_k F_{\chi^2}(pm + 2k, x; s1) + O(n^{-3})
\end{aligned}$$

$$\Omega = \Lambda^{-1} M M'$$

$$\omega_j = \text{tr } \Omega^j$$

$$m = n + (pq - 1)/2$$

$$x = m \log(x)$$

$$s = (p + q + 1)/4$$

$$r = f(p^2 + q^2 - 5)/48$$

$$a_0 = -q\omega_1 + \omega_2$$

$$a_1 = (2s + q)\omega_1 - 2\omega_2$$

$$a_2 = -2s\omega_1 + 2\omega_2$$

$$a_3 = -\omega_2$$

$$b_0 = -r - ql\omega_1 + (q + l)\omega_2 + \frac{1}{2}q^2\omega_1^2 - \frac{4}{3}\omega_3 - q\omega_1\omega_2 + \frac{1}{2}\omega_2^2$$

$$b_1 = q^2\omega_1 - 4q\omega_2 - q(q + 2s)\omega_1^2 + 4\omega_3 + (3q + 2s)\omega_1\omega_2 - 2\omega_2^2$$

$$b_2 = r - 2s(q + 2s)\omega_1 + (2p + 6q + 3)\omega_2 + (\frac{1}{2}l^2 + 6qs + 1)\omega_1^2 - 8\omega_3 - (4q + 6s)\omega_1\omega_2 + 4\omega_2^2$$

$$b_3 = 4s^2\omega_1 - (3p + 5q + 5)\omega_2 - (4s^2 + 2qs + 2)\omega_1^2 + \frac{32}{3}\omega_3 + (3q + 8s)\omega_1\omega_2 - 5\omega_2^2$$

$$b_4 = (6s + 1)\omega_2 + (2s^2 + 1)\omega_1^2 - 8\omega_3 - (q + 6s)\omega_1\omega_2 + 4\omega_2^2$$

$$b_5 = \frac{8}{3}\omega_3 + 2s\omega_1\omega_2 - 2\omega_2^2$$

$$b_6 = \frac{1}{2}\omega_2^2$$

### 37.13.4.7 Saddlepoint approximation (Null distribution)

### 37.13.4.8 Saddlepoint approximation (General Linear Model)

Butler & Wood (2005) propose the following approximation

$$F_{\Lambda, GLM}(p, m, n; y; \Omega) \approx 1 - \Phi(r) + \phi(r)(u^{-1} - r^{-1}) \quad (37.13.21)$$

where  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the cdf (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively,  $r = \text{sgn}(s)\sqrt{2(sy - K(s))}$ ,  $u = s\sqrt{K''(s)}$ , and  $s$  is the solution to the saddlepoint equation

$$K'(s) = y. \quad (37.13.22)$$

$K(s)$  denotes the cumulant generating function and is given by

$$K(s) = \ln \left[ \frac{\Gamma_p(n/2 + s)\Gamma_p((n+m)/2)}{\Gamma_p(n/2)\Gamma_p((n+m)/2 + s)} {}_1F_1 \left( s, \frac{1}{2}(n+m) + s, -\frac{1}{2}\Omega \right) \right] \quad (37.13.23)$$

and its first derivative,  $K'(s)$ , is given by

$$K'(s) = \sum_{i=1}^p \left[ \psi \left( \frac{1}{2}n + s - \frac{1}{2}(i-1) \right) - \psi \left( \frac{1}{2}(n+m) + s - \frac{1}{2}(i-1) \right) \right] \quad (37.13.24)$$

$$+ \frac{\partial}{\partial s} \ln \left[ {}_1F_1 \left( s, \frac{1}{2}(n+m) + s, -\frac{1}{2}\Omega \right) \right], \quad (37.13.25)$$

where  $\Gamma_p(\cdot)$  is the multivariate gamma function (see section 35.1.1),  $\psi(\cdot)$  is the digamma function (see section 13.7.2), and  ${}_1F_1(\cdot, \cdot, X)$  is the confluent hypergeometric function of matrix argument (see section 35.1.4). The saddlepoint equation (37.13.22) needs to be evaluated numerically. Also, the computation of  $K''(s)$  is performed using a numerical derivative of  $K'(s)$ .

### 37.13.4.9 Saddlepoint approximation (Block Independence)

Butler & Wood (2005) propose the following approximation

$$F_{\Lambda, \text{CORR}}(p, m, n; y; P) \approx 1 - \Phi(r) + \phi(r)(u^{-1} - r^{-1}) \quad (37.13.26)$$

where  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the cdf (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively,  $r = \text{sgn}(s)\sqrt{2(sy - K(s))}$ ,  $u = s\sqrt{K''(s)}$ , and  $s$  is the solution to the saddlepoint equation

$$K'(s) = y. \quad (37.13.27)$$

$K(s)$  denotes the cumulant generating function and is given by

$$K(s) = \ln \left[ \frac{\Gamma_{p_1}(n/2)\Gamma_{p_1}((n-p_2)/2 + s)}{\Gamma_{p_1}(n/2 + s)\Gamma_{p_1}((n-p_2)/2)} \times |I_{p_1} - P^2|^{n/2} {}_2F_1 \left( \frac{n}{2}, \frac{n}{2}; \frac{n}{2} + s; P^2 \right) \right] \quad (37.13.28)$$

and its first derivative,  $K'(s)$ , is given by

$$K'(s) = \sum_{i=1}^p \left[ \psi \left( \frac{1}{2}n + s - \frac{1}{2}(i-1) \right) - \psi \left( \frac{1}{2}(n-p_2) + s - \frac{1}{2}(i-1) \right) \right] \quad (37.13.29)$$

$$+ \frac{\partial}{\partial s} \ln \left[ |I_{p_1} - P^2|^{n/2} {}_2F_1 \left( \frac{n}{2}, \frac{n}{2}; \frac{n}{2} + s; P^2 \right) \right], \quad (37.13.30)$$

where  $\Gamma_p(\cdot)$  is the multivariate gamma function (see section 35.1.1),  $\psi(\cdot)$  is the digamma function (see section 13.7.2), and  ${}_2F_1(\cdot, \cdot, \cdot, X)$  is the Gauss hypergeometric function of matrix argument (see section 35.1.3). The saddlepoint equation (37.13.27) needs to be evaluated numerically. Also, the computation of  $K''(s)$  is performed using a numerical derivative of  $K'(s)$ .

### 37.13.5 Quantiles

---

Function **NoncentralWilksLambdaDistInv**(*Prob* As mpNum, *p* As mpNum, *m* As mpNum, *n* As mpNum, *Omega* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralWilksLambdaDistInv returns quantiles and related information for the the noncentral WilksLambda-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*p*: An integer greater 0, representing the number of variates

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

### 37.13.6 Properties

---

Function **NoncentralWilksLambdaDistInfo**(*p* As mpNum, *m* As mpNum, *n* As mpNum, *Omega* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function NoncentralWilksLambdaDistInfo returns moments and related information for the noncentral WilksLambda-distribution

**Parameters:**

*p*: An integer greater 0, representing the number of variates

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.13.6.1 Moments: Null distribution

#### 37.13.6.2 Moments: General Linear Model

The noncentral moments of  $W_{GLM}$  are specified in Theorem 10.5.1 of [Muirhead \(1982\)](#) as

$$E(W^s) = \frac{\Gamma_p(n/2 + s)\Gamma_p((n+m)/2)}{\Gamma_p(n/2)\Gamma_p((n+m)/2 + s)} {}_1F_1\left(s; \frac{n+m}{2} + s; -\frac{1}{2}\Omega\right) \quad (37.13.31)$$

where  $\Gamma_p(\cdot)$  is the multivariate gamma function (see section 35.1.1) and  ${}_1F_1(\cdot, \cdot, X)$  is the confluent hypergeometric function of matrix argument (see section 35.1.4). See also [Butler & Wood \(2002\)](#).

### 37.13.6.3 Moments: Block Independence

The noncentral moments of  $W_{CORR}$  are specified in Theorem 11.2.6 of [Muirhead \(1982\)](#) as

$$E(W^s) = \frac{\Gamma_{p_1}(n/2)\Gamma_{p_1}((n-p_2)/2+s)}{\Gamma_{p_1}(n/2+s)\Gamma_{p_1}((n-p_2)/2)} \times |I_{p_1} - P^2|^{n/2} {}_2F_1\left(\frac{n}{2}, \frac{n}{2}; \frac{n}{2} + s; P^2\right) \quad (37.13.32)$$

where  $P = \text{diag}\{\rho_1, \dots, \rho_{p_1}\}$  contains the population canonical correlations,  $\Gamma_p(\cdot)$  is the multivariate gamma function (see section 35.1.1) and  ${}_2F_1(\cdot, \cdot, \cdot, X)$  is the Gauss hypergeometric function of matrix argument (see section 35.1.3). See also [Butler & Wood \(2002\)](#).

### 37.13.7 Random Numbers

---

Function **NoncentralWilksLambdaDistRan**(*Size* As *mpNum*, *p* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *Generator* As *String*, *Omega* As *mpNum*, *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **NoncentralWilksLambdaDistRan** returns random numbers following a noncentral WilksLambda-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*p*: An integer greater 0, representing the number of variates

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below

#### 37.13.7.1 Null distribution

#### 37.13.7.2 General Linear Model

#### 37.13.7.3 Block Independence

[Lee \(1971b\)](#) gives the following algorithm: The set of transformed canonical correlations  $(\tilde{r}_1^2, \dots, \tilde{r}_p^2)$  is distributed like the roots of the determinantal equation

$$|\lambda W - [(\tilde{P}T + Z_1)(\tilde{P}T + Z_1)' + Z_2Z_2']| = 0, \quad \text{where} \quad (37.13.33)$$

$$\tilde{r}_i^2 = \frac{r_i^2}{1 - r_i^2}, \quad \tilde{\rho}_i^2 = \frac{\rho_i^2}{1 - \rho_i^2}, \quad \tilde{P} = \text{diag}(\tilde{\rho}_1, \dots, \tilde{\rho}_p), \quad (37.13.34)$$

$$W \sim W_p(I, N - q), \quad T(p \times p) \text{ is such that } TT' \sim W_p(I, N), \quad (37.13.35)$$

and  $Z_1(p \times p)$  and  $Z_2(p \times (q-p))$  are matrices with independent normal variates as their elements. All matrix variates figuring in equation 37.13.33 are independently distributed.

When  $p = 1$ , this reduces to the case of multiple correlation, as given in equation 37.2.22.

## 37.14 Noncentral Distribution of Hotelling's T2

The Lawley-Hotelling generalized  $T_0^2$  and Pillai's  $V$  statistic, defined respectively by

$$T_0^2 = \text{ntr}(AB^{-1}), \quad V = \text{ntr}(A(A+B)^{-1}), \quad (37.14.1)$$

have been suggested as alternatives to Wilk's criterion for testing multivariate linear hypotheses. Here  $A$  and  $B$  are independent  $p \times p$  Wishart matrices on  $q$  and  $n$  degrees of freedom respectively. [Davis \(1968, 1970b\)](#) has found the moments for both criteria and has given recurrence relations for them, together with a formula relating the moments for  $T_0^2$  and  $V$ .

### 37.14.1 Hotelling's T2, Central Moments

The moments of Hotelling's  $T$  exist up to the  $j$ th, where  $j$  is the largest integer such that  $j < \frac{1}{2}(n_2 - m + 1)$ , and can be determined as follows ([Davis, 1968](#)):

$$E(T^r) = (-1)^r r!(n_1 + n_2)! \sum_{k=0}^m \frac{l_{kr}}{(m + n_2 - k)!}, \quad \text{where} \quad (37.14.2)$$

$$\mathbf{l}_i = (l_{0i}, \dots, l_{mi})', \quad a_i = \frac{1}{2}(m - i)(n_2 - i), \quad (37.14.3)$$

$$\mathbf{l}_0 = \frac{n_2!}{(n_1 + n_2)!} (0, \dots, 0, 1)', \quad (37.14.4)$$

$$\mathbf{l}_r = \text{diag} \left( \frac{1}{(r - a_0)}, \dots, \frac{1}{(r - a_m)} \right) \sum_{s=0}^{r-1} \mathbf{l}_s, \quad (37.14.5)$$

### 37.14.2 Hotelling's T2, Exact distributions for p=1 and p=2

$$T2(1, n_1, n_2; x) = I \left( \frac{1}{2}n_1, \frac{1}{2}n_2; \frac{x}{1-x} \right) \quad (37.14.6)$$

$$\begin{aligned} T2(2, m, n, x) &= I \left( m - 1, n - 1, \frac{w}{2+w} \right) \\ &\quad - \frac{\sqrt{\pi} \Gamma(\frac{1}{2}(m+n-1))}{\Gamma(\frac{1}{2}m) \Gamma(\frac{1}{2}n)} (1+w)^{(1-n)/2} I \left( \frac{1}{2}(m-1), \frac{1}{2}(n-1); \frac{w^2}{(2+w)^2} \right) \end{aligned} \quad (37.14.7)$$

Lit.: Pillai & Young (1971)

### 37.14.3 Hotelling's T2, Approximation

Let  $m = (n_1 - p - 1)/2$  and  $n = (n_2 - p - 1)/2$ .

$$\mu_1 = p(2m + p + 1)/(2n)$$

$$\mu_2 = \mu_1(2m + 2n + p + 1)(2n + p)/[2n(n - 1)(2n + 1)]$$

$$\mu_3 = 2\mu_2(2m + n + p + 1)(n + p)/[n(n - 2)(n + 1)]$$

$$a = \frac{(2\mu_1 3\mu_2 + 3\mu_1^2 \mu_3 - 6\mu_1 \mu_2^2 - \mu_2 \mu_3)}{(\mu_2 \mu_3 + 4\mu_1 \mu_2^2 - \mu_1^2 \mu_3)}, \quad b = \frac{(a+1)(a+3) - \mu_1^2/\mu_2}{(a+1) - \mu_1^2/\mu_2} \quad (37.14.8)$$

$$K = \frac{\mu_1(b-a-2)}{(a+1)} \quad w = \frac{x}{x+K} \quad (37.14.9)$$

Then  $x = T^2/n_2$  is approximately distributed as  $I_w(a+1, b-a-1)$ .

### 37.14.4 Hotelling's T2, Noncentral distribution: GLM

' Fujikoshi (1974), Ann. Inst. Math. Statist., 26, p. 289

$$F_{T^2,GLM}(p, m, n; x; \Omega) = F_{\chi^2}(pm, x; s1) + \frac{1}{4n} \sum_{k=0}^4 a_k F_{\chi^2}(pm + 2k, x; s1) + \frac{1}{96n^2} \sum_{k=0}^8 b_k F_{\chi^2}(pm + 2k, x; s1) + O(n^{-3}) \quad (37.14.10)$$

$$\begin{aligned} a_0 &= fg \\ a_1 &= -2g(f - 2\lambda_1) \\ a_2 &= fg - 8g\lambda_1 + 4\lambda_2 \\ a_3 &= 4(g\lambda_1 - 2\lambda_2) \\ a_4 &= 4\lambda_2 \\ b_0 &= fl_0 \\ b_1 &= l_1(f - 2\lambda_1) \\ b_2 &= fl_2 + 2(l_1 - 2l_2)\lambda_1 + 48g^2\lambda_1^2 + 24(f + 4)g\lambda_2 \\ b_3 &= fl_3 + 2(2l_2 - 3l_3)\lambda_1 - 192(g^2 + 1)\lambda_1^2 - 96((f + 8)g + 2)\lambda_2 + 96g\lambda_1\lambda_2 + 128\lambda_3 \\ b_4 &= fl_4 + 2(3l_3 - 4l_4)\lambda_1 + 96(3g^2 + 7)\lambda_1^2 + 48(3(f + 12)g + 14)\lambda_2 - 384g\lambda_1\lambda_2 - 768\lambda_3 + 48\lambda_2^2 \\ b_5 &= 8l_4\lambda_1 - 192(g^2 + 4)\lambda_1^2 - 96((f + 16)g + 8)\lambda_2 + 576g\lambda_1\lambda_2 + 1536\lambda_3 - 192\lambda_2^2 \\ b_6 &= 48(g^2 + 6)\lambda_1^2 + 24((f + 20)g + 12)\lambda_2 - 384g\lambda_1\lambda_2 - 1280\lambda_3 + 288\lambda_2^2 \\ b_7 &= 96g\lambda_1\lambda_2 + 384\lambda_3 - 192\lambda_2^2 \\ b_8 &= 48\lambda_2^2 \end{aligned}$$

### 37.14.5 Hotelling's T2, Noncentral distribution: CORR

$$F_{T^2,CORR}(p, m, n; x; P) = F_{\chi^2}(pm, x; s1) + \frac{1}{4n} \sum_{k=0}^4 a_k F_{\chi^2}(pm + 2k, x; s1) + \frac{1}{96n^2} \sum_{k=0}^8 b_k F_{\chi^2}(pm + 2k, x; s1) + O(n^{-3}) \quad (37.14.11)$$

$$\begin{aligned} s_1 &= 2\lambda_1 \\ s_2 &= 4\lambda_2 \\ s_3 &= 8\lambda_3 \\ s_1^2 &= s_1 s_1 \\ s_2^2 &= s_2 s_2 \\ p1 &= p + 1 \\ p^3 &= p^2 p \\ p^4 &= p^3 p \\ q^3 &= q^2 q \\ q^4 &= q^3 q \\ h &= qp1 \\ H1 &= 2q + p1 \\ p^2 p &= p^2 + p \\ a_0 &= qp(q - p - 1) - 2qs_1 + s_2 \\ a_1 &= -2q^2 p + 4qs_1 - 2s_2 \\ a_2 &= qp(q + p + 1) - 2(2q + p + 1)s_1 + 2s_2 \\ a_3 &= 2(q + p + 1)s_1 - 2s_2 \end{aligned}$$

$$a_4 = s_2$$

$$b_0 = qp(3qp^3 - 2(3q^2 - 3q + 4)p^2 + 3(q^3 - 2q^2 + 5q - 4)p - 8q^2 + 12q + 4) - 12q^2p(q - p - 1)s_1 - 6q(p^2 - qp + p - 4)s_2 + 12q^2s_1^2 - 16s_3 - 12qs_1s_2 + 3s_2^2$$

$$b_1 = -12q^3p^2(q - p - 1) - 24q^2(p^2 - 2qp + p - 2)s_1 + 12q(p^2 - 2qp + p - 8)s_2 - 48q^2s_1^2 + 48s_3 + 48qs_1s_2 - 12s_2^2$$

$$b_2 = -6q^2p^4 - 12q^2p^3 + 18q^2(q^2 + 1)p^2 + 24q^2(2q + 1)p + 12q(p^3 + 2p^2 - 7(q^2 + 1)p - 16q - 8)s_1 - 6(qp^2 - (7q^2 - q + 8)p - 40q - 12)s_2 + 24(qp + 4q^2 + q + 1)s_1^2 - 12(p + 8q + 1)s_1s_2 - 96s_3 + 24s_2^2$$

$$b_3 = -(12q^3 + 16q)p^3 - (12q^4 + 12q^3 + 96q^2 + 48q)p^2 - (64q^3 + 96q^2 + 64q)p + 12(-qp^3 + (4q^2 - 2q + 4)p^2 + (7q^3 + 4q^2 + 31q + 12)p + 4(7q^2 + 8q + 4))s_1 - 48((q^2 + 3)p + 9q + 5)s_2 - 24(3qp + 5q^2 + 3q + 4)s_1^2 + 176s_3 + 12(3p + 11q + 3)s_1s_2 - 36s_2^2$$

$$b_4 = 3q^2p^4 + (6q^3 + 6q^2 + 24q)p^3 + (3q^4 + 6q^3 + 63q^2 + 60q)p^2 + (24q^3 + 60q^2 + 60q)p - 12(qp^3 + (5q^2 + 2q + 12)p^2 + (4q^3 + 5q^2 + 45q + 32)p + 4(6q^2 + 11q + 9))s_1 + 6(qp^2 + (7q^2 + q + 44)p + 88q + 76)s_2 + 12(p^2 + 2(4q + 1)p + 8q^2 + 8q + 17)s_1^2 - 12(4p + 11q + 4)s_1s_2 - 240s_3 + 42s_2^2$$

$$b_5 = (12qp^3 + 24(q^2 + q + 4)p^2 + 12(q^3 + 2q^2 + 21q + 20)p + 48(2q^2 + 5q + 5))s_1 - 12(qp^2 + (2q^2 + q + 24)p + 32q + 40)s_2 - 24(p^2 + (3q + 2)p + 2q^2 + 3q + 9)s_1^2 + 240s_3 + 48(p + 2q + 1)s_1s_2 - 36s_2^2$$

$$b_6 = (6qp^2 + 6(q^2 + q + 20)p + 120q + 192)s_2 + (12p^2 + 24(q + 1)p + 12(q^2 + 2q + 7))s_1^2 - 12(3p + 4q + 3)s_1s_2 - 160s_3 + 24s_2^2$$

$$b_7 = 48s_3 + 12(q + p + 1)s_1s_2 - 12s_2^2$$

$$b_8 = 3s_2^2$$

End If

## 37.15 Noncentral Distribution of Pillai's V

The Lawley-Hotelling generalized  $T_0^2$  and Pillai's  $V$  statistic, defined respectively by

$$T_0^2 = \text{ntr}(AB^{-1}), \quad V = \text{ntr}(A(A+B)^{-1}), \quad (37.15.1)$$

have been suggested as alternatives to Wilk's criterion for testing multivariate linear hypotheses. Here  $A$  and  $B$  are independent  $p \times p$  Wishart matrices on  $q$  and  $n$  degrees of freedom respectively. [Davis \(1968, 1970b\)](#) has found the moments for both criteria and has given recurrence relations for them, together with a formula relating the moments for  $T_0^2$  and  $V$ .

### 37.15.1 Pillai's V, Central Moments

The moments of Pillai's  $V$  all exist. They can be obtained from equation [37.14.2](#) by replacing  $n_2$  with  $p - n_1 - n_2 + 1$  and multiplying by  $(-1)^r$ , ( $r = 1, 2, \dots$ ).

Let  $m = (n_1 - p - 1)/2$  and  $n = (n_2 - p - 1)/2$ ,  $s = p$

$$\mu_1 = s(2m + s + 1)/(2(m + n + s + 1)) \quad (37.15.2)$$

$$\mu_2 = \frac{s(2m + s + 1)(2n + s + 1)(2m + 2n + s + 2)}{(4(m + n + s + 1)^2(m + n + s + 2)(2m + 2n + 2s + 1))} \quad (37.15.3)$$

$$m_1 = \frac{\mu_1}{p}, \quad m_2 = \frac{\mu_2}{p^2}, \quad a = \frac{m_1(m_1 - m_1^2 - m_2)}{m_2}, \quad b = \frac{a(1 - m_1)}{m_1}, \quad w = \frac{V}{pn_2} \quad (37.15.4)$$

Then  $w$  is approximately distributed as  $I_w(a, b)$ . Let  $u_\alpha$  be the  $\alpha$  quantile of a beta distribution with  $a$  and  $b$  degrees of freedom. Then  $V_\alpha \approx u_\alpha p n_2$ .

### 37.15.2 Pillai's V, Special cases

$$V(1, n_1, n_2, x) = I(\frac{1}{2}n_1, \frac{1}{2}n_2; x)$$

pdf for  $p = 2$  (Davis 1970):

$$f_{n_1, n_2}(V) = \frac{(\frac{1}{2}V)^{n_1-1}(1 - \frac{1}{2}V)^{n_2-3}}{2B(n_1, n_2 - 1)} {}_2F_1 \left( 1, \frac{1}{2}(3 - n_2); \frac{1}{2}(n_1 + 1); r^2 \right), \quad (0 < V < 1). \quad (37.15.5)$$

$$f_{n_1, n_2}(V) = \frac{(\frac{1}{2}V)^{n_1-3}(1 - \frac{1}{2}V)^{n_2-1}}{2B(n_2, n_1 - 1)} {}_2F_1 \left( 1, \frac{1}{2}(3 - n_1); \frac{1}{2}(n_2 + 1); r^{-2} \right), \quad (1 < V < 2). \quad (37.15.6)$$

where  $r = V/(2 - V)$ . These functions reduce to polynomials in  $V$  for odd  $n_2 \geq 3$  and odd  $n_1 \geq 3$ , respectively.

### 37.15.3 Pillai's V, Other Properties

$$V(p, n_1, n_2; x) = V(n_1, p, n_2 - p; x)$$

$$V(p, n_1, n_2; x) = 1 - V(p, n_2, n_1; p - x)$$

$$f_{n_1, n_2}(V) = f_{n_2, n_1}(m - V)$$

### 37.15.4 Pillai's V, Noncentral distribution: GLM

' Fujikoshi (1974), Ann. Inst. Math. Statist., 26, p. 289

$$F_{V,GLM}(p, m, n; x; \Omega) = F_{\chi^2}(pm, x; s1) + \frac{1}{4n} \sum_{k=0}^4 a_k F_{\chi^2}(pm + 2k, x; s1) + \frac{1}{96n^2} \sum_{k=0}^8 b_k F_{\chi^2}(pm + 2k, x; s1) + O(n^{-3}) \quad (37.15.7)$$

$$l_0 = (3f - 8)g^2 + 4g + 4(f + 2)$$

$$l_1 = -12fg^2$$

$$l_2 = 6(3f + 8)g^2$$

$$l_3 = -4((3f + 16)g^2 + 4g + 4(f + 2))$$

$$l_4 = 3((f + 8)g^2 + 4g + 4(f + 2))$$

$$a_0 = -fg$$

$$a_1 = 2fg$$

$$a_2 = -fg + 4g\lambda_1 + 4\lambda_2$$

$$a_3 = -4g\lambda_1$$

$$a_4 = -4\lambda_2$$

$$b_0 = fl_0$$

$$b_1 = fl_1$$

$$b_2 = fl_2 + 2l_1\lambda_1 - 24fg\lambda_2$$

$$b_3 = fl_3 + 4l_2\lambda_1 + 48(f + 4)g\lambda_2 + 128\lambda_3$$

$$b_4 = fl_4 + 6l_3\lambda_1 + 48(g^2 - 2)\lambda_1^2 - 96(g + 1)\lambda_2 + 96g\lambda_1\lambda_2 + 48\lambda_2^2$$

$$b_5 = 8(l_4\lambda_1 - 12(g^2 + 2)\lambda_1^2 - 6((f + 12)g + 4)\lambda_2 - 12g\lambda_1\lambda_2 - 48\lambda_3)$$

$$b_6 = 8(6(g^2 + 6)\lambda_1^2 + 3((f + 20)g + 12)\lambda_2 - 12g\lambda_1\lambda_2 - 16\lambda_3 - 12\lambda_2^2)$$

$$b_7 = 96(g\lambda_1\lambda_2 + 4\lambda_3)$$

$$b_8 = 48\lambda_2^2$$

### 37.15.5 Pillai's V, Noncentral distribution: CORR

$$F_{V,CORR}(p, m, n; x; P) = F_{\chi^2}(pm, x; s1) + \frac{1}{4n} \sum_{k=0}^4 a_k F_{\chi^2}(pm + 2k, x; s1) + \frac{1}{96n^2} \sum_{k=0}^8 b_k F_{\chi^2}(pm + 2k, x; s1) + O(n^{-3}) \quad (37.15.8)$$

$$a_0 = -fg - 4\lambda_2$$

$$a_1 = 2fg$$

$$a_2 = -fg + 4g\lambda_1 + 8\lambda_2$$

$$a_3 = -4g\lambda_1$$

$$a_4 = -4\lambda_2 \quad b_0 = fl_0 + 24fg\lambda_2 - 128\lambda_3 + 48\lambda_2^2$$

$$b_1 = fl_1 - 48fg\lambda_2$$

$$b_2 = fl_2 + 2l_1\lambda_1 + 96\lambda_1^2 - 24(qp^2 + q(q + 1)p - 4)\lambda_2 - 96g\lambda_1\lambda_2 - 192\lambda_2^2$$

$$b_3 = fl_3 + 4l_2\lambda_1 + 96(qp^2 + (q^2 + q + 4)p + 4(q + 1))\lambda_2 + 96g\lambda_1\lambda_2 + 640\lambda_3$$

$$b_4 = fl_4 + 6l_3\lambda_1 + 48(p^2 + 2(q + 1)p + q^2 + 2q - 3)\lambda_1^2 - 24(qp^2 + (q^2 + q + 12)p + 4(3q + 5))\lambda_2 + 192g\lambda_1\lambda_2 + 288\lambda_2^2$$

$$b_5 = 8l_4\lambda_1 - 96(p^2 + 2(q + 1)p + q^2 + 2q + 3)\lambda_1^2 - 48(qp^2 + (q^2 + q + 12)p + 4(3q + 5))\lambda_2 - 192g\lambda_1\lambda_2 - 768\lambda_3$$

$$\begin{aligned}b_6 &= 48(p^2 + 2(q+1)p + q^2 + 2q + 7)\lambda_1^2 + 24(qp^2 + (q^2 + q + 20)p + 4(5q + 8))\lambda_2 - 96g\lambda_1\lambda_2 - 128\lambda_3 - 192\lambda_2^2 \\b_7 &= 96(g\lambda_1\lambda_2 + 4\lambda_3) \\b_8 &= 48\lambda_2^2\end{aligned}$$

## 37.16 Noncentral Distribution of Bartlett's M (2 samples)

### 37.16.1 Definition

Let  $x_1, \dots, x_{N_1}$  be a random sample from a  $N_p(\mu_1, \Sigma_1)$  population and let  $y_1, \dots, y_{N_2}$  be a random sample from a  $N_p(\mu_2, \Sigma_2)$  population. The null hypothesis of equal covariance matrices (ECM) may be written

$$H_{ECM} : \Sigma_1 = \Sigma_2 = \Sigma, \quad (37.16.1)$$

with  $\Sigma, \mu_1, \mu_2$  unrestricted. The Bartlett  $M$  test (or the modified likelihood ratio test) rejects  $H_{ECM}$  for small values of

$$\Lambda_{ECM} = \frac{|A_1^{n_1/n}||A_2^{n_2/n}|}{|A_1 + A_2|}, \quad \text{where} \quad (37.16.2)$$

$$n_i = N_i - 1, n = n_1 + n_2, \quad A_1 = \sum_{i=1}^{N_1} (x_i - \bar{x})(x_i - \bar{x})^T, \quad A_2 = \sum_{i=1}^{N_2} (y_i - \bar{y})(y_i - \bar{y})^T, \quad (37.16.3)$$

and  $\bar{x}$  and  $\bar{y}$  are the respective sample means. See [Muirhead \(1982\)](#).

The non-central distribution of  $\Lambda_{ECM}$  (i.e. the distribution of  $\Lambda_{ECM}$  when  $\Sigma_1 \neq \Sigma_2$ ) is determined by the quantities  $p, n_1, n_2$ , and  $\Delta = \text{diag}(\delta_1, \dots, \delta_p)$  where  $\delta_1, \dots, \delta_p$  are the eigenvalues of  $\Sigma_1 \Sigma_2^{-1}$ . See Subrahmaniam (1975), Pillai and Nagarsenker (1972) and Manoukian (1986).

### 37.16.2 Density and CDF

---

Function **NoncentralBartlettsMDist**(*x* As mpNum, *p* As mpNum, *m* As mpNum, *n* As mpNum, *Omega* As mpNum, *Output* As String) As mpNumList

---

**NOT YET IMPLEMENTED**

---

The function **NoncentralBartlettsMDist** returns pdf, CDF and related information for the noncentral WilksLambda(CORR)-distribution

**Parameters:**

*x*: A real number

*p*: An integer greater 0, representing the number of variates

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section [32.1.3.1](#) for the options for *Output*. Algorithms and formulas are given below.

#### 37.16.2.1 Saddlepoint approximation

[Butler & Wood \(2005\)](#) propose the following approximation

$$F_{\Lambda, ECM}(p, m, n; y; P) \approx 1 - \Phi(r) + \phi(r)(u^{-1} - r^{-1}) \quad (37.16.4)$$

where  $\Phi(\cdot)$  and  $\phi(\cdot)$  denote the cdf (see section 32.11.2.2) and pdf (see section 32.11.2.1) of the normal distribution, respectively,  $r = \text{sgn}(s)\sqrt{2(sy - K(s))}$ ,  $u = s\sqrt{K''(s)}$ , and  $s$  is the solution to the saddlepoint equation

$$K'(s) = y. \quad (37.16.5)$$

$K(s)$  denotes the cumulant generating function and is given by

$$K(s) = \ln \left[ \frac{\Gamma_p(\frac{1}{2}n)\Gamma_p(\frac{1}{2}n_1(1+2s/n))\Gamma_p(\frac{1}{2}n_2(1+2s/n))}{\Gamma_p(\frac{1}{2}n_1)\Gamma_p(\frac{1}{2}n_2)\Gamma_p(\frac{1}{2}n(1+2s/n))} \right] \quad (37.16.6)$$

$$+ \ln \left[ |\Delta|^{n_1 s / n_2} {}_2F_1 \left( s, \frac{n_1}{2} \left( 1 + \frac{2s}{n} \right) \frac{n}{2} \left( 1 + \frac{2s}{n} \right); \frac{n}{2} \left( 1 + \frac{2s}{n} \right); I_p - \Delta \right) \right] \quad (37.16.7)$$

and its first derivative,  $K'(s)$ , is given by

$$K'(s) = \sum_{i=1}^p \left[ \psi \left( \frac{1}{2}n + s - \frac{1}{2}(i-1) \right) - \psi \left( \frac{1}{2}(n+m) + s - \frac{1}{2}(i-1) \right) \right] \quad (37.16.8)$$

$$+ \frac{\partial}{\partial s} \ln \left[ |\Delta|^{n_1 s / n_2} {}_2F_1 \left( s, \frac{n_1}{2} \left( 1 + \frac{2s}{n} \right) \frac{n}{2} \left( 1 + \frac{2s}{n} \right); \frac{n}{2} \left( 1 + \frac{2s}{n} \right); I_p - \Delta \right) \right] \quad (37.16.9)$$

where  $\Gamma_p(\cdot)$  is the multivariate gamma function (see section 35.1.1),  $\psi(\cdot)$  is the digamma function (see section 13.7.2), and  ${}_2F_1(\cdot, \cdot, \cdot, X)$  is the Gauss hypergeometric function of matrix argument (see section 35.1.3). The saddlepoint equation (37.13.27) needs to be evaluated numerically. Also, the computation of  $K''(s)$  is performed using a numerical derivative of  $K'(s)$ .

### 37.16.3 Quantiles

---

Function **NoncentralBartlettsMDistInv**(*Prob* As mpNum, *p* As mpNum, *m* As mpNum, *n* As mpNum, *Omega* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **NoncentralBartlettsMDistInv** returns quantiles and related information for the noncentral WilksLambda(CORR)-distribution

**Parameters:**

*Prob*: A real number between 0 and 1.

*p*: An integer greater 0, representing the number of variates

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given below.

### 37.16.4 Properties

---

Function **NoncentralBartlettsMInfo**(*p* As mpNum, *m* As mpNum, *n* As mpNum, *Omega* As mpNum, *Output* As String) As mpNumList

---

NOT YET IMPLEMENTED

---

The function **NoncentralBartlettsMInfo** returns moments and related information for the noncentral WilksLambda(CORR)-distribution

**Parameters:**

*p*: An integer greater 0, representing the number of variates

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.3 for the options for *Output*. Algorithms and formulas are given below.

#### 37.16.4.1 Moments

The noncentral moments of  $W_{CORR}$  are specified in Theorem 11.2.6 of [Muirhead \(1982\)](#) as

$$E(W^s) = \frac{\Gamma_{p_1}(n/2)\Gamma_{p_1}((n-p_2)/2+s)}{\Gamma_{p_1}(n/2+s)\Gamma_{p_1}((n-p_2)/2)} \times |I_{p_1} - P^2|^{n/2} {}_2F_1\left(\frac{n}{2}, \frac{n}{2}; \frac{n}{2} + s; P^2\right) \quad (37.16.10)$$

where  $P = \text{diag}\{\rho_1, \dots, \rho_{p_1}\}$  contains the population canonical correlations,  $\Gamma_p(\cdot)$  is the multivariate gamma function (see section 35.1.1) and  ${}_2F_1(\cdot, \cdot, \cdot, X)$  is the Gauss hypergeometric function of matrix argument (see section 35.1.3). See also [Butler & Wood \(2002\)](#).

#### 37.16.5 Random Numbers

---

Function **NoncentralBartlettsMDistRan**(*Size* As *mpNum*, *p* As *mpNum*, *m* As *mpNum*, *n* As *mpNum*, *Generator* As String, *Omega* As *mpNum*, *Output* As String) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **NoncentralBartlettsMDistRan** returns random numbers following a noncentral WilksLambda(CORR)-distribution

**Parameters:**

*Size*: A positive integer up to  $10^7$

*p*: An integer greater 0, representing the number of variates

*m*: A real number greater 0, representing the numerator degrees of freedom

*n*: A real number greater 0, representing the denominator degrees of freedom

*Generator*: A string describing the random generator

*Omega*: An array of real numbers representing the noncentrality parameter

*Output*: A string describing the output choices

See section 32.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

# Chapter 38

## Examples: Discrete Distribution Functions

## 38.1 Noncentral Distribution of Mann-Whitney's U (with Stratification)

### 38.1.1 Definition

See [Wang et al. \(2003\)](#)

See [Mehrotra et al. \(2010\)](#)

See [Divine et al. \(2010\)](#)

See [Zhao \(2006\)](#)

See [Tang \(2011\)](#)

See [Zhao et al. \(2008\)](#)

Although Mann and Whitney developed the MW test under the assumption of continuous responses with the alternative hypothesis being that one distribution is stochastically greater than the other, there are many other ways to formulate the null and alternative hypotheses such that the MW test will give a valid test.

A very general formulation is to assume that:

1. All the observations from both groups are independent of each other,
2. The responses are ordinal (i.e. one can at least say, of any two observations, which is the greater),
3. The distributions of both groups are equal under the null hypothesis, so that the probability of an observation from one population (X) exceeding an observation from the second population (Y) equals the probability of an observation from Y exceeding an observation from X. That is, there is a symmetry between populations with respect to probability of random drawing of a larger observation.
4. Under the alternative hypothesis, the probability of an observation from one population (X) exceeding an observation from the second population (Y) (after exclusion of ties) is not equal to 0.5. The alternative may also be stated in terms of a one-sided test, for example:  $P(X > Y) + 0.5P(X = Y) > 0.5$ .

Let  $x_1, \dots, x_m$  and  $y_1, \dots, y_n$  be two sets of measurements, which we denote by  $X$  and  $Y$ . The test criterion  $U$  of the Mann-Whitney test is then

$$U = \sum_{i=1}^m \sum_{j=1}^n \text{sgn}(x_i - y_j) \quad (38.1.1)$$

### 38.1.2 Central distribution

#### 38.1.2.1 Exact Central distribution

The null distribution of the MW test can be calculated as follows: Let  $p_{n,m}(u)$  denote the probability that  $U = u$  in samples of size  $n$  and  $m$ . Then ([Zimmermann, 1985b](#))

$$(m+n)p_{n,m}(u) = np_{n-1,m}(u-m) + mp_{n,m-1}(u) = np_{n-1,m}(u) + mp_{n,m-1}(u-n) \quad (38.1.2)$$

#### 38.1.2.2 Cumulants

The cumulants of  $U$  are given by ([Robillard, 1972](#)):

$$\begin{aligned} \kappa_{2j} &= \frac{B_{2j}}{2j} \left[ \sum_{s=m+1}^{m+n} s^{2j} - \sum_{s=1}^n s^{2j} \right] \\ &= \frac{B_{2j}}{2j(2j+1)} [B_{2j+1}(n+m+1) + B_{2j+1} - B_{2j+1}(m+1) - B_{2j+1}(n+1)] \end{aligned} \quad (38.1.3)$$

and  $\kappa_{2j+1} = 0$ ,  $j \geq 1$ , and  $B_{2j}$  and  $B_{2j}(x)$  are the Bernoulli numbers and polynomials, respectively, of degree  $2j$  (see section 14.2).

### 38.1.3 Confidence interval for a location parameter

Let  $(D_1, \dots, D_{nm})$  be the ordered statistics of all differences  $x_i - y_j$  from samples of size  $m$  and  $n$ . The median of the  $D_i$  is an estimator of the location parameter  $\theta$  of the shift alternative  $F_1(x) = F_0(x - \theta)$ . A confidence interval for  $\theta$  can be calculated from  $\Pr(D_{r+1} < \theta < D_{nm-r}) \geq 1 - \alpha$ , where  $r$  is the  $\alpha/2$  quantile of the distribution of  $U$ .

### 38.1.4 Noncentral distribution

The calculation of the exact noncentral distribution of  $p = U/(mn)$  requires that either  $F_0$  (the distribution of  $X$ ) and  $F_1$  (the distribution of  $Y$ ) are explicitly known, or that  $F_1$  can be expressed as function of  $F_0$ .

#### 38.1.4.1 Normal alternatives

Consider the situation where random variables  $X_1, \dots, X_m$  and  $Y_1, \dots, Y_n$  are normally distributed with means  $\mu_X$  and  $\mu_Y$ , respectively, and common variance  $\sigma^2$ , all  $m+n$  random variables being mutually independent and  $d = (\mu_Y - \mu_X)/\sigma$ .

Let  $\mathbf{U} = (U_1, \dots, U_{m+n})$ ,  $U_1 < \dots < U_{m+n}$ , denote the order statistics of the random variables  $(X_1, \dots, X_m)$ ,  $(Y_1, \dots, Y_n)$ , and let  $\mathbf{Z} = (Z_1, \dots, Z_{m+n})$  denote a random vector of zeros and ones, where the  $i$ th component  $Z_i$  is 0 (or 1) if  $U_i$  is an  $X$  (or  $Y$ ). Denote by  $\phi(x - \theta)$  the normal density with mean  $\theta$  and variance 1 (see section 32.11.2.1). If  $\mathbf{z} = (z_1, \dots, z_{m+n})$  is a fixed vector of zeros and ones, the probability of the rank order  $z$ ,  $\Pr[\mathbf{Z} = \mathbf{z}]$ , is given by

$$P_{m,n}(\mathbf{z}|d) = m!n! \int_R \cdots \int \prod_{i=1}^{m+n} \phi(t_i - z_i d) dt_i, \quad (38.1.4)$$

where the region of integration  $R$  is  $-\infty < t_1 \leq t_2 \leq \dots \leq t_{m+n} < \infty$ . Milton (1970) describes a  $p$ -dimensional midpoint algorithm that economizes the number of arithmetic operations required to evaluate equation (38.1.4), corrects for effects of the edges of the region of integration and involves no high-degree quadrature formulas.

The distribution of  $U$  is obtained by calculating for all  $\binom{N}{m}$  vectors  $\mathbf{z} = (z_1, \dots, z_{m+n})$  the criterion  $U = t(\mathbf{z}) = \sum_{i=1}^{n+m} z_i r_i$  and summing up for each  $x$  the corresponding probabilities  $P_{m,n}(\mathbf{z})$ :

$$\Pr[U = x] = \sum_{t(\mathbf{z})=x} P_{m,n}(\mathbf{z}). \quad (38.1.5)$$

#### 38.1.4.2 Lehmann alternatives

Lehmann alternatives are of the form  $F_1(x) = [F_0(x)]^k$  or  $F_1(x) = 1 - [1 - F_0(x)]^k$ . The exact distribution under this alternative can be calculated using recurrence relations, similar to the null-distribution (Shorack, 1966): Let  $p_{n,m}(u)$  denote the probability that  $U = u$  in samples of size  $n$  and  $m$ . Then

$$(km + n)p_{n,m}(u) = np_{n-1,m}(u - m) + kmp_{n,m-1}(u) \quad \text{for } F_1 = (F_0)^k \quad (38.1.6)$$

$$(km + n)p_{n,m}(u) = np_{n-1,m}(u) + kmp_{n,m-1}(u - n) \quad \text{for } F_1 = 1 - (1 - F_0)^k \quad (38.1.7)$$

This recursive procedure allows the calculation of the exact noncentral distribution also for larger samples. Under Lehmann alternatives, rank order probabilities can be expressed in closed form for  $F_1 = (F_0)^k$ : let  $S_1, \dots, S_n$  denote the ranks of  $Y$  in the combined sample, e.g.  $\Pr[S_1 = 3, S_2 = 5] = P_{3,2}(0, 0, 1, 0, 1)$ . Then

$$\Pr[S_1 = s_1, \dots, S_n = s_n] = k^n \frac{n!m!}{\Gamma(n+m+1+n(k-1))} \prod_{j=1}^n \frac{\Gamma(s_j + j(k-1))}{\Gamma(s_j + (j-1)(k-1))}. \quad (38.1.8)$$

The first 2 Moments under the alternative  $F_1 = (F_0)^k$  are given by:

$$\mu_1 = k/(k+1) \quad (38.1.9)$$

$$\mu_2 = \frac{k}{mn(k+1)^2} \left[ \frac{m-1}{k+2} + \frac{k(n-1)}{2k+1} + 1 \right]. \quad (38.1.10)$$

### 38.1.4.3 First 4 moments under the general alternative

The first 4 moments under the alternative are given by (Sundrum, 1954):

$$\mu_1 = p \quad (38.1.11)$$

$$\mu_2 = (p^2 + p - q - r)/mn + (q - p^2)/n + (r - p^2)/m + p^2 \quad (38.1.12)$$

$$\begin{aligned} \mu_3 = & 6(p^3 + u - pq - pr)/mn + (2p^3 + s - 3pq)/n^2 + (2p^3 + t - 3pr)/m^2 \\ & + 3(3pq + 2(pr - u - p^3) + q - s - p^2)/mn^2 + 3(3pr + 2(pq - u - p^3) \\ & + r - t - p^2)/m^2n + (4p^3 + 3p^2 + p + 6u + 2(s + t) - 3(q + r)(1 + 2p))/m^2n^2 \end{aligned} \quad (38.1.13)$$

$$\begin{aligned} \mu_4 = & 3(q - p^2)2/n^2 + 6(q - p^2)(r - p^2)/mn + 3(r - p^2)2/m^2 \\ & + (12qp^2 + a - 4sp - 3q^2 - 6p^4)/n^3 + (12rp^2 + b - 4tp - 3r2 - 6p^4)/m^3 \\ & + 6(7(rp^2 - p^4) + 12qp^2 + qp + 2(w - sp) - 3(q^2 + qr) - 8up - p^3)/mn^2 \\ & + (42qp^2 + 72rp^2 + 6rp + 12x - 42p^4 - 18r2 - 18qr - 12tp - 48up - 6p^3)/m^2n \\ & + 6(6(p^4 - rp^2) + 3(q^2 - qp) - 12qp^2 + 4sp + 8up + 2(p^3 + qr - w) + s - a)/mn^3 \\ & + (36p^4 + 18r2 + 12qr - 72rp^2 - 36qp^2 + 24tp - 6b + 48up \\ & - 12x + 12p^3 - 18rp + 6t)/m^3n \\ & + (105p^4 + 42p^3 + 3p^2 + 33q^2 + 33r2 + 54qr - 174qp^2 - 174rp^2 - 42pq \\ & - 42pr + 36sp + 36tp + 192up - 36w - 36x + 6v + 36u)/m^2n^2 \\ & + (132qp^2 + 108rp^2 - 66p^4 - 33q^2 - 36qr - 18r2 - 44sp - 24tp + 11a \\ & - 144up + 36w + 24x - 6v - 36p^3 - 36u - 7p^2 + 54pq + 36pr - 18s + 7q)/m^2n^3 \\ & + (132rp^2 + 108qp^2 - 66p^4 - 33r2 - 36qr - 18q^2 - 44tp - 24sp + 11b \\ & - 144up + 24w + 36x - 6v - 36p^3 - 36u - 7p^2 + 54pr + 36pq - 18t + 7r)/m^3n^2 \\ & + (6(3(q^2 + r2) - 12((q + r)p^2) + 4(p^3 + qr + sp + tp - w - x) - (a + b - v) + 16up \\ & - 6(p^4 + pq + pr + u) + 2(s + t)) - 7(q + r - p^2) + p)/m^3n^3 \end{aligned} \quad (38.1.14)$$

The parameters  $p, q, r, s, t, v, u, a, b, w$  and  $x$  can be calculated from the following rank order probabilities:

$$\begin{aligned}
 p &= P_{1,1}(0, 0) & (38.1.15) \\
 q &= P_{2,1}(0, 0, 1) \\
 r &= P_{1,2}(0, 1, 1) \\
 s &= P_{3,1}(0, 0, 0, 1) \\
 t &= P_{1,3}(0, 1, 1, 1) \\
 v &= P_{2,2}(0, 0, 1, 1) \\
 u &= v + (1/4)P_{2,2}(0, 1, 0, 1) \\
 a &= P_{4,1}(0, 0, 0, 0, 1) \\
 b &= P_{1,4}(0, 1, 1, 1, 1) \\
 w &= 2a + (2/3)P_{3,2}(0, 0, 1, 0, 1) + (1/6)P_{3,2}(0, 1, 0, 0, 1) \\
 x &= 2b + (2/3)P_{2,3}(0, 1, 0, 1, 1) + (1/6)P_{2,3}(0, 1, 1, 0, 1)
 \end{aligned}$$

#### 38.1.4.4 Estimation of rank order probabilities from the sample

The rank order probabilities which are required for the calculation of the first 4 moments can be estimated from the sample as follows: Let  $U_i$  be the number of  $Y$ 's in the sample greater than  $X_{(i)}$ , where  $X_{(i)}$  is the  $i$ th ordered values of the  $X$ 's amongst themselves. Then

$$P_{1,1}(0, 1) = \frac{1}{mn} \sum_{i=1}^m U_i \quad (38.1.16)$$

$$P_{2,1}(0, 0, 1) = \frac{2}{mn(m-1)} \sum_{i=1}^m (i-1)U_i \quad (38.1.17)$$

$$P_{3,1}(0, 0, 0, 1) = \frac{3}{mn(m-1)(m-2)} \sum_{i=1}^m (i-1)(i-2)U_i \quad (38.1.18)$$

$$P_{4,1}(0, 0, 0, 0, 1) = \frac{4}{mn(m-1)(m-2)(m-3)} \sum_{i=1}^m (i-1)(i-2)(i-3)U_i \quad (38.1.19)$$

$$P_{1,2}(0, 1, 1) = \frac{1}{mn(n-1)} \sum_{i=1}^m U_i(U_i - 1) \quad (38.1.20)$$

$$P_{1,3}(0, 1, 1, 1) = \frac{1}{mn(n-1)(n-2)} \sum_{i=1}^m U_i(U_i - 1)(U_i - 2) \quad (38.1.21)$$

$$P_{1,4}(0, 1, 1, 1, 1) = \frac{1}{mn(n-1)(n-2)(n-3)} \sum_{i=1}^m U_i(U_i - 1)(U_i - 2)(U_i - 3) \quad (38.1.22)$$

$$P_{2,2}(0, 0, 1, 1) = \frac{2}{mn(m-1)(n-1)} \sum_{i=1}^m (i-1)U_i(U_i - 1) \quad (38.1.23)$$

$$P_{3,2}(0, 0, 0, 1, 1) = \frac{3}{mn(m-1)(m-2)(n-1)} \sum_{i=1}^m (i-1)(i-2)U_i(U_i - 1) \quad (38.1.24)$$

$$P_{2,3}(0, 0, 1, 1, 1) = \frac{2}{mn(m-1)(n-1)(n-2)} \sum_{i=1}^m (i-1)U_i(U_i - 1)(U_i - 2) \quad (38.1.25)$$

$$P_{2,2}(0, 1, 0, 1) = \frac{4}{mn(m-1)(n-1)} \sum_{i=1}^m \sum_{j=i+1}^m (U_i - U_j) U_j \quad (38.1.26)$$

$$P_{3,2}(0, 0, 1, 0, 1) = \frac{12}{mn(m-1)(m-2)(n-1)} \sum_{i=1}^m \sum_{j=i+1}^m (i-1)(U_i - U_j) U_j \quad (38.1.27)$$

$$P_{3,2}(0, 1, 0, 0, 1) = \frac{12}{mn(m-1)(m-2)(n-1)} \sum_{i=1}^m \sum_{j=i+1}^m (i-j-1)(U_i - U_j) U_j \quad (38.1.28)$$

$$P_{2,3}(0, 1, 0, 1, 1) = \frac{6}{mn(m-1)(n-1)(n-2)} \sum_{i=1}^m \sum_{j=i+1}^m (U_i - U_j) U_j (U_j - 1) \quad (38.1.29)$$

$$P_{2,3}(0, 1, 0, 1, 1) = \frac{6}{mn(m-1)(n-1)(n-2)} \sum_{i=1}^m \sum_{j=i+1}^m (U_i - U_j) U_j (U_j - 1) \quad (38.1.30)$$

$$P_{2,3}(0, 1, 1, 0, 1) = \frac{6}{mn(m-1)(n-1)(n-2)} \sum_{i=1}^m \sum_{j=i+1}^m (U_i - U_j) (U_i - U_j - 1) U_j \quad (38.1.31)$$

### 38.1.4.5 Confidence interval for $\Pr(X > Y)$

If  $F_0$  and  $F_1$  are known, or if  $F_1$  is a function of  $F_0$ , then the (noncentral) distribution of  $U$  depends only on  $m, n$  and  $p$ , and we can write the pdf as  $f(i|m, n; p)$ . The positive solutions of the two equations

$$\sum_{i=0}^x f(i|m, n; p_U) = \alpha/2, \quad \text{and} \quad \sum_{i=x}^m f(i|m, n; p_L) = \alpha/2 \quad (38.1.32)$$

then form a confidence interval of  $p$  in the sense that

$$\Pr[p_L \leq p \leq p_U] \geq 1 - \alpha. \quad (38.1.33)$$

### 38.1.5 Power for Ordered Categorical Data

See [Kolassa \(1995\)](#)

## 38.2 Noncentral Distribution of Wilcoxon's Signed Rank Test (with Stratification)

See [Good \(2005\)](#)

See [Wang \*et al.\* \(2003\)](#)

See [Govindarajulu \(2007\)](#)

See [Shieh \*et al.\* \(2007\)](#)

See [Agresti \(2010\)](#)

### 38.2.1 Definition

We consider  $N$  continuously distributed random variables  $D_i, i = 1 \dots N$ , with common pdf  $h_0$ . In a sample  $(d_1, \dots, d_N)$  of size  $N$  let  $r_i$  be the rank of  $d_i$  in the ordered sample.

#### 38.2.1.1 The $T_N$ and $W_N$ test criteria

The test criterion of Wilcoxon's Signed Rank is  $T_N = \sum_{i=1}^N S(d_i)r_i$ , where  $S(d_i) = 1$  for  $x > 0$  and  $S(d_i) = 0$  for  $x < 0$ .  $T_N$  can assume values between 0 and  $\frac{1}{2}N(N + 1)$  in steps of 1.

Equivalent to  $T_N$  is the criterion  $W_N = \sum_{i=1}^N \text{sgn}(d_i)r_i$ .  $W_N$  can assume values between  $-\frac{1}{2}N(N + 1)$  and  $\frac{1}{2}N(N + 1)$  in steps of 2.

We have  $W_N = 2T_N - \frac{1}{2}N(N + 1)$  and  $T_N = (W_N + \frac{1}{2}N(N + 1))$ .

#### 38.2.1.2 The $\tilde{T}_N$ and $\tilde{W}_N$ test criteria

$T$  can also be written as  $T = T_1 + \tilde{T}_N$ ,  $T_1 = \sum_{i=1}^N S(d_i)$ ,  $\tilde{T}_N = \sum_{i=1}^{N-1} \sum_{j=i+1}^N S(d_i + d_j)$ .  $T_1$  is the test criterion of the Sign test (see section ...).

$\tilde{T}_N$  is the test criterion of the modified Signed Rank test.

$\tilde{T}_N$  can also be written as  $\tilde{T}_N = \sum_{i=1}^N s(d_i)(r_i - 1)$ .  $\tilde{T}_N$  can assume values between 0 and  $\frac{1}{2}N(N - 1)$  in steps of 1.

Equivalent to  $\tilde{T}_N$  is the criterion  $\tilde{W}_N = \sum_{i=1}^N \text{sgn}(d_i)(r_i - 1)$ .

$\tilde{W}_N$  can assume values between  $-\frac{1}{2}N(N - 1)$  and  $\frac{1}{2}N(N - 1)$  in steps of 2.

We have  $\tilde{W}_N = 2\tilde{T}_N - \frac{1}{2}N(N - 1)$  and  $\tilde{T}_N = \frac{1}{2}(\tilde{W}_N + \frac{1}{2}N(N - 1))$ .

#### 38.2.1.3 Parameters, expected values and hypotheses

We define  $p_1 = \Pr[D_1 > 0]$ ,  $p_2 = \Pr[D_1 + D_2 > 0]$ ,  $p_{XY} = \Pr[X > Y] - \Pr[X < Y]$ .

Here  $X$  and  $Y$  are random variables: the  $X$  are distributed like positive  $D$  with pdf  $g_0$ , and the  $Y$  like the absolute values of negative  $D$  with pdf  $g_1$  (see below).  $p_{XY}$  can assume values between  $-1$  and  $1$ . The following relationships hold ([Noether, 1967](#); [Noether & Dueker, 1990](#)):

$$p_1 = \frac{1}{2p_{XY}} \left( p_{XY} + 1 - \sqrt{(p_{XY} + 1)^2 - 4p_{XY}p_2} \right) \quad p_2 = (p_1 - p_1^2)p_{XY} + p_1, \quad p_{XY} = \frac{p_2 - p_1}{p_1 - p_1^2}. \quad (38.2.1)$$

Consistent estimators for  $p_1$  and  $p_2$  are  $\hat{p}_1 = T_1/N$  and  $\hat{p}_2 = \tilde{T}_N/(N(N - 1))$ .

The expected values of  $T_N$  and  $W_N$  are weighted sums of the parameters  $p_1$  and  $p_2$ :

$$e(T_N) = Np_1 + \frac{1}{2}N(N - 1)p_2.$$

$$e(W_N) = 2E(T_N) - \frac{1}{2}N(N + 1),$$

whereas the expected values of  $\tilde{T}_N$  and  $\tilde{W}_N$  only depend on  $p_2$ :

$$e(\tilde{T}_N) = \frac{1}{2}N(N - 1)p_2.$$

$$e(\tilde{W}_N) = 2E(\tilde{T}_N) - \frac{1}{2}N(N-1).$$

Under the null hypothesis we have:  $p_{XY} = 0$ ;  $p_1 = p_2 = \frac{1}{2}$ ;  $e(T_N) = \frac{1}{4}N(N+1)p_2$ ;  $e(W_N) = 0$ . In the special case of a shift-hypothesis, assuming a normal distribution, i.e.  $h_0 = \phi(x - \delta)$ , we have

$$p_1 = \Phi(\delta), \quad p_2 = \Phi(\delta\sqrt{2}), \quad \text{and} \quad p_{XY} = \frac{\Phi(\delta\sqrt{2}) - \Phi(\delta)}{\Phi(\delta)\Phi(-\delta)}. \quad (38.2.2)$$

The two-sided test is consistent against alternatives with  $|p_2 - \frac{1}{2}| \neq 0$ . In particular, the test is consistent against alternatives with a median of zero (i.e.  $p_1 = \frac{1}{2}$ ) if  $p_{XY} \neq 0$ , which implies a certain amount of asymmetry. If the distribution of the  $D$  is symmetric around  $c \neq 0$ , then  $|p_2 - \frac{1}{2}|c \neq 0$ , and a right-sided test is consistent against alternatives with  $c > 0$ , a left-sided against  $c < 0$ , and a two-sided against  $c \neq 0$  (see 38.2.2).

### 38.2.2 Central Density

Let  $p_N(w)$  denote the probability  $\Pr[W_N = w]$  in a sample of size  $N$ . Then the following recurrence relation holds (Zimmermann, 1985a) :

$$p_N(w) = \frac{1}{2} (p_{N-1}(w) + p_{N-1}(w - N)). \quad (38.2.3)$$

#### 38.2.2.1 Cumulants of $W_N$

The cumulants of  $W$  are given by (Fellingham & Stoker, 1964) :

$$\kappa_{2j}(W_N) = \frac{2^{2j}(2^{2j}-1)B_{2j}}{2j} \sum_{i=1}^N r_i^{2j} = \frac{2^{2j}(2^{2j}-1)B_{2j}}{2j} \frac{B_{2j+1}(N+1) - B_{2j+1}}{2j+1}, \quad \text{and} \quad (38.2.4)$$

$$\kappa_{2j+1}(W_N) = 0, \quad \text{for } j \geq 0. \quad (38.2.5)$$

In particular,  $\kappa_1(W_N) = N(N+1)/4$ , and  $\kappa_2(W_N) = N(N+1)(2N+1)/24$ , and  $B_{2j}$  and  $B_{2j}(x)$  are the Bernoulli numbers and polynomials, respectively, of degree  $2j$  (see section 14.2).

#### 38.2.2.2 Cumulants of $\tilde{W}_N$

The cumulants of  $\tilde{W}_N$  are given by

$$\kappa_{2j}(\tilde{W}_N) = \frac{2^{2j}(2^{2j}-1)B_{2j}}{2j} \sum_{i=1}^N (r_i - 1)^{2j} = \frac{2^{2j}(2^{2j}-1)B_{2j}}{2j} \frac{B_{2j+1}(N) - B_{2j+1}}{2j+1}, \quad \text{and} \quad (38.2.6)$$

$$\kappa_{2j+1}(\tilde{W}_N) = 0, \quad \text{for } j \geq 0. \quad (38.2.7)$$

In particular,  $\kappa_1(\tilde{W}_N) = N(N-1)/4$ , and  $\kappa_2(\tilde{W}_N) = N(N-1)(2N-1)/24$ , and  $B_{2j}$  and  $B_{2j}(x)$  are the Bernoulli numbers and polynomials, respectively, of degree  $2j$  (see section 14.2).

### 38.2.3 Confidence interval for the median

We assume that the differences  $D_i$  are distributed symmetrically around then median  $c$ . If all  $\frac{1}{2}N(N+1)$  values  $\frac{1}{2}(D_i + D_j)$  are ordered in the sample, and we denoted the ordered statistics as  $(D_1, \dots, D_{N(N+1)/2})$ , then a confidence interval for  $c$  is obtained as

$$\Pr[D_{r+1} < c < D_{N(N+1)/2} - r] \geq 1 - \alpha, \quad (38.2.8)$$

where  $r$  denotes the  $\alpha/2$  quantile of the distribution of  $W_N$ .

### 38.2.4 Noncentral distribution

We consider a sample of size of the random variable  $D$ , which can assume positive or negative values. This sample can be partitioned in  $N + 1$  different ways in sub-samples of size  $m$  (number of positive values) and  $N - m$  (number of negative values), where  $m = 0, \dots, N$ .

#### 38.2.4.1 Method 1

Let  $Z_i = \frac{1}{2}(\text{sgn}(X_i) + 1)$ . For each partition with  $m$  positive values there exist  $\binom{N}{m}$  different permutations  $Z_{N-m,m} = (Z_1, \dots, Z_N)$ , which represent a random vector of  $N - m$  zeros and  $m$  ones. We define

$$f_0(x) = h_0(x) \text{ for } x > 0, \text{ and } 0 \text{ otherwise.}$$

$$f_1(x) = h_0(-x) \text{ for } x < 0, \text{ and } 0 \text{ otherwise.}$$

Then for each fixed vector  $\mathbf{z} = (z_1, \dots, z_N)$  of zeros and ones, the (unconditional) probability of the rank order  $z$ ,  $\Pr[\mathbf{Z} = \mathbf{z}]$ , is given by

$$P_{N-m,m}(\mathbf{z}) = N! \int_R \cdots \int \prod_{i=1}^N f_{z_i}(t_i) dt_i, \quad (38.2.9)$$

where the region of integration  $R$  is  $-\infty < t_1 \leq t_2 \leq \cdots \leq t_N < \infty$  (Milton, 1970). See also Klotz (1963). The distribution of  $T_N$  is obtained by calculating for all  $\sum_{m=0}^N \binom{N}{m} = 2^N$  vectors  $\mathbf{z} = (z_1, \dots, z_N)$  the criterion  $T_N = t(\mathbf{z}) = \sum_{i=1}^N z_i r_i$  and summing up for each  $T$  the corresponding probabilities  $P_{N-m,m}(\mathbf{z})$ :

$$\Pr[T_N = x] = \sum_{t(z)=x} P_{N-m,m}(\mathbf{z}). \quad (38.2.10)$$

See Arnold (1965) and Klotz (1963).

#### 38.2.4.2 Method 2

The probability that in a sample of size  $N$  exactly  $m$  values are positive is

$$\Pr[M = m] = \binom{N}{m} p_1^m (1 - p_1)^{N-m}. \quad (38.2.11)$$

We define

$$g_0 = \frac{1}{p_1} f_0 \text{ and } g_1 = \frac{1}{1 - p_1} f_1, \text{ so that } \int_0^\infty g_0(x) dx = 1 \text{ and } \int_0^\infty g_1(x) dx = 1. \quad (38.2.12)$$

For a given number of  $m$  positive values the conditional probability  $\Pr[T_{N-m,m} = x]$  is the same as  $\Pr[U_{N-m,m} = x]$  of the Mann-Whitney U-statistic with sample sizes  $m$  and  $N - m$ , when using the distributions  $g_0$  and  $g_1$ . Since  $\Pr[M = m]$  and  $\Pr[T_{N-m,m} = x]$  are independent, the nonconditional distribution of  $T$  is then obtained as

$$\Pr[T_N = x] = \sum_{m=0}^N \Pr[M = m] \times \Pr[U_{N-m,m} = x]. \quad (38.2.13)$$

### 38.2.5 Moments of the noncentral distribution

For the raw moments of the noncentral distribution we have

$$\mu_r(T_N) = \sum_{m=0}^N \Pr[M = m] \times \mu_r(U_{N-m,m}). \quad (38.2.14)$$

This assumes using the distributions  $g_0$  and  $g_1$  for the calculation of the rank order probabilities. The positive values will be assigned to  $X$  and the negative to  $Y$  if the rank order probabilities are estimated from the sample.

### 38.2.6 Sample Size

See [Noether \(1987\)](#)

## 38.3 Noncentral Distribution of Kendall's Tau

### 38.3.1 Definition

See [Wang et al. \(2003\)](#)

Let  $(X_1, Y_1), \dots, (X_N, Y_N)$  be independent random variables, the  $X_i$  with a continuous distribution  $F_0$ , the  $Y_i$  with df  $G_0$ . Let  $R_i$  and  $S_i$  be the ranks of  $X_i$  and  $Y_i$ , respectively. Consider the Kendall rank correlation coefficient

$$\tau = \frac{1}{N(N-1)} \sum_{i=1}^N \sum_{j=1}^N \text{sgn}(R_i - R_j) \text{sgn}(S_i - S_j) \quad (38.3.1)$$

and its transformation  $T_N$  defined by

$$T_N = \frac{1}{4}(\tau + 1)N(N-1) \quad (38.3.2)$$

Then  $E(T_N) = N(N-1)/4$ ,  $\text{Var}(T_N) = N(N-1)(2N+5)/72$ , and the distribution of  $T_N$  is asymptotically normal.  $T_N$  can assume values between 0 and  $N(N-1)/2$ .

### 38.3.2 Central distribution

#### 38.3.2.1 Exact distribution

Let  $p_N(t) = \Pr[T_N = t]$ . Then the following recurrence relation holds:

$$p_N(t) = p_N(t-1) + [p_{N-1}(t) - p_{N-1}(t-N)]/N, \quad (38.3.3)$$

where  $p_N(t) = 0$  for  $t < 0$  or  $t > N(N-1)/2$ , and  $p_N(0) = 1/N!$ .

#### 38.3.2.2 Cumulants

The cumulants of  $T_N$  are given by

$$\kappa_{2j}(T_N) = \frac{B_{2j}}{2j} \sum_{s=1}^N s^{2j} = \frac{B_{2j}}{2j} \left[ \frac{B_{2j+1}(N+1) - B_{2j+1}}{2j+1} - N \right], \quad \text{and} \quad (38.3.4)$$

$$\kappa_{2j+1}(T_N) = 0, \quad \text{for } j \geq 1. \quad (38.3.5)$$

In particular,  $\kappa_1(T_N) = N(N-1)/4$ , and  $\kappa_2(T_N) = N(N-1)(2N+5)/72$ , and  $B_{2j}$  and  $B_{2j}(x)$  are the Bernoulli numbers and polynomials, respectively, of degree  $2j$  (see section 14.2).

### 38.3.3 Noncentral distribution

#### 38.3.3.1 General case

Consider  $k$  pairs of random variables  $(x_1, y_1), \dots, (x_k, y_k)$ . Let  $r_1, \dots, r_k$  be a permutation of  $1, \dots, k$  and let  $s_1, \dots, s_k$  be its reciprocal. We define as in [Snow \(1962\)](#)

$P(r_1, \dots, r_k) = \Pr[\text{if } x\text{'s are ranked in order, ranks of corresponding } y\text{'s are } r_1, \dots, r_k]$ .

$P(r_1, \dots, r_k) = \Pr[\text{if } y\text{'s are ranked in order, ranks of corresponding } x\text{'s are } s_1, \dots, s_k]$ .

Let  $H(t) = 1$  if  $t > 0$  and  $H(t) = 0$  otherwise. Then

$$P(r_1, \dots, r_k) = k!P \left[ \prod_{i=2}^k H(x_i - x_{i-1})H(y_{s_i} - y_{s_{i-1}}) \right] = k!P \left[ \prod_{i=2}^k H(y_i - y_{i-1})H(x_{r_i} - x_{r_{i-1}}) \right] \quad (38.3.6)$$

### 38.3.3.2 Bivariate normal distribution

When  $(x_i, y_i)$ ,  $i = 1, 2, \dots, k$ , are normal variables with correlation  $\rho$ , then

$$P(r_1, \dots, r_k) = k!P(U_1 > 0, U_2 > 0, \dots, U_{2k-2} > 0), \quad (38.3.7)$$

where the  $U$ 's are normal variables defined below with zero mean and correlation matrix  $R$ :

$$R = \begin{pmatrix} A & \rho B \\ \rho B' & A \end{pmatrix} \quad (38.3.8)$$

$$A_{k-1, k-1} = \begin{pmatrix} 1 & -\frac{1}{2} & 0 & 0 & \dots & \dots & \dots & \dots \\ -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & \dots & \dots & \dots \\ 0 & -\frac{1}{2} & 1 & -\frac{1}{2} & 0 & 0 & \dots & \dots \\ \vdots & \dots \\ \vdots & \dots \end{pmatrix} \quad (38.3.9)$$

$$B = (b_{ij}), \quad b_{i,j} = \frac{1}{2} (\delta_{r_{i+1}, j+1} - \delta_{r_{i+1}, j} - \delta_{r_i, j+1} - \delta_{r_i, j}) \quad (38.3.10)$$

### 38.3.3.3 First 4 moments in the general case

The first 4 moments of  $T_N$  are given as [Sundrum \(1953\)](#):

$$\mu_1 = p \quad (38.3.11)$$

$$\mu_2 = 2[p(1-p) + 2(k-p^2)(n-2)]/[n(n-1)] \quad (38.3.12)$$

$$\begin{aligned} \mu_3 = 4[(t - 18kp + 10p^3)n^2 + (6k + 2u - 6p^2 - 5t + 72kp - 34p^3)n \\ + (p + 9p^2 + 30p^3 - 12k - 4u + 6t - 72kp)]/[n(n-1)]^2 \end{aligned} \quad (38.3.13)$$

$$\begin{aligned} \mu_4 = 8[6(k-p^2)2n^4 + (6kp + 2y - 6p^3 - 16tp - 84k2 + 270kp^2 - 108p4)n^3 \\ + (1.5p^2 + 6t + 2b - 126kp - 24up - 18y + 75p^3 + 120tp + 426k2 - 1446kp^2 + 505.5p4)n^2 \\ + (14k + 12u - 15.5p^2 - 30t - 10b + 444kp + 96up + 52y - 213p^3 - 296tp - 924k2 \\ + 2988kp^2 - 943.5p4)n \\ + (p - 28k - 24u + 21p^2 + 36t + 12b - 432kp - 96up - 48y + 180p^3 + 240tp + 720k2 \\ - 2160kp^2 + 630p4)]/[n(n-1)]^3 \end{aligned} \quad (38.3.14)$$

The parameters  $p, k, u, t, b, y$  can be calculated from the following rank order probabilities, writing  $P_{(1423)}$  for the probability that the ranks of the  $Y$  occur in this order when the  $X$  are sorted in ascending order:

$$p = P_{(12)} \quad (38.3.15)$$

$$u = P_{(123)}$$

$$k = P_{(123)} + \frac{1}{4}P_{(132)}$$

$$t = 8(P_{(1234)} + P_{(1243)}) + 6P_{(1342)} + 4P_{(1324)} + 2P_{(2143)} + P_{(2413)} + P_{(1432)}$$

$$b = 15P_{(1234)} + 10P_{(1243)} + 5P_{(1324)} + 4P_{(1342)} + P_{(2143)}$$

$$\begin{aligned} y = (160P_{(12453)} + 150(P_{(12435)} + P_{(12354)}) + 125P_{(12345)} + 96P_{(21453)} + 90P_{(13254)} + 84P_{(13524)} \\ + 80P_{(13425)} + 64P_{(13452)} + 45P_{(21354)} + 40P_{(12543)} + 32(P_{(23514)} + P_{(14352)} + P_{(13542)}) \\ + 24P_{(21543)} + 22P_{(24153)} + 20P_{(14325)} + 18P_{(14523)} + 16P_{(25143)} + 12(P_{(24513)} + P_{(14532)}) \\ + 6(P_{(25314)} + P_{(15342)}) + 4P_{(25413)} + 2(P_{(15432)} + P_{(35142)}))/5 \end{aligned}$$

When  $(x_i, y_i)$ ,  $i = 1, 2, \dots, k$ , are normal variables with correlation  $\rho$ , then closed form expressions exist for  $p, k$  and  $u$  (writing  $a = \arcsin(\rho) / \pi$  and  $b = 2 \arcsin\left(\frac{1}{2}\rho\right) / \pi$ ):

$$\begin{aligned} p &= \frac{1}{2} + a & (38.3.16) \\ k &= \frac{1}{4}\left(\frac{10}{9} + 4a + 4a^2 - b^2\right) \\ u &= \frac{3}{8}\left(\frac{4}{9} + 4a - 2b + 4a^2 - b^2\right) \end{aligned}$$

The parameters  $t, b$  and  $y$  have to be evaluated by numerical intergration of the rank order probabilities listed in (38.3.15), as described in section 38.3.3.2.

## 38.4 Noncentral Distribution of Jonckheere-Terpsta's S

### 38.4.1 Definition

The Jonckheere-Terpstra test is a test for an ordered alternative hypothesis within an independent samples (between-participants) design. It is similar to the Kruskal-Wallis test in that the null hypothesis is that several independent samples are from the same population. However, with the Kruskal-Wallis test there is no a priori ordering of the populations from which the samples are drawn. When there is an a priori ordering, the Jonckheere test has more statistical power than the Kruskal-Wallis test.

The null and alternative hypotheses can be conveniently expressed in terms of population medians for  $k$  populations (where  $k > 2$ ). Letting  $\theta_i$  be the population median for the  $i$ th population, the null hypothesis is:

The alternative hypothesis is that the population medians have an a priori ordering e.g.: with at least one strict inequality.

### 38.4.2 Central distribution of Jonckheere-Terpsta's S

See [Murakami & Kamakura \(2009\)](#) for saddlepoint approximation.

See also [Neuhäuser & Hothorn \(1998\)](#)

Consider two vectors of scores  $X = (x_1, \dots, x_N)$  and  $Y = (y_1, \dots, y_N)$ . Let  $J_N$  denote Kendall's  $T_N$  statistic (see ...) with ties in  $X$ , let  $k$  denote the number of blocks with ties in  $X$  ( $k \leq N$ ), let  $n_i$  denote the size of the  $i$ th block.

#### 38.4.2.1 Exact distribution

Let  $p(n_1, \dots, n_k; t) = \Pr[J_N = t]$ . If  $J_N$  is based on  $k$  independent samples of sizes  $n_1, \dots, n_k$ , then (Skillings 1980):

$$p(n_1, \dots, n_k; t) = \sum_x p(n_1, \dots, n_k; x) \times p(n_1, \dots, n_k; t - x) \quad (38.4.1)$$

where the sum is over all  $x$  with positive  $p(\cdot)$ .

#### 38.4.2.2 Cumulants

The cumulants of  $J_N$  are given by (Robillard, 1972):

$$\begin{aligned} \kappa_{2j} &= \frac{B_{2j}}{2j} \left[ \sum_{s=1}^N s^{2j} - \sum_{i=1}^k \sum_{s=1}^{n_i} s^{2j} \right] \\ &= \frac{B_{2j}}{2j(2j+1)} \left[ B_{2j+1}(N+1) + (k-1)B_{2j+1} - \sum_{i=1}^k B_{2j+1}(n_i+1) \right] \end{aligned} \quad (38.4.2)$$

and  $\kappa_{2j+1} = 0$ ,  $j \geq 1$ , and  $B_{2j}$  and  $B_{2j}(x)$  are the Bernoulli numbers and polynomials, respectively, of degree  $2j$  (see section 14.2).

See also [Skillings \(1980\)](#)

## 38.5 Noncentral Distribution of Spearman's Rho

### 38.5.1 Definition

Spearman's  $\rho$  is calculated like Pearson's correlation coefficient, using the rank-transform on the  $x_i$  and  $y_i$ . The exact test for  $H_0$  is the permutation test. Techniques for obtaining the exact distribution and approximations are given below

### 38.5.2 Central Cumulants

The first 8 cumulants are given by (David *et al.*, 1951)

$$\kappa_2 = \frac{1}{n-1} \quad (38.5.1)$$

$$\kappa_4 = \frac{-6(19n^2 + 5n - 36)}{25n(n+1)(n-1)^3} \quad (38.5.2)$$

$$\kappa_6 = \frac{48(583n^6 + 723n^5 - 2603n^4 - 2637n^3 + 4054n^2 + 2760n - 1800)}{(245n^3(n-1)^5n1^3)} \quad (38.5.3)$$

$$\begin{aligned} \kappa_8 = & \frac{144(41939n^{10} - 83709n^9 + 304254n^8 + 578442n^7 - 1012323n^6 - 1690125n^5)}{875n^5(n+1)^5(n-1)^7} \\ & + \frac{144(1800776n^4 + 2358048n^3 - 1616688n^2 - 1080567n + 846720)}{875n^5(n+1)^5(n-1)^7} \end{aligned} \quad (38.5.4)$$

Note: See `PermCumulants.SpearmanCum` for further details.

### 38.5.3 Noncentral Moments

The expected value for  $\rho_s$  in samples from a bivariate normal population with parameter  $rho$  is

$$E(r_s) = \frac{6}{\pi(n+1)}(\arcsin(\rho) + (n-2)\arcsin(\rho/2)) \quad (38.5.5)$$

See Xu *et al.* (2013) for exact 2nd noncentral moment, and also van de Wiel & Di Buccianico (2001), and Iman *et al.* (1975), and Koning & Does (1988), and Lee & Lin (1992)

### 38.5.4 Recursive algorithm

Let  $X = (x_1, \dots, x_N)$  and  $Y = (y_1, \dots, y_N)$  denote 2 fixed ordered sets of scores.

Let  $Z = (z_1, \dots, z_n)$  denote any ordered subset of  $Y$  of size  $n$ . Let  $S = x_1z_1 + \dots + x_nz_n$ .

Let  $W_n(S; (x_1, z_1), \dots, (x_n, z_n))$  denote the number of ways of forming  $S$  when based on the first  $n$  elements of  $X$  and  $Z$ . Then the following recurrence relation holds:

$$W_n(S; (x_1, z_1), \dots, (x_n, z_n)) = \sum_{i=1}^n W_{n-1}(S - x_nz_i; (x_1, z_1), \dots, (x_{i-1}, z_{i-1}), (x_i, z_{i+1}), \dots, (x_{n-1}, z_n)) \quad (38.5.6)$$

Algorithm 2: Complete enumeration of all permutations

See van de Wiel & Di Buccianico (2001), and Iman *et al.* (1975), and Koning & Does (1988), and Lee & Lin (1992)

## 38.6 Noncentral Distribution of Page's $L$

### 38.6.1 Definition

Spearman's  $L$  is calculated like Pearson's correlation coefficient, using the rank-transform on the  $x_i$  and  $y_i$ . The exact test for  $H_0$  is the permutation test. Techniques for obtaining the exact distribution and approximations are given below

### 38.6.2 Central Cumulants

The first 8 cumulants are given by (David *et al.*, 1951)

$$\kappa_2 = \frac{1}{n-1} \quad (38.6.1)$$

$$\kappa_4 = \frac{-6(19n^2 + 5n - 36)}{25n(n+1)(n-1)^3} \quad (38.6.2)$$

$$\kappa_6 = \frac{48(583n^6 + 723n^5 - 2603n^4 - 2637n^3 + 4054n^2 + 2760n - 1800)}{(245n^3(n-1)^5n1^3)} \quad (38.6.3)$$

$$\begin{aligned} \kappa_8 = & \frac{144(41939n^{10} - 83709n^9 + 304254n^8 + 578442n^7 - 1012323n^6 - 1690125n^5)}{875n^5(n+1)^5(n-1)^7} \\ & + \frac{144(1800776n^4 + 2358048n^3 - 1616688n^2 - 1080567n + 846720)}{875n^5(n+1)^5(n-1)^7} \end{aligned} \quad (38.6.4)$$

Note: See `PermCumulants.SpearmanCum` for further details.

### 38.6.3 Noncentral Moments

The expected value for  $L_s$  in samples from a bivariate normal population with parameter  $L$  is

$$E(r_s) = \frac{6}{\pi(n+1)}(\arcsin(L) + (n-2)\arcsin(L/2)) \quad (38.6.5)$$

See Xu *et al.* (2013) for exact 2nd noncentral moment, and also van de Wiel & Di Buccianico (2001), and Iman *et al.* (1975), and Koning & Does (1988), and Lee & Lin (1992)

### 38.6.4 Recursive algorithm

Let  $X = (x_1, \dots, x_N)$  and  $Y = (y_1, \dots, y_N)$  denote 2 fixed ordered sets of scores.

Let  $Z = (z_1, \dots, z_n)$  denote any ordered subset of  $Y$  of size  $n$ . Let  $S = x_1z_1 + \dots + x_nz_n$ .

Let  $W_n(S; (x_1, z_1), \dots, (x_n, z_n))$  denote the number of ways of forming  $S$  when based on the first  $n$  elements of  $X$  and  $Z$ . Then the following recurrence relation holds:

$$W_n(S; (x_1, z_1), \dots, (x_n, z_n)) = \sum_{i=1}^n W_{n-1}(S - x_nz_i; (x_1, z_1), \dots, (x_{i-1}, z_{i-1}), (x_i, z_{i+1}), \dots, (x_{n-1}, z_n)) \quad (38.6.6)$$

Algorithm 2: Complete enumeration of all permutations

See van de Wiel & Di Buccianico (2001), and Iman *et al.* (1975), and Koning & Does (1988), and Lee & Lin (1992)

## 38.7 Noncentral Distribution of Kruskal-Wallis' H

### 38.7.1 Definition

The exact or approximate distributions of may commonly used rank-based procedures in a one-way or two-way layout can be obtained using algorithms which first calculate the density of the total scores and then in a second step the statistic of interest (an analogue to parametric ANOVA, Many-one comparisons, all-pairs comparisons, etc). In this section the general steps are covered, and relationships to some specific procedures are given.

### 38.7.2 Density of total scores of a one-way layout

#### 38.7.2.1 Null distribution: Recursive generation of the pdf of all total scores

Under the null-hypothesis, all permutations are equally likely to occur, and recursive generation of the whole pdf is possible for small sample sizes. Let  $S_N$  denote the  $N^{th}$  element in an ordered vector of scores, let

$$P_{n_1, \dots, n_k} = (T_1, \dots, T_i, \dots, T_k)W(n_1, \dots, n_i, \dots, n_k) \quad (38.7.1)$$

denote the number of ways of forming the score totals  $T_1, \dots, T_i, \dots, T_k$  with samples of sizes  $n_1, \dots, n_i, \dots, n_k$ , and let  $N = n_1 + n_2 + \dots + n_k$ . Then the following recursion relation holds:

$$P_{n_1, \dots, n_k} == \sum_{i=1}^k (T_1, \dots, T_i - S_N, \dots, T_k)W(n_1, \dots, n_i - 1, \dots, n_k) \quad (38.7.2)$$

#### 38.7.2.2 Noncentral distribution: General case

Under a non-null hypothesis, the assumption that all permutations are equally likely to occur usually does not hold, and the recursive scheme above cannot be applied. In this case, for each vector  $T$  of  $k$  total scores a complete enumeration of all permutations forming this vector is necessary (see [Chase \(1970\)](#) and [Stockmal \(1962\)](#)). For each such permutation, the probability to occur needs to be calculated, based on the specification of the non-null hypothesis, and the sum of all these probabilities is the probability of the is the probability of  $T$ .

#### 38.7.2.3 Noncentral distribution: Alternatives based on the Normal distribution

We consider  $k$  continuous random variables  $X_i$  with density functions  $f_i$  and sample sizes  $n_i$ ,  $i = 1 \dots k$ , and  $N = n_1 + \dots + n_k$ .

Let  $\mathbf{U} = (U_1, \dots, U_N)$ ,  $U_1 < \dots < U_N$ , denote the order statistics of the random variables  $(X_{1,1}, \dots, X_{1,n_1}, \dots, X_{k,1}, \dots, X_{k,n_k})$ , and let  $\mathbf{Z} = (Z_1, \dots, Z_N)$  denote a random vector of integers  $1, 2, \dots, k$ , where the  $i$ th component  $\mathbf{Z}_i$  is  $i$  if  $U_i$  is an  $X_i$ . If  $\mathbf{z} = (z_1, \dots, z_N)$  is a fixed vector of integers  $1, 2, \dots, k$  (with each  $i$  occurring  $n_i$  times), the probability of the rank order  $z$ ,  $\Pr[\mathbf{Z} = \mathbf{z}]$ , is given by

$$P_{n_1, \dots, n_k}(\mathbf{z}|d) = n_1! \dots n_k! \int_R \dots \int \prod_{i=1}^N f_i(t_i) dt_i, \quad (38.7.3)$$

where the region of integration  $R$  is  $-\infty < t_1 \leq t_2 \leq \dots \leq t_N < \infty$  ([Milton, 1970](#)).

### 38.7.2.4 Noncentral distribution: Lehmann Alternatives

Lehmann alternatives are of the form  $F_1(x) = [F_0(x)]^k$  or  $F_1(x) = 1 - [1 - F_0(x)]^k$ . The exact distribution under this alternative can be calculated using recurrence relations, similar to the null-distribution ([Shorack, 1966](#)): Let  $p_{n,m}(u)$  denote the probability that  $U = u$  in samples of size  $n$  and  $m$ . Then

$$(km + n)p_{n,m}(u) = np_{n-1,m}(u - m) + kmp_{n,m-1}(u) \quad \text{for } F_1 = (F_0)^k \quad (38.7.4)$$

$$(km + n)p_{n,m}(u) = np_{n-1,m}(u) + kmp_{n,m-1}(u - n) \quad \text{for } F_1 = 1 - (1 - F_0)^k \quad (38.7.5)$$

This recursive procedure allows the calculation of the exact noncentral distribution also for larger samples.

### 38.7.3 Moments and cumulants of linear rank statistics for the one-way layout

For both central and noncentral distributions, the moments, and from the moments the cumulants (see section [36.1](#)) of linear rank statistics can be determined numerically from the pdf. Linear rank statistics for which this can be easily done are orthogonal polynomials, the Jonckhere-Terpstra statistic and Halperin type Mann-Whitney statistics.

### 38.7.4 Special cases for the one-way layout (H-type)

#### 38.7.4.1 Kruskal-Wallis Test

See [Spurrier \(2003\)](#)

See [Robinson \(1980\)](#)

See [Di Bucchianico & van de Wiel \(2005\)](#)

See [van de Wiel \(2004\)](#)

See [Fan \*et al.\* \(2011\)](#)

See [Fan & Zhang \(2012\)](#)

#### 38.7.4.2 Koziol Test

See also [Koziol \(1982\)](#)

## 38.8 Noncentral Distributions of Friedman's S

### 38.8.1 Density of total scores in a two-way layout

Under the assumption of independent blocks, the following procedures are valid both for the central and noncentral distributions described above.

#### 38.8.1.1 Linear Rank Statistics

For the linear rank statistics described above, the pdf of the statistics can be first calculated per block. Then the full distribution can be calculated as

$$p(n_1, \dots, n_k; t) = \sum_x p(n_1, \dots, n_k; x) \times p(n_1, \dots, n_k; t - x) \quad (38.8.1)$$

#### 38.8.1.2 Friedman Type

Consider a two-way layout of  $n$  rows ("blocks") and  $k$  columns ("treatments"), where the observations in different blocks are assumed to be independent. Let  $W(n; T_1, \dots, T_k)$  denote the number of ways of forming the ordered vector of column rank totals  $(T_1, \dots, T_k)$  when based on  $(n-1)$  blocks, and let  $(X_{1(i)}, \dots, X_{k(i)})$  denote the  $i^{th}$  permutation of the rank totals in the  $n^{th}$  block. Then the following recurrence relation holds:

$$W(n; T_1, \dots, T_k) = \sum_{i=1}^k W(n-1; T_1 - X_{1(i)}, \dots, T_k - X_{k(i)}) \quad (38.8.2)$$

For literature, see Odeh(1977b), Patil (1975)

#### 38.8.1.3 General case

Consider a two-way layout of  $k$  columns ("treatments"), and  $l$  independent blocks with sample sizes  $n_{ij}$  for the  $i^{th}$  treatment in the  $j^{th}$  block,  $i = 1 \dots k$  and  $j = 1 \dots l$ . Let  $Q(l; T_1, \dots, T_k)$  denote the probability of the column-total vector  $(T_1, \dots, T_k)$ , based on  $l$  blocks, and let  $p_j(x_1, \dots, x_k)$  denote the probability of the column-total vector  $(x_1, \dots, x_k)$  in the  $j^{th}$  block. Then the following recurrence relationship holds:

$$Q(l+1; T_1, \dots, T_k) = \sum_{x_1, \dots, x_k} Q(l; T_1 - x_1, \dots, T_k - x_k) \times p_{n+1}(x_1, \dots, x_k) \quad (38.8.3)$$

where the sum is over all  $x$  with positive  $p(\cdot)$ .

### 38.8.2 Moments and cumulants of linear rank statistics for the two-way layout

For both central and noncentral distributions, the cumulants of linear rank statistics which have been determined per block as described in section 38.7.3 can be used to calculate the cumulants for the two-way layout, since the blocks are assumed to be independent. Let  $\kappa_{r,j}$  denote the  $r^{th}$  cumulant of the linear rank statistic in the  $i^{th}$  block. Then the  $r^{th}$  cumulant of the combined statistic is given by

$$\kappa_r = \sum_{i=1}^l \kappa_{r,j} \quad (38.8.4)$$

### 38.8.3 Special cases for the two-way layout

#### 38.8.3.1 Friedman Test

See [Skillings & Mack \(1981\)](#)

See [Röhmel \(1997\)](#)

See [Gansky \*et al.\* \(2001\)](#)

#### 38.8.3.2 Cochran Test

#### 38.8.3.3 Quade Test

See [Quade \(1979\)](#)

#### 38.8.3.4 Mack-Skillings Test

See also [Mack & Skillings \(1980\)](#)

See also [Skillings & Mack \(1981\)](#)

#### 38.8.3.5 Koziol Test

See also [Koziol \(1982\)](#)

# Chapter 39

## Examples: Statistical Procedures

### 39.1 Introduction to Inferential Statistics Functions

Reference [Bortz et al. \(1990\)](#)

Reference [Conover & Iman \(1981\)](#)

#### 39.1.1 The Linear Model

This section describes the following computational and conceptual relationships (see "a tale of two regressions")

- Simple Linear regression between two random variables and between one random variable and one design (dummy) variable (Classical t-test)
- Multiple Linear regression between one predicted random variable and multiple predictor variables and between one predicted random variable and multiple design (dummy) variables (Classical ANOVA)
- Canonical correlation analysis between two sets of random variables and between one set of random variables and one set of design (dummy) variables (Classical MANOVA)

#### 39.1.2 Transformations

This section describes the following computational and conceptual relationships

- The t-test for 2 independent groups, the Mann-Whitney Test, and the  $2 \times 2$  contingency table
- The classical ANOVA for  $k$  independent groups, the Kruskal-Wallis Test, and the  $r \times 2$  contingency table
- The classical MANOVA for  $k$  independent groups, the Multivariate Kruskal-Wallis Test, and the  $r \times c$  contingency table

#### 39.1.3 Randomisation, Block-Randomisation, and Permutation Tests

This section reviews Randomisation and Block-Randomisation and discusses the implications for permutation tests and normal-theory based procedures like classical ANOVA.

### 39.1.4 Commonly Used Function Types

#### 39.1.4.1 Functions returning p-Values and Confidence Intervals

These functions have the form `Test?(Proc; Alpha; [Parameters;], OutputString)`. Here “?” is a placeholder for the type data to be analyzed, e.g. “2i” for 2 independent samples, “Proc” specifies the procedures which should be performed, e.g. “TTest” for Student’s t-test, “Alpha” is the type I error used for the test, it is also used to specify the confidence level (1-Alpha) which will be used for the construction of confidence intervals, “[Parameters;]” denote any parameters (like sample sizes, means, standard deviations) which the tests require, and “OutputString” specifies the computed results which will be returned. This can be any of the following:

- **Input:** a list of the input used for the test.
- **Description:** a list of all test-specific descriptive results, like degrees of freedom, t-values etc. If only some of these are desired, then they need to be listed individually, as stated in the description of each test.
- **TCrit1:** the critical value for a one-sided test
- **TCrit2:** the critical value for a two-sided test
- **PValueH01[method]:** the p-value for  $H_{01}$  (depends on the test, e.g.  $\mu_1 \geq \mu_2$ )
- **PValueH02[method]:** the p-value for  $H_{02}$  (depends on the test, e.g.  $\mu_1 \leq \mu_2$ )
- **PValueH03[method]:** the p-value for  $H_{03}$  (depends on the test, e.g.  $\mu_1 = \mu_2$ )
- **LowerCL1[item],[method]:** lower confidence limit, 1-sided
- **UpperCL1[item],[method]:** upper confidence limit, 1-sided
- **LowerCL2[item],[method]:** lower confidence limit, 2-sided
- **UpperCL2[item],[method]:** upper confidence limit, 2-sided
- **LengthCI2[item],[method]:** length of the 2-sided confidence interval

Note: the optional parameter **[method]** allows you to specify the method of calculation for p-values and confidence intervals (e.g. an exact method or an approximation). The optional parameter **[item]** allows you to choose for what the confidence interval will be calculated (e.g. difference of the means or effect size). The available choices depend on the test. If these optional parameters are omitted, the defaults will be used.

As an example, for a test for 2 independent samples, “2i” is used to specify the type of data to be analyzed, and there are 3 test parameters, the sample size **N**, the means **Mean**, and the standard deviations **StDev**.

**Test2i(Proc As String;  $\alpha$  As mpNum, **N** As mpNum[], **Mean** As mpNum[], **StDev** As mpNum[], OutputString As String) As mpNumList,**

and an actual call to the function, requesting Student’s t-test with description, the critical value for a two-sided test, the p-value for  $H_{03}$  (in the case of **TTest** this is  $\mu_1 \neq \mu_2$ ), for 2 independent

samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

---

```
Result = Test2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 + PValueH03")
mp.Print Result
```

---

which produces the output

```
df: 18
Difference of Means: -2.2
t-value: -4.9193496
TCrit2: 2.100922
PValueH03: 0.0001106
```

### 39.1.4.2 Functions returning Power Estimates

These functions have the form `TestPower?(Proc; Alpha; [Parameters;], OutputString)`. Here “?” is a placeholder for the type data to be analyzed, e.g. “2i” for 2 independent samples, “Proc” specifies the procedures which should be performed, e.g. “TTest” for Student’s t-test, “Alpha” is the type I error used for the test, it is also used to specify the confidence level (1-Alpha) which will be used for the construction of confidence intervals, “[Parameters;]” denote any parameters (like sample sizes, means, standard deviations) which the tests require, and “OutputString” specifies the computed results which will be returned. This can be any of the following:

- **Input:** a list of the input used for the test.
- **Description:** a list of all test-specific descriptive results, like degrees of freedom, t-values etc. If only some of these are desired, then they need to be listed individually, as stated in the description of each test.
- **TCrit1:** the critical value for a one-sided test
- **TCrit2:** the critical value for a two-sided test
- **PowerHA1[method]:** the power to reject  $H_{01}$  (depends on the test, e.g.  $\mu_1 \geq \mu_2$ ) and to accept  $H_{A1}$
- **PowerHA2[method]:** the power to reject  $H_{02}$  (depends on the test, e.g.  $\mu_1 \leq \mu_2$ ) and to accept  $H_{A2}$
- **PowerHA3[method]:** the power to reject  $H_{03}$  (depends on the test, e.g.  $\mu_1 = \mu_2$ ) and to accept  $H_{A3}$

Note: the optional parameter **[method]** allows you to specify the method of calculation for p-values and confidence intervals (e.g. an exact method or an approximation). The available choices depend on the test. If these optional parameters are omitted, the defaults will be used.

As an example, for a test for 2 independent samples, “2i” is used to specify the type of data to be analyzed, and there are 3 test parameters, the sample size **N**, the means **Mean**, and the standard deviations **StDev**.

**TestPower2i(Proc As String;  $\alpha$  As mpNum, **N** As mpNum[], **Mean** As mpNum[], **StDev** As mpNum[], OutputString As String) As mpNumList,**

and an actual call to the function, requesting Student’s t-test with description, the critical value for a two-sided test, the power for  $H_{A3}$  (in the case of TTest this is  $\mu_1 \neq \mu_2$ ), for 2 independent samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

---

```
Result = TestPower2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 +
    PowerHA3")
mp.Print Result
```

---

which produces the output

```
df: 18
Difference of Means: -2.2
t-value: -4.9193496
TCrit2: 2.100922
PValueH03: 0.996354
```

### 39.1.4.3 Functions returning Sample Size Estimates

These functions have the form `TestSampleSize?(Proc; Alpha; Beta; [Parameters;], OutputString)`. Here

”?” is a placeholder for the name of the distribution,

”Proc” specifies the procedures which should be performed, e.g. ”TTest” for Student’s t-test,

”Alpha” specifies the confidence level (or Type I error),

”Beta” specifies the Type I error (or  $1 - \text{Power}$ ),

”[Parameters;]” denote any additional parameters of the distribution (if any) which are not a function of the sample size, and

”OutputString” specifies the computed results which will be returned. This can be any of the following:

- **ExactN**: returns an ”exact”, i.e. typically non-integer sample size estimate
- **UpperN**: upper integer sample size estimate
- **LowerN**: lower integer sample size estimate
- **UpperNPower**: actual power when using UpperN
- **LowerNPower**: actual power when using LowerN

As an example, for a test for 2 independent samples, ”2i” is used to specify the type of data to be analyzed, and there are 2 test parameters, the means **Mean**, and the standard deviations **StDev**.

`TestSampleSize2i(Proc As String,  $\alpha$  As mpNum,  $\beta$  As mpNum, Mean As mpNum[], StDev As mpNum[], OutputString As String) As mpNumList`

and an actual call to the function, requesting an upper sample size estimate (and actual power) for  $\alpha = 0.95$ ,  $\beta = 0.1$ , and standard deviations  $\sigma_1 = \sigma_2 = 1$ , means  $\mu_1 = 2.3$  and  $\mu_2 = 4.5$ , would be

---

```
Result = TestSampleSize2i("TTest", 0.05, 0.1, {2.3, 4.5}, 1, "UpperN + UpperNPower")
mp.Print Result
```

---

which produces the output

```
UpperN: 2 x 6 = 12
UpperNPower: 0.9285919
```

## 39.2 Tests for the mean from 1 sample (Student's t-test)

### 39.2.1 Overview

Let  $(X_1, X_2, \dots, X_N)$  denote a random sample of size  $N$  from a normal distribution with mean  $\mu$  and variance  $\sigma^2$ , and let

$$\bar{x}_1 = \frac{1}{N} \sum_{i=1}^N X_i \quad \text{and} \quad s^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{x}_1)^2 \quad (39.2.1)$$

be the usual sample estimates of the unknown population mean  $\mu$  and unknown population variance  $\sigma^2$ . Then Student's t-test can be used to test hypotheses concerning  $\mu$  with regard to a reference value  $\mu_0$ .

### 39.2.2 Tests and Confidence Intervals

---

Function **StudentTTest1**(*Type1Error* As *mpNum*, *N* As *mpNum*[], *Mean* As *mpNum*[], *StDev* As *mpNum*[], *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **StudentTTest1** returns p-values, confidence intervals and related information for Student's t-test

#### Parameters:

*Type1Error*: A real number greater than 0 and less than 1.

*N*: An array of integers greater than 1, representing the sample sizes

*Mean*: An array of reals, representing the means

*StDev*: An array of positive reals, representing the standard deviations

*Output*: A string describing the output choices

See section 39.1.4.1 for the options for *Type1Error* and *Output*). Algorithms and formulas are given in sections 39.2.2.1 and 39.2.2.3.

#### 39.2.2.1 Tests: algorithms and formulas

Let  $F_t(\cdot, \nu)$  denote the CDF (see section 32.13.2.2) and let  $t_{\nu, \alpha}$  denote the  $\alpha$ -quantile (see section 32.13.3) of the  $t$ -distribution with  $\nu$  degrees of freedom. Define

$$t = \frac{\bar{x}_1 - \mu_0}{s}, \quad s = \sqrt{s_1^2/N}, \quad \nu = N - 1. \quad (39.2.2)$$

Then  $p$ -values and rejection criteria for  $H_0$  can be calculated as summarized in Table 39.1.

Test problem	<i>p</i> -value	Reject $H_0$
$H_{01} : \mu \leq \mu_0$ vs $H_{A1} : \mu > \mu_0$	$F_t(-t, \nu)$	$t > t_{\nu; 1-\alpha}$
$H_{02} : \mu \geq \mu_0$ vs $H_{A2} : \mu < \mu_0$	$F_t(t, \nu)$	$t > t_{\nu; \alpha}$
$H_{03} : \mu = \mu_0$ vs $H_{A3} : \mu \neq \mu_0$	$F_t(t, \nu) - F_t(-t, \nu)$	$t > t_{\nu; 1-\alpha/2}$ or $t > t_{\nu; \alpha/2}$

Table 39.1: Student's t-test, 1 sample, tests for the mean

The test can also be expressed in terms of a correlation coefficient  $r$  between the combined  $X$  and an indicator variable, where  $t$  and  $r$  are related by

$$r = \frac{t}{\sqrt{t^2 + \nu}}, \quad t = \nu \frac{r}{1 - r^2}. \quad (39.2.3)$$

### 39.2.2.2 Tests: examples

An actual call to the function, requesting Student's t-test with description, the critical value for a two-sided test, the p-value for  $H_{03}$  (in the case of `TTest` this is  $\mu_1 \neq \mu_2$ ), for 2 independent samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

---

```
Result = Test2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 + PValueH03")
mp>Show Result
```

---

which produces the output

```
df: 18
Difference of Means: -2.2
t-value: -4.9193496
TCrit2: 2.100922
PValueH03: 0.0001106
```

### 39.2.2.3 Confidence Intervals: algorithms and formulas

Let  $A_1 = t_{\nu, \alpha} \cdot s$  and  $A_2 = t_{\nu, \alpha/2} \cdot s$ , where  $s$  and  $\nu$  are defined in (39.2.2), and  $t_{\nu, \alpha}$  denotes the  $\alpha$ -quantile of the (central)  $t$ -distribution with  $\nu$  degrees of freedom (see section 32.13.3).

Then confidence intervals for  $\mu_1 - \mu_0$  can be calculated as summarized in Table 39.2.

Type	Confidence Interval (Difference of Means)
Left-sided	$-\infty \leq \mu_1 - \mu_0 \leq (\bar{x}_1 - \mu_0) + A_1$
Right-sided	$(\bar{x}_1 - \mu_0) - A_1 \leq \mu_1 - \mu_0 \leq +\infty$
Two-sided	$(\bar{x}_1 - \mu_0) - A_2 \leq \mu_1 - \mu_0 \leq (\bar{x}_1 - \mu_0) + A_2$

**Table 39.2:** Student's t-test, 1 sample, confidence intervals for the mean

### 39.2.2.4 Confidence Intervals: examples

An actual call to the function, requesting Student's t-test with description, the critical value for a two-sided test, the p-value for  $H_{03}$  (in the case of `TTest` this is  $\mu_1 \neq \mu_2$ ), for 2 independent samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

---

```
Result = Test2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 + PValueH03")
mp>Show Result
```

---

which produces the output

```
df: 18
Difference of Means: -2.2
```

t-value: -4.9193496  
TCrit2: 2.100922  
PValueH03: 0.0001106

### 39.2.3 Power

---

Function **StudentTTestPower1**(*Type1Error* As *mpNum*, *N* As *mpNum*[], *Mean* As *mpNum*[], *StDev* As *mpNum*[], *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **StudentTTestPower1** returns power estimations and related information for Student's t-test

**Parameters:**

*Type1Error*: An real number greater then 0 and less than 1.

*N*: An array of integers greater than 1, representing the samples sizes

*Mean*: An array of reals, representing the means

*StDev*: An array of positive reals, representing the standard deviations

*Output*: A string describing the output choices

See section 39.1.4.2 for the options for *Type1Error* and *Output*). Algorithms and formulas are given in section 39.2.3.1.

#### 39.2.3.1 Power calculations: algorithms and formulas

Let  $\sigma_1^2 = \sigma^2$  and  $\nu = N - 1$ . Define

$$\tilde{\rho} = \frac{\mu_1 - \mu_0}{\sigma} \text{ and } \delta = \sqrt{\nu} \tilde{\rho}. \quad (39.2.4)$$

Let  $F_{t'}(\cdot, \nu, \delta)$  denote the CDF of the (singly) noncentral  $t$ -distribution with  $\nu$  degrees of freedom and noncentrality parameter  $\delta$  (see section and let  $t_{\nu, \alpha}$  denote the  $\alpha$ -quantile of the central  $t$ -distribution with  $\nu$  degrees of freedom (see section 32.13.3).

Then the power for accepting  $H_A$  at the confidence level  $\alpha$  can be calculated as summarized in Table 39.3.

Test	Null Hypothesis	Alternative	Power
1 sided	$H_{01} : \mu \leq \mu_0$	$H_{A1} : \mu > \mu_0$	$F_{t'}(-t_{\nu;1-\alpha}, \nu, \delta)$
1 sided	$H_{02} : \mu \geq \mu_0$	$H_{A2} : \mu < \mu_0$	$F_{t'}(t_{\nu;1-\alpha}, \nu, \delta)$
2 sided	$H_{03} : \mu = \mu_0$	$H_{A1} : \mu > \mu_0$	$F_{t'}(-t_{\nu;1-\alpha/2}, \nu, \delta)$
2 sided	$H_{03} : \mu = \mu_0$	$H_{A2} : \mu < \mu_0$	$F_{t'}(t_{\nu;1-\alpha/2}, \nu, \delta)$
2 sided	$H_{03} : \mu = \mu_0$	$H_{A3} : \mu \neq \mu_0$	$F_{t'}(t_{\nu;1-\alpha/2}, \nu, \delta) - F_{t'}(-t_{\nu;1-\alpha/2}, \nu, \delta)$

**Table 39.3:** Student's t-test, 1 sample, power calculations

#### 39.2.3.2 Power calculations: examples

An actual call to the function, requesting Student's t-test with description, the critical value for a two-sided test, the power for  $H_{A3}$  (in the case of TTest this is  $\mu_1 \neq \mu_2$ ), for 2 independent samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

---

```
Result = TestPower2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 +
PowerHA3")
```

---

```
mp.Print Result
```

---

which produces the output

```
df: 18
Difference of Means: -2.2
t-value: -4.9193496
TCrit2: 2.100922
PValueH03: 0.996354
```

### 39.2.4 Sample Size Calculation

---

Function **StudentTTestSampleSize1**( *Type1Error* As *mpNum*, *Type2Error* As *mpNum*, *Mean* As *mpNum*[], *StDev* As *mpNum*[], *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **StudentTTestSampleSize1** returns returns sample size estimations and related information for Student's t-test

**Parameters:**

*Type1Error*: An real number greater then 0 and less than 1.

*Type2Error*: An real number greater then 0 and less than 1.

*Mean*: An array of reals, representing the means

*StDev*: An array of positive reals, representing the standard deviations

*Output*: A string describing the output choices

See section 39.1.4.2 for the options for *Type1Error*, *Type2Error* and *Output*. Algorithms and formulas are given in section 39.2.4.1.

#### 39.2.4.1 Sample size calculations: algorithms and formulas

Let  $N_{t'}(\alpha, \beta, \tilde{\rho})$  denote the sample size function of the (singly) noncentral  $t$ -distribution (see section 37.8.7) for a given confidence level  $\alpha$ , power  $\beta$  and noncentrality parameter  $\tilde{\rho}$  (as defined in equation 39.2.4).

The required total sample size  $N$  can be calculated as summarized in Table 39.4.

Test	Null Hypothesis	Alternative	Minimal sample size
1 sided	$H_{01} : \mu \leq \mu_0$	$H_{A1} : \mu > \mu_0$	$N_{t'}(\alpha, \beta, \tilde{\rho})$
1 sided	$H_{02} : \mu \geq \mu_0$	$H_{A2} : \mu < \mu_0$	$N_{t'}(\alpha, \beta, \tilde{\rho})$
2 sided	$H_{03} : \mu = \mu_0$	$H_{A1} : \mu > \mu_0$	$N_{t'}(\alpha, \beta, \tilde{\rho})$
2 sided	$H_{03} : \mu = \mu_0$	$H_{A2} : \mu < \mu_0$	$N_{t'}(\alpha, \beta, \tilde{\rho})$
2 sided	$H_{03} : \mu = \mu_0$	$H_{A3} : \mu \neq \mu_0$	$N2_{t'}(\alpha, \beta, \tilde{\rho})$

**Table 39.4:** Student's t-test, 1 sample, sample size calculations

Note that the returned value of  $N$  will in general not be an integer, and rounding up may be required.

### 39.2.4.2 Sample size calculations: examples

An actual call to the function, requesting an upper sample size estimate (and actual power) for  $\alpha = 0.95$ ,  $\beta = 0.1$ , and standard deviations  $\sigma_1 = \sigma_2 = 1$ , means  $\mu_1 = 2.3$  and  $\mu_2 = 4.5$ , would be

---

```
Result = TestSampleSize2i("TTest", 0.05, 0.1, {2.3, 4.5}, 1, "UpperN + UpperNPower")
mp.Print Result
```

---

which produces the output

```
UpperN: 2 x 6 = 12
UpperNPower: 0.9285919
```

### 39.2.5 Confidence Interval (Effect size)

Let  $T$  be a statistic according to the non-central  $t$ -distribution with  $n$  degrees of freedom and a non-centrality parameter  $\delta$ . Then the lower confidence limit  $\hat{\delta}$  of level  $1 - \alpha$  and the two-sided confidence interval  $[\underline{\delta}, \bar{\delta}]$  of the non-centrality parameter  $\delta$  of level  $1 - \alpha$  are given by [Akahira et al. \(1995\)](#):

$$\hat{\delta} = bT - z_\alpha \sqrt{k} + hT^3(z_\alpha^2 - 1)/k, \quad (39.2.5)$$

$$\underline{\delta} = bT - z_{\alpha/2} \sqrt{k} + hT^3(z_{\alpha/2}^2 - 1)/k, \quad (39.2.6)$$

$$\bar{\delta} = bT + z_{\alpha/2} \sqrt{k} - hT^3(z_{\alpha/2}^2 - 1)/k, \quad (39.2.7)$$

where  $k = 1 + (1 - b^2)T^2$ ,  $h$  and  $b$  are defined in equation (37.8.19), and  $z_\alpha$  denotes the  $\alpha$ -quantile of the normal distribution (see section 32.11.3).

Let  $A_1 = t_{\nu, \alpha} \cdot s$  and  $A_2 = t_{\nu, \alpha/2} \cdot s$ , where  $s$  and  $\nu$  are defined in (39.3.3), and  $t_{\nu, \alpha}$  denotes the  $\alpha$ -quantile of the (central)  $t$ -distribution with  $\nu$  degrees of freedom (see section 32.13.3). Then confidence intervals for the difference of 2 means can be calculated (needs to be worked out).

### 39.2.6 Tolerance Intervals

Let  $X$  be a normally distributed variable following a  $N(\mu, \sigma^2)$  distribution, with  $\mu, \sigma^2$  unknown, and let  $X_1, X_2, \dots, X_n$  be a random sample. A one-sided tolerance interval covers with confidence  $1 - \alpha$  at least the fraction  $p$  of values of the distribution  $N(\mu, \sigma^2)$ .

Right-hand tolerance interval  $(-\infty, \bar{X} + kS) : P[P(X < \bar{X} + kS) \geq p] = 1 - \alpha]$ ,

Leftt-hand tolerance interval  $(\bar{X} - kS, \infty) : P[P(X > \bar{X} - kS) \geq p] = 1 - \alpha]$ ,

where  $\bar{X}$  and  $S$  denote the estimates of mean and standard deviation from the sample, and

$$k = -\frac{t_\alpha(n-1, -u_p \sqrt{n})}{\sqrt{n}} = \frac{t_{1-\alpha}(n-1, u_p \sqrt{n})}{\sqrt{n}} \quad (39.2.8)$$

$$k = -\frac{t_{n-1, -\delta; \alpha}}{\sqrt{n}} = \frac{t_{n-1, \delta; 1-\alpha}}{\sqrt{n}} \quad (39.2.9)$$

where  $t_{n, \delta; \alpha}$  denotes the  $\alpha$ -quantile of the noncentral  $t$ -distribution with  $n$  degrees of freedom and noncentrality parameter  $\delta$  (see section ,  $\delta = z_p \sqrt{n}$ , and  $z_p$  denotes the  $p$ -quantile of the normal distribution (see section 32.11.3)).

See Janiga, 2006.

See [Odeh & Owen \(1980\)](#)

### 39.2.7 Prediction Intervals

The following formulas are from "06. Statistical Details for the Distribution Platform", which refer to Hahn and Meeker (1991), pages 61-64. But see also 04. Odeh, 1989, for a different definition.

For  $m$  future observations:

$$[\underline{y}_m, \bar{y}_m] = \bar{X} \pm t_{n-1;1-\alpha/(2m)} \times \sqrt{1 + \frac{1}{n}} \times s, \text{ for } m \geq 1. \quad (39.2.10)$$

For the mean of  $m$  future observations:

$$[Y_l, Y_u] = \bar{X} \pm t_{n-1;1-\alpha/2} \times \sqrt{\frac{1}{m} + \frac{1}{n}} \times s, \text{ for } m \geq 1. \quad (39.2.11)$$

where  $m$  is the number of future observations, and  $n$  is the number of points in the current analysis sample. The one-sided intervals are formed by using  $1 - \alpha$  in the quantile functions.

## 39.3 Tests for means from 2 independent samples (Student's t-test)

### 39.3.1 Overview

Let  $(X_1, X_2, \dots, X_N)$  denote a random sample of size  $N$  from a normal distribution with mean  $\mu$  and variance  $\sigma^2$ , and let

$$\bar{x}_1 = \frac{1}{N} \sum_{i=1}^N X_i \quad \text{and} \quad s^2 = \frac{1}{N-1} \sum_{i=1}^N (X_i - \bar{x}_1)^2 \quad (39.3.1)$$

be the usual sample estimates of the unknown population mean  $\mu$  and unknown population variance  $\sigma^2$ . Then Student's t-test can be used to test hypotheses concerning  $\mu$  with regard to a reference value  $\mu_0$ .

### 39.3.2 Tests and Confidence Intervals

---

Function **StudentTTest2i**(*Type1Error* As *mpNum*, **N** As *mpNum*[], **Mean** As *mpNum*[], **StDev** As *mpNum*[], **Output** As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **StudentTTest2i** returns p-values, confidence intervals and related information for Student's t-test

#### Parameters:

*Type1Error*: A real number greater than 0 and less than 1.

*N*: An array of integers greater than 1, representing the samples sizes

*Mean*: An array of reals, representing the means

*StDev*: An array of positive reals, representing the standard deviations

*Output*: A string describing the output choices

See section 39.1.4.1 for the options for *Type1Error* and *Output*). Algorithms and formulas are given in sections 39.3.2.1 and 39.3.2.3.

#### 39.3.2.1 Tests: algorithms and formulas

Let  $F_t(\cdot, \nu)$  denote the CDF (see section 32.13.2.2) and let  $t_{\nu, \alpha}$  denote the  $\alpha$ -quantile (see section 32.13.3) of the *t*-distribution with  $\nu$  degrees of freedom. Define

$$t = \frac{(\bar{x}_1 - \bar{x}_2)}{s}, \quad \text{where} \quad (39.3.2)$$

$$s = \sqrt{MSE \left( \frac{1}{n_1} + \frac{1}{n_2} \right)}, \quad MSE = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{\nu}, \quad \nu = n_1 + n_2 - 2. \quad (39.3.3)$$

Then *p*-values and rejection criteria for  $H_0$  can be calculated as summarized in Table 39.5.

The test can also be expressed in terms of a correlation coefficient  $r$  between the combined  $X$  and an indicator variable, where  $t$  and  $r$  are related by

$$r = \frac{t}{\sqrt{t^2 + \nu}}, \quad t = \nu \frac{r}{1 - r^2}. \quad (39.3.4)$$

Test problem	$p$ -value	Reject $H_0$
$H_{01} : \mu_1 \leq \mu_2$ vs $H_{A1} : \mu_1 > \mu_2$	$F_t(-t, \nu)$	$t > t_{\nu;1-\alpha}$
$H_{02} : \mu_1 \geq \mu_2$ vs $H_{A2} : \mu_1 < \mu_2$	$F_t(t, \nu)$	$t > t_{\nu;\alpha}$
$H_{03} : \mu_1 = \mu_2$ vs $H_{A3} : \mu_1 \neq \mu_2$	$F_t(t, \nu) - F_t(-t, \nu)$	$t > t_{\nu;1-\alpha/2}$ or $t > t_{\nu;\alpha/2}$

**Table 39.5:** Student's t-test, 2 independent samples, tests for the mean

### 39.3.2.2 Tests: examples

An actual call to the function, requesting Student's t-test with description, the critical value for a two-sided test, the p-value for  $H_{03}$  (in the case of TTest this is  $\mu_1 \neq \mu_2$ ), for 2 independent samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

```
Result = Test2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 + PValueH03")
mp>Show Result
```

which produces the output

```
df: 18
Difference of Means: -2.2
t-value: -4.9193496
TCrit2: 2.100922
PValueH03: 0.0001106
```

### 39.3.2.3 Confidence Intervals: algorithms and formulas

Let  $A_1 = t_{\nu,\alpha} \cdot s$  and  $A_2 = t_{\nu,\alpha/2} \cdot s$ , where  $s$  and  $\nu$  are defined in (39.3.3), and  $t_{\nu,\alpha}$  denotes the  $\alpha$ -quantile of the (central)  $t$ -distribution with  $\nu$  degrees of freedom (see section 32.13.3). Then confidence intervals for  $\mu_1 - \mu_2$  can be calculated as summarized in Table 39.6.

Type	Confidence Interval (Difference of Means)
Left-sided	$-\infty \leq \mu_1 - \mu_2 \leq (\bar{x}_1 - \bar{x}_2) + A_1$
Right-sided	$(\bar{x}_1 - \bar{x}_2) - A_1 \leq \mu_1 - \mu_2 \leq +\infty$
Two-sided	$(\bar{x}_1 - \bar{x}_2) - A_2 \leq \mu_1 - \mu_2 \leq (\bar{x}_1 - \bar{x}_2) + A_2$

**Table 39.6:** Student's t-test, 2 independent samples, confidence intervals for the difference of the means

### 39.3.2.4 Confidence Intervals: examples

An actual call to the function, requesting Student's t-test with description, the critical value for a two-sided test, the p-value for  $H_{03}$  (in the case of TTest this is  $\mu_1 \neq \mu_2$ ), for 2 independent samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

```
Result = Test2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 + PValueH03")
mp>Show Result
```

which produces the output

39.3. TESTS FOR MEANS FROM 2 INDEPENDENT SAMPLES (STUDENT'S T-TEST)763

df: 18  
Difference of Means: -2.2  
t-value: -4.9193496  
TCrit2: 2.100922  
PValueH03: 0.0001106

### 39.3.3 Power

---

Function **StudentTTestPower2i**(*Type1Error* As *mpNum*, *N* As *mpNum*[], *Mean* As *mpNum*[], *StDev* As *mpNum*[], *Output* As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **StudentTTestPower2i** returns power estimations and related information for Student's t-test

**Parameters:**

*Type1Error*: An real number greater then 0 and less than 1.

*N*: An array of integers greater than 1, representing the samples sizes

*Mean*: An array of reals, representing the means

*StDev*: An array of positive reals, representing the standard deviations

*Output*: A string describing the output choices

See section 39.1.4.2 for the options for *Type1Error* and *Output*). Algorithms and formulas are given in section 39.2.3.1.

#### 39.3.3.1 Power calculations: algorithms and formulas

Let  $\sigma_1^2 = \sigma_2^2 = \sigma^2$ ,  $N = n_1 + n_2$ ,  $q_1 = n_1/N$ ,  $q_2 = n_2/N$ , and  $\nu = N - 2$ . Define

$$\tilde{\rho} = \frac{\mu_1 - \mu_2}{\sigma} \sqrt{\frac{q_1 + q_2}{q_1 \cdot q_2}}, \text{ and } \delta = \sqrt{N} \tilde{\rho}. \quad (39.3.5)$$

Let  $F_{t'}(\cdot, \nu, \delta)$  denote the CDF of the (singly) noncentral *t*-distribution with  $\nu$  degrees of freedom and noncentrality parameter  $\delta$  (see section and let  $t_{\nu, \alpha}$  denote the  $\alpha$ -quantile of the central *t*-distribution with  $\nu$  degrees of freedom (see section 32.13.3).

Then the power for accepting  $H_A$  at the confidence level  $\alpha$  can be calculated as summarized in Table 39.7.

Test	Null Hypothesis	Alternative	Power
1 sided	$H_{01} : \mu_1 \leq \mu_2$	$H_{A1} : \mu_1 > \mu_2$	$F_{t'}(-t_{\nu;1-\alpha}, \nu, \delta)$
1 sided	$H_{02} : \mu_1 \geq \mu_2$	$H_{A2} : \mu_1 < \mu_2$	$F_{t'}(t_{\nu;1-\alpha}, \nu, \delta)$
2 sided	$H_{03} : \mu_1 = \mu_2$	$H_{A1} : \mu_1 > \mu_2$	$F_{t'}\left(-t_{\nu;1-\alpha/2}, \nu, \delta\right)$
2 sided	$H_{03} : \mu_1 = \mu_2$	$H_{A2} : \mu_1 < \mu_2$	$F_{t'}\left(t_{\nu;1-\alpha/2}, \nu, \delta\right)$
2 sided	$H_{03} : \mu_1 = \mu_2$	$H_{A3} : \mu_1 \neq \mu_2$	$F_{t'}\left(t_{\nu;1-\alpha/2}, \nu, \delta\right) - F_{t'}\left(-t_{\nu;1-\alpha/2}, \nu, \delta\right)$

**Table 39.7:** Student's t-test, 2 independent samples, power calculations

#### 39.3.3.2 Power calculations: examples

An actual call to the function, requesting Student's t-test with description, the critical value for a two-sided test, the power for  $H_{A3}$  (in the case of **TTest** this is  $\mu_1 \neq \mu_2$ ), for 2 independent samples of size 10 and standard deviation 1 each, with means 2.3 and 4.5, and a type I error  $\alpha = 0.05$  would be

---

```
Result = TestPower2i("TTest", 0.05, 10, {2.3, 4.5}, 1, "Description + TCrit2 +
PowerHA3")
```

mp.Print Result

---

which produces the output

```
df: 18
Difference of Means: -2.2
t-value: -4.9193496
TCrit2: 2.100922
PValueH03: 0.996354
```

### 39.3.4 Sample Size Calculation

---

Function **StudentTTestSampleSize2i**( **Type1Error** As *mpNum*, **Type2Error** As *mpNum*, **Mean** As *mpNum*[], **StDev** As *mpNum*[], **Output** As *String*) As *mpNumList*

---

NOT YET IMPLEMENTED

---

The function **StudentTTestSampleSize2i** returns sample size estimations and related information for Student's t-test

**Parameters:**

*Type1Error*: An real number greater then 0 and less than 1.

*Type2Error*: An real number greater then 0 and less than 1.

*Mean*: An array of reals, representing the means

*StDev*: An array of positive reals, representing the standard deviations

*Output*: A string describing the output choices

See section 39.1.4.2 for the options for *Type1Error*, *Type2Error* and *Output*. Algorithms and formulas are given below.

#### 39.3.4.1 Sample size calculations: algorithms and formulas

Let  $N_t'(\alpha, \beta, \tilde{\rho})$  denote the sample size function of the (singly) noncentral  $t$ -distribution (see section 37.8.7) for a given confidence level  $\alpha$ , power  $\beta$  and noncentrality parameter  $\tilde{\rho}$  (as defined in equation 39.3.5).

The required total sample size  $N$  can be calculated as summarized in Table 39.8.

Test	Null Hypothesis	Alternative	Minimal sample size
1 sided	$H_{01} : \mu_1 \leq \mu_2$	$H_{A1} : \mu_1 > \mu_2$	$N_t'(\alpha, \beta, \tilde{\rho})$
1 sided	$H_{02} : \mu_1 \geq \mu_2$	$H_{A2} : \mu_1 < \mu_2$	$N_t'(\alpha, \beta, \tilde{\rho})$
2 sided	$H_{03} : \mu_1 = \mu_2$	$H_{A1} : \mu_1 > \mu_2$	$N_t'(\alpha, \beta, \tilde{\rho})$
2 sided	$H_{03} : \mu_1 = \mu_2$	$H_{A2} : \mu_1 < \mu_2$	$N_t'(\alpha, \beta, \tilde{\rho})$
2 sided	$H_{03} : \mu_1 = \mu_2$	$H_{A3} : \mu_1 \neq \mu_2$	$N_{2t'}(\alpha, \beta, \tilde{\rho})$

**Table 39.8:** Student's t-test, 2 independent samples, sample size calculations

Note that the returned value of  $N$  will in general not be an integer, and rounding up may be required to ensure that all  $n_i$  are integers.

#### 39.3.4.2 Sample size calculations: examples

An actual call to the function, requesting an upper sample size estimate (and actual power) for  $\alpha = 0.95$ ,  $\beta = 0.1$ , and standard deviations  $\sigma_1 = \sigma_2 = 1$ , means  $\mu_1 = 2.3$  and  $\mu_2 = 4.5$ , would be

---

```
Result = TestSampleSize2i("TTest", 0.05, 0.1, {2.3, 4.5}, 1, "UpperN + UpperNPower")
mp.Print Result
```

---

which produces the output

```
UpperN: 2 x 6 = 12
UpperNPower: 0.9285919
```

### 39.3.5 Confidence Interval (Effect size)

Let  $T$  be a statistic according to the non-central  $t$ -distribution with  $n$  degrees of freedom and a non-centrality parameter  $\delta$ . Then the lower confidence limit  $\hat{\delta}$  of level  $1 - \alpha$  and the two-sided confidence interval  $[\underline{\delta}, \bar{\delta}]$  of the non-centrality parameter  $\delta$  of level  $1 - \alpha$  are given by [Akahira et al. \(1995\)](#):

$$\hat{\delta} = bT - z_\alpha \sqrt{k} + hT^3(z_\alpha^2 - 1)/k, \quad (39.3.6)$$

$$\underline{\delta} = bT - z_{\alpha/2} \sqrt{k} + hT^3(z_{\alpha/2}^2 - 1)/k, \quad (39.3.7)$$

$$\bar{\delta} = bT + z_{\alpha/2} \sqrt{k} - hT^3(z_{\alpha/2}^2 - 1)/k, \quad (39.3.8)$$

where  $k = 1 + (1 - b^2)T^2$ ,  $h$  and  $b$  are defined in equation (37.8.19), and  $z_\alpha$  denotes the  $\alpha$ -quantile of the normal distribution (see section 32.11.3).

Let  $A_1 = t_{\nu, \alpha} \cdot s$  and  $A_2 = t_{\nu, \alpha/2} \cdot s$ , where  $s$  and  $\nu$  are defined in (39.3.3), and  $t_{\nu, \alpha}$  denotes the  $\alpha$ -quantile of the (central)  $t$ -distribution with  $\nu$  degrees of freedom (see section 32.13.3). Then confidence intervals for the difference of 2 means can be calculated (needs to be worked out).

# **Part VIII**

## **Appendices**

# Appendix A

## Interfaces to the C family of languages

### A.1 Windows, GNU/Linux, Mac OSX: GNU Compiler Collection

The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

Originally named the GNU C Compiler, because it only handled the C programming language, GCC 1.0 was released in 1987 and the compiler was extended to compile C++ in December of that year.[1] Front ends were later developed for Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others.[3]

As well as being the official compiler of the unfinished GNU operating system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including Linux and the BSD family. A port to RISC OS has also been developed extensively in recent years. There is also an old (3.0) port of GCC to Plan9, running under its ANSI/POSIX Environment (APE).[4] GCC is also available for Microsoft Windows operating systems and for the ARM processor used by many portable devices.

For further information on the GNU Compiler Collection, see [Wikipedia: GCC](#) (the text above has been copied from this reference), or the [GCC Homepage](#).

### A.2 Windows: MSVC

Microsoft Visual C++ (often abbreviated as MSVC or VC++) is a commercial (free version available), integrated development environment (IDE) product from Microsoft for the C, C++, and C++/CLI programming languages. It features tools for developing and debugging C++ code, especially code written for the Microsoft Windows API, the DirectX API, and the Microsoft .NET Framework.

Although the product originated as an IDE for the C programming language, the compiler's support for that language conforms only to the original edition of the C standard, dating from 1989. The later revisions of the standard, C99 and C11, are not supported.[41] According to Herb Sutter, the C compiler is only included for "historical reasons" and is not planned to be further

developed. Users are advised to either use only the subset of the C language that is also valid C++, and then use the C++ compiler to compile their code, or to just use a different compiler such as Intel C++ Compiler or the GNU Compiler Collection instead.[42]

For further information on Microsoft Visual C++, see [Wikipedia: MSVC](#) (the text above has been copied from this reference), or the [MSVC Homepage](#).

## A.3 Windows, GNU/Linux, Mac OSX: C

In computing, C is a general-purpose programming language initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.[5][6] Like most imperative languages in the ALGOL tradition, C has facilities for structured programming and allows lexical variable scope and recursion, while a static type system prevents many unintended operations. Its design provides constructs that map efficiently to typical machine instructions, and therefore it has found lasting use in applications that had formerly been coded in assembly language, most notably system software like the Unix computer operating system.[7]

C is one of the most widely used programming languages of all time,[8][9] and C compilers are available for the majority of available computer architectures and operating systems.

Many later languages have borrowed directly or indirectly from C, including D, Go, Rust, Java, JavaScript, Limbo, LPC, C#, Objective-C, Perl, PHP, Python, Verilog (hardware description language),[4] and Unix's C shell. These languages have drawn many of their control structures and other basic features from C. Most of them (with Python being the most dramatic exception) are also very syntactically similar to C in general, and they tend to combine the recognizable expression and statement syntax of C with underlying type systems, data models, and semantics that can be radically different. C++ and Objective-C started as compilers that generated C code; C++ is currently nearly a superset of C,[10] while Objective-C is a strict superset of C.[11]

Before there was an official standard for C, many users and implementors relied on an informal specification contained in a book by Dennis Ritchie and Brian Kernighan; that version is generally referred to as "K&R" C. In 1989 the American National Standards Institute published a standard for C (generally called "ANSI C" or "C89"). The next year, the same specification was approved by the International Organization for Standardization as an international standard (generally called "C90"). ISO later released an extension to the internationalization support of the standard in 1995, and a revised standard (known as "C99") in 1999. The current version of the standard (now known as "C11") was approved in December 2011.[12]

For further information on C++, see [Wikipedia: C](#) (the text above has been copied from this reference), or the [Wikipedia: GCC](#) (the text above has been copied from this reference), or the [GCC Homepage](#).

Example in C

---

```
#include <iostream>
#include "mpreal.h"

using mpfr::mpreal;
using std::cout;
using std::endl;

// double - version
double schwefel(double x)
{
    return 418.9829 - x * sin(sqrt(abs(x)));
}

//MPFR C - version
void mpfr_schwefel(mpfr_t y, mpfr_t x)
```

```

{
mpfr_t t;
mpfr_init(t);
mpfr_abs(t,x,GMP_RNDN);
mpfr_sqrt(t,t,GMP_RNDN);
mpfr_sin(t,t,GMP_RNDN);
mpfr_mul(t,t,x,GMP_RNDN);
mpfr_set_str(y,"418.9829",10,GMP_RNDN);
mpfr_sub(y,y,t,GMP_RNDN);
mpfr_clear(t);
}

// MPFR C++ - version
mpreal mpfr_schwefel(mpreal& x)
{
return "418.9829" - x*sin(sqrt(abs(x)));
}

int main(int argc, char* argv[])
{
const int digits = 50;
mpreal::set_default_prec(mpfr::digits2bits(digits));
const mpreal pi      = mpfr::const_pi();
mpreal x      = "-343.5";
mpreal SResult = mpfr_schwefel(x);
cout.precision(digits); // Show all the digits
cout << "pi      = " << pi      << endl;
cout << "SResult = " << SResult << endl;
return 0;
}

```

---

## A.4 Windows, GNU/Linux, Mac OSX: C++

C++ (pronounced "see plus plus") is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is regarded as an intermediate-level language, as it comprises both high-level and low-level language features.[3] Developed by Bjarne Stroustrup starting in 1979 at Bell Labs, C++ was originally named C with Classes, adding object oriented features, such as classes, and other enhancements to the C programming language. The language was renamed C++ in 1983,[4] as a pun involving the increment operator.

C++ is one of the most popular programming languages[5][6] and is implemented on a wide variety of hardware and operating system platforms. As an efficient compiler to native code, its application domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games.[7] Several groups provide both free and proprietary C++ compiler software, including the GNU Project, LLVM, Microsoft, Intel and Embarcadero Technologies. C++ has greatly influenced many other popular programming languages, most notably C# and Java.

C++ is also used for hardware design, where the design is initially described in C++, then analyzed, architecturally constrained, and scheduled to create a register-transfer level hardware description language via high-level synthesis.[8]

The language began as enhancements to C, first adding classes, then virtual functions, operator overloading, multiple inheritance, templates and exception handling, among other features. After years of development, the C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. The standard was amended by the 2003 technical corrigendum, ISO/IEC 14882:2003. The current standard extending C++ with new features was ratified and published by ISO in September 2011 as ISO/IEC 14882:2011 (informally known as C++11).[9]

For further information on C++, see [Wikipedia: C++](#) (the text above has been copied from this reference), or the [GCC Homepage](#).

Example in C++

---

```
#include <iostream>
#include "mpreal.h"

int main(int argc, char* argv[])
{
using mpfr::mpreal;
using std::cout;
using std::endl;

// Required precision of computations in decimal digits
// Play with it to check different precisions
const int digits = 50;

// Setup default precision for all subsequent computations
// MPFR accepts precision in bits - so we do the conversion
mpreal::set_default_prec(mpfr::digits2bits(digits));

// Compute all the vital characteristics of mpreal (in current precision)
```

```

// Analogous to lamch from LAPACK
const mpreal one      = 1.0;
const mpreal zero     = 0.0;
const mpreal eps      = std::numeric_limits<mpreal>::epsilon();
const int   base      = std::numeric_limits<mpreal>::radix;
const mpreal prec     = eps * base;
const int   bindigits = std::numeric_limits<mpreal>::digits(); // eqv. to
                     mpfr::mpreal::get_default_prec();
const mpreal rnd      = std::numeric_limits<mpreal>::round_error();
const mpreal maxval   = std::numeric_limits<mpreal>::max();
const mpreal minval   = std::numeric_limits<mpreal>::min();
const mpreal small    = one / maxval;
const mpreal sfmin    = (small > minval) ? small * (one + eps) : minval;
const mpreal round    = std::numeric_limits<mpreal>::round_style();
const int   min_exp   = std::numeric_limits<mpreal>::min_exponent;
const mpreal underflow = std::numeric_limits<mpreal>::min();
const int   max_exp   = std::numeric_limits<mpreal>::max_exponent;
const mpreal overflow = std::numeric_limits<mpreal>::max();

// Additionally compute pi with required accuracy - just for fun :)
const mpreal pi        = mpfr::const_pi();

cout.precision(digits); // Show all the digits
cout << "pi      = " << pi      << endl;
cout << "eps     = " << eps     << endl;
cout << "base    = " << base    << endl;
cout << "prec    = " << prec    << endl;
cout << "b.digits = " << bindigits << endl;
cout << "rnd     = " << rnd     << endl;
cout << "maxval   = " << maxval   << endl;
cout << "minval   = " << minval   << endl;
cout << "small    = " << small    << endl;
cout << "sfmin    = " << sfmin    << endl;
cout << "1/sfmin  = " << 1 / sfmin << endl;
cout << "round    = " << round    << endl;
cout << "max_exp  = " << max_exp  << endl;
cout << "min_exp  = " << min_exp  << endl;
cout << "underflow = " << underflow << endl;
cout << "overflow = " << overflow << endl;

return 0;
}

```

---

## A.5 Mac OSX: Objective C

Objective-C is a general-purpose, object-oriented programming language that adds Smalltalk-style messaging to the C programming language. It is the main programming language used by Apple for the OS X and iOS operating systems, and their respective application programming interfaces (APIs), Cocoa and Cocoa Touch.

The programming language Objective-C was originally developed in the early 1980s. It was selected as the main language used by NeXT for its NeXTSTEP operating system, from which OS X and iOS are derived.[2] Generic Objective-C programs that do not use the Cocoa or Cocoa Touch libraries, or using parts that may be ported or reimplemented for other systems can also be compiled for any system supported by GCC or Clang.

Objective-C source code program files usually have .m filename extensions, while Objective-C header files have .h extensions, the same as for C header files. Objective-C++ files are denoted with a .mm file extension.

For further information on C++, see [Wikipedia: Objective C](#) (the text above has been copied from this reference), or the [GCC Homepage](#).

Example in Objective C

---

```
# import "Forwarder.h"
# import "Recipient.h"

int main(void) {
Forwarder *forwarder = [Forwarder new];
Recipient *recipient = [Recipient new];

[forwarder setRecipient:recipient]; //Set the recipient.
/*
 * Observe forwarder does not respond to a hello message! It will
 * be forwarded. All unrecognized methods will be forwarded to
 * the recipient
 * (if the recipient responds to them, as written in the Forwarder)
 */
[forwarder hello];

[recipient release];
[forwarder release];

return 0;
}
```

---

## A.6 Mac OSX: Objective C++

Objective-C++ is a language variant accepted by the front-end to the GNU Compiler Collection and Clang, which can compile source files that use a combination of C++ and Objective-C syntax. Objective-C++ adds to C++ the extensions that Objective-C adds to C. As nothing is done to unify the semantics behind the various language features, certain restrictions apply:

A C++ class cannot derive from an Objective-C class and vice versa.

C++ namespaces cannot be declared inside an Objective-C declaration.

Objective-C declarations may appear only in global scope, not inside a C++ namespace

Objective-C classes cannot have instance variables of C++ classes that do not have a default constructor or that have one or more virtual methods,[citation needed] but pointers to C++ objects can be used as instance variables without restriction (allocate them with new in the -init method).

C++ "by value" semantics cannot be applied to Objective-C objects, which are only accessible through pointers.

An Objective-C declaration cannot be within a C++ template declaration and vice versa. However, Objective-C types, (e.g., Classname \*) can be used as C++ template parameters.

Objective-C and C++ exception handling is distinct; the handlers of each cannot handle exceptions of the other type. This is mitigated in recent runtimes as Objective-C exceptions are either replaced by C++ exceptions completely (Apple runtime), or partly when Objective-C++ library is linked (GNUstep libobjc2).

Care must be taken since the destructor calling conventions of Objective-C and C++'s exception run-time models do not match (i.e., a C++ destructor will not be called when an Objective-C exception exits the C++ object's scope). The new 64-bit runtime resolves this by introducing interoperability with C++ exceptions in this sense.[15]

Objective-C blocks and C++11 lambdas are distinct entities, however a block is transparently generated on Mac OS X when passing a lambda where a block is expected.[16]

Objective-C++ is Objective-C (probably with COCOA Framework) with the ability to link with C++ code (probable classes).

Yes, you can use this language in XCODE to develop for Mac OS X, iPhone/iPodTouch, iPad. It works very well.

You don't have to do anything weird in your project to use Objective-C++. Just name your Objective-C files with the extension .mm (instead of .m) and you are good to go.

It is my favorite architecture: develop base class library of my game/application in C++ so I can reuse it in other platforms (Windows, Linux) and use COCOA just for the iPhone/iPad UI specific stuff.

For further information on Objective C++, see [Wikipedia: C++](#) (the text above has been copied from this reference), or the [GCC Homepage](#).

Example in C++

---

```
#include <iostream>
#include "mpreal.h"

int main(int argc, char* argv[])
{
    using mpfr::mpreal;
```

```

using std::cout;
using std::endl;

// Required precision of computations in decimal digits
// Play with it to check different precisions
const int digits = 50;

// Setup default precision for all subsequent computations
// MPFR accepts precision in bits - so we do the conversion
mpreal::set_default_prec(mpfr::digits2bits(digits));

// Compute all the vital characteristics of mpreal (in current precision)
// Analogous to lamch from LAPACK
const mpreal one      = 1.0;
const mpreal zero     = 0.0;
const mpreal eps      = std::numeric_limits<mpreal>::epsilon();
const int   base       = std::numeric_limits<mpreal>::radix;
const mpreal prec      = eps * base;
const int bindigits    = std::numeric_limits<mpreal>::digits(); // eqv. to
                     mpfr::mpreal::get_default_prec();
const mpreal rnd       = std::numeric_limits<mpreal>::round_error();
const mpreal maxval    = std::numeric_limits<mpreal>::max();
const mpreal minval    = std::numeric_limits<mpreal>::min();
const mpreal small     = one / maxval;
const mpreal sfmin     = (small > minval) ? small * (one + eps) : minval;

// Additionally compute pi with required accuracy - just for fun :)
const mpreal pi        = mpfr::const_pi();

cout.precision(digits); // Show all the digits
cout << "pi      = " << pi      << endl;
cout << "eps     = " << eps     << endl;
cout << "base    = " << base    << endl;
cout << "prec    = " << prec    << endl;
cout << "b.digits = " << bindigits << endl;
cout << "rnd     = " << rnd     << endl;
cout << "maxval  = " << maxval  << endl;
cout << "minval  = " << minval  << endl;
cout << "small   = " << small   << endl;
cout << "sfmin   = " << sfmin   << endl;
cout << "1/sfmin = " << 1 / sfmin << endl;

return 0;
}

```

---

# Appendix B

## Languages with CLR Support

### B.1 Visual Basic .NET

Visual Basic .NET (VB.NET) is an object-oriented computer programming language that can be viewed as an evolution of the classic Visual Basic (VB), implemented on the .NET Framework. Microsoft currently supplies two main editions of IDEs for developing in Visual Basic: Microsoft Visual Studio 2012, which is commercial software and Visual Basic Express Edition 2012, which is free of charge. The command-line compiler, VBC.EXE, is installed as part of the freeware .NET Framework SDK. Mono also includes a command-line VB.NET compiler.

For further information on Visual Basic .NET, see [Wikipedia: Visual Basic .NET](#) (the text above has been copied from this reference).

Example for using the library

---

```
Imports System
Imports System.Console
Imports Microsoft.VisualBasic
Imports Microsoft.VisualBasic.Strings
Imports MatrixClass2

Module Module1
Sub Main()
mp.Prec10() = 100 : mp.FloatingPointType() = 3
Dim Y1, Y2, Y3, Y4 As New mpNum
Y1 = mp.Sqrt(2)
Writeline("#Sqrt(12): ")
Writeline("@" & Y1)
Y2 = Sqrt(2)
Writeline("#Sqrt(12): ")
Writeline("@" & Y2)
Y3 = Y1 - Y2
Y4 = Y3 + CNum("1.4")
Writeline("#Diff:")
Writeline("@" & Y4)
End Sub
End Module
```

---

Example for using Excel

---

```
Imports System
Imports System.Console
Imports Microsoft.VisualBasic
Imports Microsoft.VisualBasic.Strings

Module Module1
Sub DemoExcel()
Dim objExcel As Object
objExcel = CreateObject("Excel.Application")
'objExcel.Workbooks.Open("C:\Extra\mpNumerics\Output\mpTemp00.html")
objExcel.Visible = True
objExcel.Workbooks.Add
objExcel.Cells(1, 1).Value = "Test value"
objExcel = Nothing
End Sub

Sub Main()
Call DemoExcel()
End Sub
End Module
```

---

Example for using Forms

---

```
Imports System.Windows.Forms

Partial Class MyForm : Inherits Form
    'Component's Declaration
    Friend WithEvents lblFirstName As Label = New Label
    Friend WithEvents lblLastName As Label = New Label
    Friend WithEvents txtFirstName As TextBox = New TextBox
    Friend WithEvents txtLastName As TextBox = New TextBox
    Friend WithEvents btnShow As Button = New Button

    Private Sub InitializeComponent()
        Me.Text = "My Second Example Form"

        'lblFirstName Setting
        lblFirstName.Text = "First Name : "
        'Set the label into AutoSize
        lblFirstName.AutoSize = True
        'Set the location/position of the lblFirstName Object relative to the form
        'System.Drawing.Point(x, y)
        lblFirstName.Location = New System.Drawing.Point(10, 10)

        'lblLastName Setting
        lblLastName.Text = "Last Name : "
        lblLastName.AutoSize = True
        lblLastName.Location = New System.Drawing.Point(10, 60)

        'txtFirstName Setting
        txtFirstName.MaxLength = 50
        txtFirstName.Size = New System.Drawing.Size(150, 40)
        txtFirstName.Location = New System.Drawing.Point(100, 10)

        'txtLastName Setting
        txtLastName.MaxLength = 50
        txtLastName.Size = New System.Drawing.Size(150, 40)
        txtLastName.Location = New System.Drawing.Point(100, 60)

        'btnShow Setting
        btnShow.Text = "&Show"
        btnShow.Size = New System.Drawing.Size(50, 30)
        btnShow.Location = New System.Drawing.Point(10, 100)

        'Adding the control/component into the Form
        Me.Controls.Add(lblLastName)
        Me.Controls.Add(lblFirstName)
        Me.Controls.Add(txtLastName)
        Me.Controls.Add(txtFirstName)
        Me.Controls.Add(btnShow)
        Me.Size = New System.Drawing.Size(txtLastName.Right + 20, btnShow.Top + 70)
        Me.StartPosition = FormStartPosition.CenterScreen
```

```
End Sub

Private Sub btnShow_Clicked(ByVal sender As System.Object, ByVal e As
    System.EventArgs) Handles btnShow.Click
    MessageBox.Show("Welcome " & txtFirstName.Text & " " & txtLastName.Text, "Welcome")
End Sub

Public Sub New()
    InitializeComponent()
End Sub

End Class

Module Module1

    Function Main(ByVal cmdArgs() As String) As Integer
        Application.EnableVisualStyles()
        Dim theForm As New MyForm
        theForm.ShowDialog()
        Return 0
    End Function

End Module
```

---

Example for using .NET Charts

---

```
Imports System.Windows.Forms
Imports System.Windows.Forms.DataVisualization.Charting

Module Module1

Function Main(ByVal cmdArgs() As String) As Integer
Dim Chart1 As System.Windows.Forms.DataVisualization.Charting.Chart
Chart1 = New Chart()
Dim chartArea1 As New ChartArea()
Chart1.ChartAreas.Add("Default")
Chart1.Series.Add("Default")

' Populate series data
Dim yValues As Double() = {65.62, 75.54, 60.45, 34.73, 85.42}
Dim xValues As String() = {"France", "Canada", "Germany", "USA", "Italy"}
Chart1.Series("Default").Points.DataBindXY(xValues, yValues)

' Set Doughnut chart type
Chart1.Series("Default").ChartType = SeriesChartType.Doughnut

' Set labels style
Chart1.Series("Default")("PieLabelStyle") = "Outside"

' Set Doughnut radius percentage
Chart1.Series("Default")("DoughnutRadius") = "60"

' Explode data point with label "Italy"
Chart1.Series("Default").Points(4)("Exploded") = "true"

' Enable 3D
Chart1.ChartAreas("Default").Area3DStyle.Enable3D = false

' Set drawing style
chart1.Series("Default")("PieDrawingStyle") = "SoftEdge"

' Set Chart control size
Chart1.Size = New System.Drawing.Size(360, 260)

Dim FileName As String
FileName = cmdArgs(0)
'FileName = "I:\mpNew\mpNumerics\VBNET.emf"
'Chart1.SaveImage(FileName, ChartImageFormat.EmfDual)
Chart1.Serializer.Save(FileName)
Return 0
End Function

End Module
```

---

Example for using the speech synthesizer

---

```
Imports System.Windows.Forms
Imports System.Speech.Synthesis

Module Module1

Function Main(ByVal cmdArgs() As String) As Integer
Dim speaker as New SpeechSynthesizer()
speaker.Rate = 1
speaker.Volume = 100
speaker.Speak("Hello world")
speaker.SetOutputToWaveFile("c:\soundfile.wav")
speaker.Speak("Hello world")
speaker.SetOutputToDefaultAudioDevice()
'Must remember to reset out device or the next call to speak
'will try to write to a file
End Function

End Module
```

---

Example for using Matlab as a COM Server from Visual Basic

This example calls a user-defined MATLAB function named solve\_bvp from a Microsoft Visual Basic client application through a COM interface. It also plots a graph in a new MATLAB window and performs a simple computation:

---

```
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MIImag(1, 3) As Double

MatLab = CreateObject("Matlab.Application")

'Calling MATLAB function from VB
'Assuming solve_bvp exists at specified location
Result = MatLab.Execute("cd d:\matlab\work\bvp")
Result = MatLab.Execute("solve_bvp")

'Executing other MATLAB commands
Result = MatLab.Execute("surf(peaks)")
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8]")
Result = MatLab.Execute("b = a + a ")
'Bring matrix b into VB program
MatLab.GetFullMatrix("b", "base", MReal, MIImag)
```

---

The following examples require NetOffice to be installed.

Example for calling Excel using NetOffice

---

```

Imports NetOffice
' Imports Office = NetOffice.OfficeApi
Imports Excel = NetOffice.ExcelApi
' Imports NetOffice.ExcelApi.Enums

Module Program

Private Sub GetActiveExcel()
Dim xlProxy As Object =
    System.Runtime.InteropServices.Marshal.GetActiveObject("Excel.Application")
Dim xlApp As Excel._Application = New Excel._Application(Nothing, xlProxy)
Dim workBook As Excel.Workbook = xlApp.ActiveWorkbook
Dim workSheet As Excel.Worksheet = xlApp.ActiveSheet
Dim wbName As String = workBook.Name
System.Console.WriteLine(wbName)
' VERY IMPORTANT! OTHERWISE LATER CALLS WILL FAIL!
xlApp.Dispose()
End Sub

Sub Main()
GetActiveExcel()
End Sub

End Module

```

---

Example for calling Word using NetOffice

---

```

Imports NetOffice
Imports Office = NetOffice.OfficeApi
Imports Word = NetOffice.WordApi
Imports NetOffice.WordApi.Enums

Module Program

Private Sub GetActiveWord()
Dim wdProxy As Object =
    System.Runtime.InteropServices.Marshal.GetActiveObject("Word.Application")
Dim wdApp As Word._Application = New Word._Application(Nothing, wdProxy)
' VERY IMPORTANT! OTHERWISE LATER CALLS WILL FAIL!
wdApp.Dispose()
End Sub

Sub Main()
GetActiveWord()
End Sub

End Module

```

---

Example for calling PowerPoint using NetOffice

---

```
Imports NetOffice
Imports Office = NetOffice.OfficeApi
Imports PowerPoint = NetOffice.PowerPointApi
Imports NetOffice.PowerPointApi.Enums

Module Program

Private Sub GetActivePowerpoint()
Dim ppProxy As Object =
    System.Runtime.InteropServices.Marshal.GetActiveObject("Powerpoint.Application")
Dim ppApp As Powerpoint._Application = New Powerpoint._Application(Nothing, ppProxy)

' VERY IMPORTANT! OTHERWISE LATER CALLS WILL FAIL!
ppApp.Dispose()
End Sub

Sub Main()
GetActivePowerpoint()
End Sub

End Module
```

---

## B.2 C# 4.0

C# C#[note 1] (pronounced see sharp) is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, procedural, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed by Microsoft within its .NET initiative and later approved as a standard by Ecma (ECMA-334) and ISO (ISO/IEC 23270:2006). C# is one of the programming languages designed for the Common Language Infrastructure.

C# is intended to be a simple, modern, general-purpose, object-oriented programming language.[6] Its development team is led by Anders Hejlsberg. The most recent version is C# 5.0, which was released on August 15, 2012.

C# has the following syntax:

Semicolons are used to denote the end of a statement. Curly braces are used to group statements. Statements are commonly grouped into methods (functions), methods into classes, and classes into namespaces. Variables are assigned using an equals sign, but compared using two consecutive equals signs. Square brackets are used with arrays, both to declare them and to get a value at a given index in one of them

By design, C# is the programming language that most directly reflects the underlying Common Language Infrastructure (CLI).[30] Most of its intrinsic types correspond to value-types implemented by the CLI framework. However, the language specification does not state the code generation requirements of the compiler: that is, it does not state that a C# compiler must target a Common Language Runtime, or generate Common Intermediate Language (CIL), or generate any other specific format. Theoretically, a C# compiler could generate machine code like traditional compilers of C++ or Fortran. Some notable features of C# that distinguish it from C and C++ (and Java, where noted) are:

C# supports strongly typed implicit variable declarations with the keyword `var`, and implicitly typed arrays with the keyword `new[]` followed by a collection initializer. Meta programming via C# attributes is part of the language. Many of these attributes duplicate the functionality of GCC's and VisualC++'s platform-dependent preprocessor directives.

Like C++, and unlike Java, C# programmers must use the keyword `virtual` to allow methods to be overridden by subclasses. Extension methods in C# allow programmers to use static methods as if they were methods from a class's method table, allowing programmers to add methods to an object that they feel should exist on that object and its derivatives.

The type dynamic allows for run-time method binding, allowing for JavaScript like method calls and run-time object composition. C# has strongly typed and verbose function pointer support via the keyword `delegate`.

Like the Qt framework's pseudo-C++ signal and slot, C# has semantics specifically surrounding publish-subscribe style events, though C# uses delegates to do so. C# offers Java-like synchronized method calls, via the attribute `[MethodImpl(MethodImplOptions.Synchronized)]`, and has support for mutually-exclusive locks via the keyword `lock`. The C# language does not allow for global variables or functions. All methods and members must be declared within classes. Static members of public classes can substitute for global variables and functions.

Local variables cannot shadow variables of the enclosing block, unlike C and C++.

A C# namespace provides the same level of code isolation as a Java package or a C++ namespace,

with very similar rules and features to a package. C# supports a strict Boolean data type, `bool`. Statements that take conditions, such as `while` and `if`, require an expression of a type that implements the `true` operator, such as the `boolean` type. While C++ also has a `boolean` type, it can be freely converted to and from integers, and expressions such as `if(a)` require only that `a` is convertible to `bool`, allowing `a` to be an `int`, or a pointer. C# disallows this "integer meaning true or false" approach, on the grounds that forcing programmers to use expressions that return exactly `bool` can prevent certain types of programming mistakes common in C or C++ such as `if(a = b)` (use of assignment = instead of equality ==).

In C#, memory address pointers can only be used within blocks specifically marked as `unsafe`, and programs with `unsafe` code need appropriate permissions to run. Most object access is done through safe object references, which always either point to a "live" object or have the well-defined `null` value; it is impossible to obtain a reference to a "dead" object (one that has been garbage collected), or to a random block of memory. An `unsafe` pointer can point to an instance of a value-type, array, string, or a block of memory allocated on a stack. Code that is not marked as `unsafe` can still store and manipulate pointers through the `System.IntPtr` type, but it cannot dereference them. Managed memory cannot be explicitly freed; instead, it is automatically garbage collected. Garbage collection addresses the problem of memory leaks by freeing the programmer of responsibility for releasing memory that is no longer needed.

In addition to the `try...catch` construct to handle exceptions, C# has a `try...finally` construct to guarantee execution of the code in the `finally` block, whether an exception occurs or not.

Multiple inheritance is not supported, although a class can implement any number of interfaces. This was a design decision by the language's lead architect to avoid complication and simplify architectural requirements throughout CLI. When implementing multiple interfaces that contain a method with the same signature, C# allows the programmer to implement each method depending on which interface that method is being called through, or, like Java, allows the programmer to implement the method once and have that be the single invocation on a call through any of the class's interfaces.

C#, unlike Java, supports operator overloading. Only the most commonly overloaded operators in C++ may be overloaded in C#. C# is more type safe than C++. The only implicit conversions by default are those that are considered safe, such as widening of integers. This is enforced at compile-time, during JIT, and, in some cases, at runtime. No implicit conversions occur between booleans and integers, nor between enumeration members and integers (except for literal 0, which can be implicitly converted to any enumerated type). Any user-defined conversion must be explicitly marked as explicit or implicit, unlike C++ copy constructors and conversion operators, which are both implicit by default.

C# has explicit support for covariance and contravariance in generic types, unlike C++ which has some degree of support for contravariance simply through the semantics of return types on virtual methods.

Enumeration members are placed in their own scope. C# provides properties as syntactic sugar for a common pattern in which a pair of methods, accessor (getter) and mutator (setter) encapsulate operations on a single attribute of a class. No redundant method signatures for the getter/setter implementations need be written, and the property may be accessed using attribute syntax rather than more verbose method calls.

Checked exceptions are not present in C# (in contrast to Java). This has been a conscious

decision based on the issues of scalability and versionability. For further information on C#, see [Wikipedia: C#](#) (the text above has been copied from this reference)

Example for using the library

---

```
using System;
using System.Collections.Generic;
using System.Text;
using MatrixClass2;

namespace ConsoleSimple
{
    class Program
    {
        static void Main(string[] args)
        {
            mp.Prec10 = 339;
            mp.FloatingPointType = 3;
            double x1 = 15.0;
            mpNum Y1 = "5.12";
            mpNum Y2 = Y1 * x1;
            mpNum Y3 = mp.Sqrt(Y1);
            Console.WriteLine(" x1: " + x1 + "; Y1: " );
            Console.WriteLine("@" + Y1.Str());
            Console.WriteLine(" Y2: " + Y2.Str() + "; Y3: " );
            Console.WriteLine("@" + Y3.Str());
        }
    }
}
```

---

Example for using Excel

---

```
using System;

namespace DemoExcel
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic xlApp = Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));
            xlApp.Visible = true;
            xlApp.Workbooks.Add();
            xlApp.Cells(1, 1).Value = "Test value";
        }
    }
}
```

---

## B.3 JScript 10.0

JScript .NET is a .NET programming language developed by Microsoft.

The primary differences between JScript and JScript .NET can be summarized as follows:

Firstly, JScript is a scripting language, and as such programs (or more suggestively, scripts) can be executed without the need to compile the code first. This is not the case with the JScript .NET command-line compiler, since this next-generation version relies on the .NET Common Language Runtime (CLR) for execution, which requires that the code be compiled to Common Intermediate Language (CIL), formerly called Microsoft Intermediate Language (MSIL), code before it can be run. Nevertheless, JScript .NET still provides full support for interpreting code at runtime (e.g., via the Function constructor or the eval function) and indeed the interpreter can be exposed by custom applications hosting the JScript .NET engine via the VSA[jargon] interfaces.

Secondly, JScript has a strong foundation in Microsoft's ActiveX/COM technologies, and relies primarily on ActiveX components to provide much of its functionality (including database access via ADO, file handling, etc.), whereas JScript .NET uses the .NET Framework to provide equivalent functionality. For backwards-compatibility (or for where no .NET equivalent library exists), JScript .NET still provides full access to ActiveX objects via .NET / COM interop using both the ActiveXObject constructor and the standard methods of the .NET Type class.

Although the .NET Framework and .NET languages such as C# and Visual Basic .NET have seen widespread adoption, JScript .NET has never received much attention, by the media or by developers. It is not supported in Microsoft's premier development tool, Visual Studio .NET. However, ASP.NET supports JScript .NET.

For further details, see [Wikipedia: JScript.NET](#) (the text above has been copied from this reference).

Example for using the library:

---

```
//Load the mpNumerics library
import MatrixClass2;

//Set Floating point type to MPFR with 60 decimal digits precision
mp.FloatingPointType = 3;
mp.Prec10 = 60;

//Assign values to x1 and x2
var x1 = mp.CNum("32.47");
var x2 = mp.CNum("12.41");

//Calculate x3 = x1 / x2
var x3 = x1 / x2;

//Print the value of x3
print("Result: ", x3.Str());
```

---

Example for using Excel:

---

```
// Declare the variables
var Excel, Book;

// Create the Excel application object.
Excel = new ActiveXObject("Excel.Application");
```

```
// Make Excel visible.  
Excel.Visible = true;  
  
// Create a new work book.  
Book = Excel.Workbooks.Add()  
  
// Place some text in the first cell of the sheet.  
Book.ActiveSheet.Cells(1,1).Value = "This is column A, row 1";  
  
// Save the sheet.  
Book.SaveAs("C:\\TEST.XLS");  
  
// Close Excel with the Quit method on the Application object.  
Excel.Application.Quit();
```

---

## B.4 C++ 10.0, Visual Studio

C++/CLI (Common Language Infrastructure) is a language specification created by Microsoft and intended to supersede Managed Extensions for C++. It is a complete revision that aims to simplify the older Managed C++ syntax, which is now deprecated.[1] C++/CLI was standardized by Ecma as ECMA-372. It is currently available in Visual Studio 2005, 2008, 2010 and 2012, including the Express editions.

Syntax changes[edit]C++/CLI should be thought of as a language of its own (with a new set of keywords, for example), instead of the C++ superset-oriented Managed C++. Because of this, there are some major syntactic changes, especially related to the elimination of ambiguous identifiers and the addition of .NET-specific features.

Many conflicting syntaxes, such as the multiple versions of operator new() in MC++ have been split: in C++/CLI, .NET reference types are created with the new keyword gcnew. Also, C++/CLI has introduced the concept of generics (conceptually similar to standard C++ templates, but quite different in their implementation).

In C++/CLI the only type of pointer is the normal C++ pointer, and the .NET reference types are accessed through a handle, with the new syntax `ClassName^` instead of `ClassName`. This new construct is especially helpful when managed and standard C++ code is mixed; it clarifies which objects are under .NET automatic garbage collection and which objects the programmer must remember to explicitly destroy.

Operator overloading works analogously to standard C++. Every `*` becomes a `;` every `&` becomes an `%`, but the rest of the syntax is unchanged, except for an important addition: Operator overloading is possible not only for classes themselves, but also for references to those classes. This feature is necessary to give a ref class the semantics for operator overloading expected from .NET ref classes. In reverse, this also means that for .Net framework ref classes, reference operator overloading often is implicitly implemented in C++/CLI.

For further information, see [Wikipedia: C++/CLI](#) (the text above has been copied from this reference).

Example for using the library

---

```
// compile with: /clr
using namespace System;
using namespace MatrixClass2;

int main()
{
    mp^ MP = gcnew mp;
    MP->Prec10 = 30;
    MP->FloatingPointType = 3;
    mpNum^ x1 = gcnew mpNum;
    x1 = "3.4";
    mpNum^ x2 = gcnew mpNum;
    x2 = "13.4";
    mpNum^ x3 = gcnew mpNum;
    x3 = x1 / x2;
    String^ Result = x1->Str();
    Console::WriteLine("Result: {0} ", Result);
    return 0;
}
```

---

Example for mixing managed and unmanaged code

---

```
// compile with: /clr
using namespace System;
using namespace MatrixClass2;

int main()
{
// pragma_directives_managed_unmanaged.cpp
// compile with: /clr
#include <stdio.h>
#include <iostream>

// func1 is managed
void func1() {
System::Console::WriteLine("In managed C++ function (func1).");
}

// #pragma unmanaged
// push managed state on to stack and set unmanaged state
#pragma managed(push, off)

// func2 is unmanaged
void func2() {
printf("In unmanaged C function (func2).\n");
}

// func3 is unmanaged
void func3() {
std::cout << "In unmanaged C++ function (func3)." << std::endl;
}

// #pragma managed
#pragma managed(pop)

// main is managed
int main() {
func1();
func2();
func3();
}
```

---

## B.5 F# 3.0

F# (pronounced F Sharp) is a strongly typed, multi-paradigm programming language encompassing functional, imperative and object-oriented programming techniques. F# is most often used as a cross-platform CLI language, but can also be used to generate JavaScript[3] and GPU[4] code.

F# is developed by the F# Software Foundation,[5] Microsoft and open contributors. An open source, cross-platform compiler for F# is available from the F# Software Foundation.[6] F# is also a fully supported language in Visual Studio.[7] Other tools supporting F# development[clarification needed] include Mono, MonoDevelop, SharpDevelop and WebSharper

F# originated as a variant of ML and has been influenced by OCaml, C#, Python, Haskell,[2] Scala and Erlang.

For further information, see the [F# Homepage](#) or [Wikipedia: F#](#) (the text above has been copied from this reference).

Example for using the library

---

```

open System.Windows.Forms
open MatrixClass2

// Create a window and set a few properties
let form = new Form(Visible=true, TopMost=true, Text="Welcome to F#")

// mp.FloatingPointType = 3 // Does not work, need mp.SetFloatType(3)

let x1 = mp.CNum("32.47")
let x2 = mp.CNum("32.47")
let x3 = x1 + x2
let s = x3.Str()

let label = new Label(Text = s)

// Add the label to the form
form.Controls.Add(label)

// Finally, run the form
[<System.STAThread>]
Application.Run(form)

```

---

Example for using functions

---

```

/// Iteration using a 'for' loop
let printList lst =
  for x in lst do
    printfn "%d" x

/// Iteration using a higher-order function
let printList2 lst =
  List.iter (printfn "%d") lst

/// Iteration using a recursive function and pattern matching

```

---

```
let rec printList3 lst =
  match lst with
  | [] -> ()
  | h :: t ->
    printfn "%d" h
    printList3 t
```

## B.6 IronPython 2.7

IronPython is an implementation of the Python programming language targeting the .NET Framework and Mono. Jim Hugunin created the project and actively contributed to it up until Version 1.0 which was released on September 5, 2006.[2] Thereafter, it was maintained by a small team at Microsoft until the 2.7 Beta 1 release; Microsoft abandoned IronPython (and its sister project IronRuby) in late 2010, after which Hugunin left to work at Google.[3] IronPython 2.0 was released on December 10, 2008.[4] The project is currently maintained by a group of volunteers at Microsoft's CodePlex open-source repository. It is free and open-source software, and can be implemented with Python Tools for Visual Studio, which is a free and open-source extension for free, isolated, and commercial versions of Microsoft's Visual Studio IDE.[5] [6]

IronPython is written entirely in C#, although some of its code is automatically generated by a code generator written in Python.

IronPython is implemented on top of the Dynamic Language Runtime (DLR), a library running on top of the Common Language Infrastructure that provides dynamic typing and dynamic method dispatch, among other things, for dynamic languages.[7] The DLR is part of the .NET Framework 4.0 and is also a part of trunk builds of Mono. The DLR can also be used as a library on older CLI implementations.

For further information, see [Wikipedia: IronPython](#) (the text above has been copied from this reference).

The original distribution of SharpDevelop includes IronPython (it is not included in the mp-Formula IDE). Ironpython can be downloaded from the [IronPython Homepage](#). Visual Studio integration is available through [Python Tools for Visual Studio](#).

---

```
#Load CLR
import clr

#Load the mpNumerics library
clr.AddReference('MatrixClass2')
from MatrixClass2 import mp, mpNum

#Set Floating point type to MPFR with 60 decimal digits precision
mp.FloatingPointType = 3;
mp.Prec10 = 60;

#Assign values to x1 and x2
x1 = mp.CNum("32.47");
x2 = mp.CNum("12.41");

#Calculate x3 = x1 / x2
x3 = x1 / x2;

#Print the value of x3
print "Result: ", x3.Str();
```

---

## B.7 MatLab (.NET interface)

MATLAB (matrix laboratory) is a numerical computing environment and fourth-generation programming language. Developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, Java, and Fortran.

Although MATLAB is intended primarily for numerical computing, an optional toolbox uses the MuPAD symbolic engine, allowing access to symbolic computing capabilities. An additional package, Simulink, adds graphical multi-domain simulation and Model-Based Design for dynamic and embedded systems.

The MATLAB application is built around the MATLAB language, and most use of MATLAB involves typing MATLAB code into the Command Window (as an interactive mathematical shell), or executing text files containing MATLAB codes, including scripts and/or functions.[6]

Variables are defined using the assignment operator, `=`. MATLAB is a weakly typed programming language because types are implicitly converted.[7] It is a dynamically typed language because variables can be assigned without declaring their type, except if they are to be treated as symbolic objects,[8] and that their type can change. Values can come from constants, from computation involving values of other variables, or from the output of a function.

As suggested by its name (a contraction of "Matrix Laboratory"), MATLAB can create and manipulate arrays of 1 (vectors), 2 (matrices), or more dimensions. In the MATLAB vernacular, a vector refers to a one dimensional ( $1 \times n$  or  $n \times 1$ ) matrix, commonly referred to as an array in other programming languages. A matrix generally refers to a 2-dimensional array, i.e. an  $m \times n$  array where  $m$  and  $n$  are greater than 1. Arrays with more than two dimensions are referred to as multidimensional arrays. Arrays are a fundamental type and many standard functions natively support array operations allowing work on arrays without explicit loops.

MATLAB can call functions and subroutines written in the C programming language or Fortran. A wrapper function is created allowing MATLAB data types to be passed and returned. The dynamically loadable object files created by compiling such functions are termed "MEX-files" (for MATLAB executable).[13][14]

Libraries written in Java, ActiveX or .NET can be directly called from MATLAB and many MATLAB libraries (for example XML or SQL support) are implemented as wrappers around Java or ActiveX libraries. Calling MATLAB from Java is more complicated, but can be done with a MATLAB extension,[15] which is sold separately by MathWorks, or using an undocumented mechanism called JMI (Java-to-MATLAB Interface),[16] which should not be confused with the unrelated Java Metadata Interface that is also called JMI.

As alternatives to the MuPAD based Symbolic Math Toolbox available from MathWorks, MATLAB can be connected to Maple or Mathematica.[17]

Libraries also exist to import and export MathML. MATLAB has a COM interface which is described in section ??.

For further information on MatLab, see [Wikipedia: MatLab](#) (the text above has been copied from this reference), or the [MatLab Homepage](#). See also [MatLab External Interfaces](#).

Example for using the library

---

```
x = 4.3;
fprintf('Start: x is equal to %6.2f.\n',x);

%Load mpFormulaPy .NET assembly
NET.addAssembly('MatrixClass2');

%Instantiate local reference to mpFormulaPy
mp = MatrixClass2.mp;

%Set Floating point type to MPFR with 60 decimal digits precision
NET.setStaticProperty('MatrixClass2.mp.FloatingPointType', 3);
NET.setStaticProperty('MatrixClass2.mp.Prec10', 50);
Myprec = mp.Prec10;

fprintf('End: Myprec is equal to %6.2f.\n',Myprec);

%Assign values to x1 and x2
x1 = mp.CNum(4.5);
x2 = mp.CNum('1.1');
x3 = x1 / x2;
x4 = MatrixClass2.mpNum;

x4 = mp.CNum(4.5356346345);
s = x3.Str();
s2 = char(s);
fprintf('s is equal to %s.\n',s2);
fprintf('End: x is equal to %6.2f.\n',x);
quit;
```

---

# Appendix C

## Building the library

Building the toolbox and the library from scratch is a much more involved process than just using them.

Conceptually, it could be described as a top-down process which starts at the level of the modification of the source files for the documentation, the following (automated) generation of various \*.xml, \*.cs, \*.h files and their processing with appropriate tools, which create the .NET, COM, native DLL and spreadsheet interfaces, ultimately leading to the connecting point of the mpNumC.h header file.

It could also be described as a bottom-up process, starting with the compilation of the \*.c, \*.h and \*.asm of the GMP, MPFR and FLINT library, followed by the compilation of the Eigen and Boost template libraries with the various supported data types, again leading to the connecting point of the mpNumC.h header file.

In practice, it is easiest to start any rebuilding of the toolbox or the library with an already working installation, with the following steps in mind:

- When changing a function, or introducing a new one, always start at the documentation, and provide all information which is required for automated generation of dependent files.
- Compile the documentation in latex, and process the output with makemenu etc.
- Run the routines which are necessary to update the .NET, COM, native DLL and spreadsheet interfaces.
- Decide whether you need to update the mpNumC.h header file.

Alternatively, you could start with a breaking change in one of the underlying libraries (e.g. GMP), recompile them first, then recompile all of the dependent libraries.

## C.1 Building the Library, Part 1

To compile the basic underlying libraries, MSYS2 and a recent version of GCC with pthread support are required.

### C.1.1 Downloading GCC and the MinGW-w64 toolchain

From <http://tdm-gcc.tdragon.net/download> download

`tdm64-gcc-4.9.2-3.exe`

and run the program.

In the start dialogue, select "Create", and in the following dialogue, select "MinGW-w64/TDM64 (32-bit and 64-bit)". Click "Next" to proceed to the licence terms and accept them by clicking "Next" again.

Choose an installation directory, in this example

`C:\TDM-GCC-64`

and confirm by clicking "Next". Select a download mirror that is geographically close to you, and click "Next" again. In the following panel, select the type of install (for our purposes, "TDM-GCC Recommended. C/C++" is fine), and click "Install".

### C.1.2 Downloading, installing and configuring MSYS2

MSYS2 is an updated, modern version of MSYS, both of which are Cygwin (POSIX compatibility layer) forks with the aim of better interoperability with native Windows software.

The name is a contraction of Minimal SYStem 2, and aims to provide support to facilitate using the bash shell, Autotools, revision control systems and the like for building native Windows applications using MinGW-w64 toolchains.

For additional information, see the project's website <http://msys2.github.io/>.

#### C.1.2.1 Downloading, installing and configuring MSYS2 32 bit

Download the latest version of MSYS2 32 bit, in this example the file

`msys2-i686-20150202.exe`

from <http://msys2.github.io/>.

or alternatively from <http://sourceforge.net/projects/msys2/files/Base/i686/>.

Start the program and wait until the installation has finished. Then confirm to start MSYS2. In the MSYS2 command prompt, type:

`pacman-key --init #Download keys`

Restart MSYS2, then type

`pacman -Syu #Update package database and full system upgrade`

Restart MSYS2, then type

`pacman -S diffutils git m4 make patch tar msys/openssl`

Copy the files from

C:\TDM-GCC-64

into

C:\msys32\mingw32

Restart MSYS2, then type

```
export PATH=/usr/local/bin:/usr/bin:/opt/bin:/mingw32/bin
```

Finally, in the directory

C:\Mingw32

execute (as administrator) the command file

autorebase.bat

### C.1.3 Downloading installing and configuring Code::Blocks

Code::Blocks is a free C, C++ and Fortran IDE. It is designed to be very extensible and fully configurable. It can be downloaded from

Codeblocks: [Codeblocks](#).

Unpack the file ExternalTools.exe in the mpFormula40 directory. Start the mpFormula Shell and select Compilers - Codeblocks. This will start Codeblocks 13.

In Codeblocks, choose the menu item Settings - Compiler... In the dialogue box, open the drop-down list Selected Compiler and navigate to GNU GCC Compiler 32 bit (at the very bottom of the list, you may need to scroll down).

Select the tab "Toolchain executables". In the text box "Compiler's installation directory", enter the absolute path to the directory containing the compiler, e.g.

C:\msys32\mingw32

Select the sub-tab "Program Files" and make sure that the entries are as follows:

```
C Compiler: x86_64-w64-mingw32-gcc.exe
C++ compiler: x86_64-w64-mingw32-g++.exe
Linker for dynamic libs: x86_64-w64-mingw32-g++.exe
Linker for static libs: ar.exe
Debugger: GDB/CDB debugger: Default
Resource compiler: windres.exe
Make program: mingw32-make.exe
```

### C.1.4 Downloading, compiling and installing GMP

The source code of GMP can be downloaded from GMP: <http://gmplib.org/>.

The license is the GNU Lesser General Public License, Version 3 (see appendix F.1.3)

The contributors are listed in section E.1.1

The mpformula library uses GMP version 6.0.0.

Compiling and installing needs to be done separately for 32 bit and 64 bit. The following detailed instructions are for 32 bit. For 64 bit, replace MP32 by MP64 and Win32 by Win64 in all file names and directory names, e.g. instead of `gmp_configure_Win32_Release.sh` for 32 bit use `gmp_configure_Win64_Release.sh` for 64 bit.

Within Windows Explorer, copy the directory `gmp-6.0.0` from the `ExternalLibraries` directory to the `ExternalTools\msys32\home\MP32` directory.

Within Windows Explorer, copy the file `gmp_configure_Win32_Release.sh` from the `mpFormulaC\Source\Configure\gmp` directory to the `ExternalTools\msys32\home\MP32\gmp-6.0.0` directory.

Start MSYS2. Within MSYS2, type

`cd ..`

to change to the home directory, and then type

`cd MP32/gmp-6.0.0`

to change to the

`/home/MP32/gmp-6.0.0`

directory. Within MSYS2, type

`ls *.sh`

to list only the shell scripts. Within MSYS2, type

`bash gmp_configure_Win32_Release.sh`

to start the shell script which will configure the GMP source files for compilation of a 32 bit static library. This will take some time to complete. Once this has been completed (i.e. the command prompt has returned), type

`make`

to start the actual compilation. Again, this will take some time to complete. Once this has been done (i.e. the command prompt has returned), type

`make check`

to start the compilation and execution of a number of test programs. All tests should pass. Only if all of these steps have been completed successfully, you should finally type

`make install`

to install the compiled files into the folder

`C:\Extra\mpFormula40\ExternalTools1\msys\local\Win32`

### C.1.5 Downloading, compiling and installing MPFR

The source code of MPFR can be downloaded from MPFR: <http://www.mpfr.org/>.

The license is the GNU Lesser General Public License, Version 3 (see appendix F.1.3)

The contributors are listed in section E.1.2

The mpformula library uses MPFR version 3.1.2.

Compiling and installing needs to be done separately for 32 bit and 64 bit. The following detailed instructions are for 32 bit. For 64 bit, replace MP32 by MP64 and Win32 by Win64 in all file names and directory names, e.g. instead of `mpfr_configure_Win32_Release.sh` for 32 bit use `mpfr_configure_Win64_Release.sh` for 64 bit.

Within Windows Explorer, copy the directory `mpfr-3.1.2` from the `ExternalLibraries` directory to the `ExternalTools\msys32\home\MP32` directory.

Within Windows Explorer, copy the file `mpfr_configure_Win32_Release.sh` from the `mpFormulaC\Source\Configure\mpfr` directory to the `ExternalTools\msys32\home\MP32\mpfr-3.1.2` directory.

Start MSYS2. Within MSYS2, type

`cd ..`

to change to the home directory, and then type

`cd MP32/mpfr-3.1.2`

to change to the

`/home/MP32/mpfr-3.1.2`

directory. Within MSYS2, type

`ls *.sh`

to list only the shell scripts. Within MSYS2, type

`bash mpfr_configure_Win32_Release.sh`

to start the shell script which will configure the MPFR source files for compilation of a 32 bit static library. This will take some time to complete. Once this has been completed (i.e. the command prompt has returned), type

`make`

to start the actual compilation. Again, this will take some time to complete. Once this has been done (i.e. the command prompt has returned), type

`make check`

to start the compilation and execution of a number of test programs. All tests should pass. Only if all of these steps have been completed successfully, you should finally type

`make install`

to install the compiled files into the folder

`C:\Extra\mpFormula40\ExternalTools1\msys\local\Win32`

### C.1.6 Downloading, compiling and installing FLINT

FLINT: <http://www.flintlib.org/> .

The license is the GNU General Public License, Version 2 or later (see appendix F.1.1 and F.1.4)

The contributors are listed in section E.1.3

The mpFormulaC library uses FLINT version 2.4.5.

Compiling and installing needs to be done separately for 32 bit and 64 bit. The following detailed instructions are for 32 bit. For 64 bit, replace MP32 by MP64 and Win32 by Win64 in all file names and directory names, e.g. instead of `flint_configure_Win32_Release.sh` for 32 bit use `flint_configure_Win64_Release.sh` for 64 bit.

Within Windows Explorer, copy the directory `flint-2.4.5` from the `ExternalLibraries` directory to the `ExternalTools\msys32\home\MP32` directory.

Within Windows Explorer, copy the file `flint_configure_Win32_Release.sh` from the  
`..\mpFormulaC\Source\Configure\flint` directory to the  
`..\ExternalTools\msys32\home\MP32\flint-2.4.5` directory.

Start MSYS2. Within MSYS2, type

`cd ..`

to change to the home directory, and then type

`cd MP32/flint-2.4.5`

to change to the

`/home/MP32/flint-2.4.5`

directory. Within MSYS2, type

`ls *.sh`

to list only the shell scripts. Within MSYS2, type

`bash flint_configure_Win32_Release.sh`

to start the shell script which will configure the FLINT source files for compilation of a 32 bit static library. This will take some time to complete. Once this has been completed (i.e. the command prompt has returned), type

`make`

to start the actual compilation. Again, this will take some time to complete. Once this has been done (i.e. the command prompt has returned), type

`make check`

to start the compilation and execution of a number of test programs. All tests should pass. Only if all of these steps have been completed successfully, you should finally type

`make install`

to install the compiled files into the folder

`..\ExternalTools\msys\local\Win32`

### C.1.6.1 Remaining issues with the 64 bit build

For FLINT 32 bit compilation works without errors, and all tests pass.

For FLINT 64 bit, compilation works without errors or warnings; however, in the original distribution, five test programs crash. To run all test programs, copy the files `t-clog.c`, `t-clog_ui.c`, `t-flog.c`, `t-flog_ui.c` from the folder

```
..\mpFormulaC\Source\Configure\flint\FailedTests64bit_Dummy\fmpz\test
```

into the folder

```
..\mpFormulaC\ExternalTools\msys\home\MP\flint-2.4.5\fmpz\test
```

Also, copy the file `t-factor_zassenhaus.c` from the folder

```
..\mpFormulaC\Source\Configure\flint\FailedTests64bit_Dummy\fmpz_poly_factor\test
```

into the folder

```
..\mpFormulaC\ExternalTools\msys\home\MP\flint-2.4.5\fmpz_poly_factor\test
```

and only then run make check.

### C.1.7 Downloading, compiling and installing ARB

ARB: <http://fredrikj.net/arb/#> .

The actual download site is

<https://github.com/fredrik-johansson/arb/releases> .

Additional information is available from the blog of the author of Arb: <http://fredrikj.net/blog/> .

From the same author a FLINT/ARB Wrapper in Python is available: <http://fredrikj.net/python-flint/> .

Also from the same author a related Python library is available:

<http://mpmath.org/> .

The license is the GNU General Public License, Version 2 or later (see appendix [F.1.1](#) and [F.1.4](#))

The contributors are listed in section [E.1.4](#)

The mpformula library uses ARB version 2.5.0.

Compiling and installing needs to be done separately for 32 bit and 64 bit. The following detailed instructions are for 32 bit. For 64 bit, replace MP32 by MP64 and Win32 by Win64 in all file names and directory names, e.g. instead of `arb_configure_Win32_Release.sh` for 32 bit use `arb_configure_Win64_Release.sh` for 64 bit.

Within Windows Explorer, copy the directory `arb-2.5.0` from the `ExternalLibraries` directory to the `ExternalTools\msys32\home\MP32` directory.

Within Windows Explorer, copy the file `arb_configure_Win32_Release.sh` from the `mpFormulaC\Source\Configure\arb` directory to the `ExternalTools\msys32\home\MP32\arb-2.5.0` directory.

Start MSYS2. Within MSYS2, type

`cd ..`

to change to the home directory, and then type

`cd MP32/arb-2.5.0`

to change to the

`/home/MP32/arb-2.5.0`

directory. Within MSYS2, type

`ls *.sh`

to list only the shell scripts. Within MSYS2, type

`bash arb_configure_Win32_Release.sh`

to start the shell script which will configure the ARB source files for compilation of a 32 bit static library. This will take some time to complete. Once this has been completed (i.e. the command prompt has returned), type

`make`

to start the actual compilation. Again, this will take some time to complete. Once this has been done (i.e. the command prompt has returned), type

```
make check
```

to start the compilation and execution of a number of test programs. All tests should pass. Only if all of these steps have been completed successfully, you should finally type

```
make install
```

to install the compiled files into the folder

```
..\ExternalTools1\msys\local\Win32
```

#### C.1.7.1 Remaining issues with the 32 bit build

For ARB 32 bit, compilation works without errors or warnings, and all tests but two pass. To run all test programs, copy the file `t-epsilon_arg.c` from the folder

```
..\mpFormulaC\Source\Configure\arb\FailedTests32bit_Dummy\acb_modular\test
```

into the folder

```
..\ExternalTools\msys\home\MP32\arb-2.5.0\acb_modular\test
```

and also copy the file `t-set_str.c` from the folder

```
..\mpFormulaC\Source\Configure\arb\FailedTests32bit_Dummy\arb\test
```

into the folder

```
..\ExternalTools\msys\home\MP32\arb-2.5.0\arb\test
```

and only then run `make check`.

For ARB 64 bit, compilation works without errors but a number of warnings, and currently all tests crash (reason unknown).

## C.2 Building the Library, Part 2

### C.2.1 Boost Math

Boost Math: [http://www.boost.org/doc/libs/1\\_57\\_0/libs/math/doc/html/index.html](http://www.boost.org/doc/libs/1_57_0/libs/math/doc/html/index.html) .

The roadmap for the development version is available at

[http://www.boost.org/doc/libs/master/libs/math/doc/html/math\\_toolkit/history2.html](http://www.boost.org/doc/libs/master/libs/math/doc/html/math_toolkit/history2.html) .

<http://www.boost.org/doc/libs/master/libs/math/doc/html/index.html> .

The license is the Boost Software License, Version 1.0(see appendix F.2.2)

The contributors are listed in section E.1.8

The mpformula library uses Boost Math version 1.57.0.

### C.2.2 Boost Random

Boost Random: [http://www.boost.org/doc/libs/1\\_57\\_0/doc/html/boostrandom.html](http://www.boost.org/doc/libs/1_57_0/doc/html/boostrandom.html) .

The license is the Boost Software License, Version 1.0(see appendix F.2.2)

The contributors are listed in section E.1.9

The mpformula library uses Boost Random version 1.57.0.

### C.2.3 Eigen

Eigen: <http://eigen.tuxfamily.org/index.php?title=MainPage> .

The license is the Mozilla Public License, Version 2 (see appendix F.2.1)

The contributors are listed in section E.1.6

The mpformula library uses Eigen version 3.2.2.

### C.2.4 Source Code derived from other libraries

#### C.2.4.1 MPFRC++

MPFRC++: <http://www.holoborodko.com/pavel/mpfr/> . LGPL version 2.1 ?

The current license on the MPFRC++ web site is the GNU General Public License, Version 3 (see appendix F.1.4)

The MPFRC++ has not used an official versioning system from early on. The version used by the mpformula library is a version optimized for use with the Eigen library, which is part of the Eigen distribution (in the folder \unsupported\test\mpreal, dated 26Feb2014), and which does not have a version number. The license of this version is the GNU Library General Public License, Version 2.1 (see appendix F.1.2)

The contributors are listed in section E.1.5

### C.3 Building the documentation

The following software was used to build the documentation:

miktex 2.9.4813 : [miktex](#).

texniccenter 2.02: [texniccenter](#).

After installing these programs one additional step is needed:

Build → Select Output Profile: Latex ⇒ PDF

Build → Define Output Profile: Latex ⇒ PDF. In this dialogue box, select the tab "Postprocessor"

In the Listbox "Processors", add an item and name it "Nomenclature"

In the Textfield "Executable:", enter the full path to "miktex-makeindex.exe" ,like

```
C:\Program Files\MiKTeX 2.9\miktex\bin\x64\miktex-makeindex.exe
```

In the Textfield "Arguments:", enter the following:

```
-s nomencl.ist "%tm.nlo" -o "%tm.nls"
```

## C.4 Additional libraries

The source code of AMath and DAMath can be downloaded from:

<http://www.wolfgang-ehrhardt.de> .

You need to download 2 separate files to support compilation of both 32 bit and 64 bit dlls:

- amath\_2014-05-11.zip contains the AMath package
- damath\_2014-05-11.zip contains the DAMath package

### C.4.1 MPIR

MPFR: <http://www.mpir.org> .

The license is the GNU Lesser General Public License, Version 3 (see appendix F.1.3)

The contributors are listed in section E.1.1

The mpformula library uses MPFR version 2.6.0.

### C.4.2 gmpfrxx

gmpfrxx: <http://math.berkeley.edu/~wilken/code/gmpfrxx/> .

The license is the GNU Lesser General Public License, Version 3 (see appendix F.1.3)

## C.5 Working Notes

## C.6 Where to find VB Code

Most new code is loaded when starting the current version of Stats32.xls

The code itself is then dynamically loaded into Source.xls.

Within Source.xls, most new noncentral code is in Approx.bas.

This includes Doubly noncentral t and F.

New Code for Pearson Rho is in ALL\_DistribN (at the end).

LambdaDemo is in ALL\_DistribX.

The finite series for cdis, tdis and fdis are in tdisn\_Owen.

## C.7 How to run Permutation Code

Within Stats32.xls, goto Interface.bas, and within this module to DemoTabularProcs

For additional fine tuning: Within Source.xls, goto XLInterface.bas, and within this module to TabularProcs.

# Appendix D

## Roadmap

## D.1 CPython

CPython, the reference implementation of Python, is free and open source software and has a community-based development model, as do nearly all of its alternative implementations. CPython is managed by the non-profit Python Software Foundation.

For further information on Python, see [Wikipedia: Python](#) (the text above has been copied from this reference), or the [Python Homepage](#). Support for COM is included in the distribution of the [ActivePython Community Edition](#).

Python can use GMP und MPFR thanks to [GMPY2](#), with documentation [here](#).

IPython is an integrations platform for various scientific libraries (NumPy, SciPy, matlibplot, pandas etc.) <http://ipython.org/>. Popular distributions are the Community Edition of Anaconda: <http://docs.continuum.io/anaconda/index.html>,

Book recommendation: [McKinney \(2012\)](#).

To compile the mpmath library libraries, Python 2.7 is required.

### D.1.1 Downloading and installing CPython 2.7

ActivePython is ActiveState's complete and ready-to-install distribution of Python. It provides a one-step installation of all essential Python modules, as well as extensive documentation. The Windows distribution ships with PyWin32 – a suite of Windows tools developed by Mark Hammond, including bindings to the Win32 API and Windows COM. ActivePython can be downloaded from

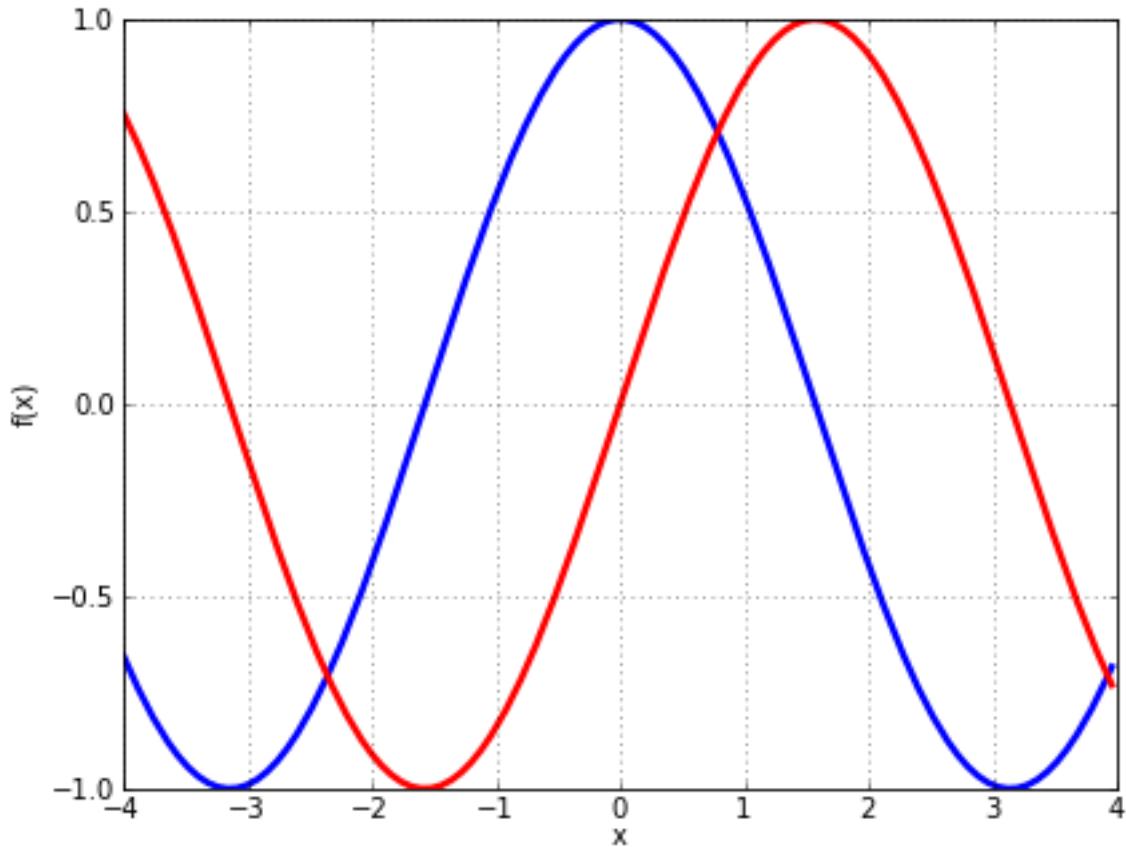
<http://www.activestate.com/activepython/downloads>.

The latest release version of the 2.7x series is 2.7.6.9. You need to download 2 separate files to support compilation of both 32 bit and 64 bit dlls.

## D.1.2 Plotting

If matplotlib is available, the functions `plot` and `cplot` in mpmath can be used to plot functions respectively as x-y graphs and in the complex plane. Also, `splot` can be used to produce 3D surface plots.

### D.1.2.1 Function curve plots



**Figure D.1:** MpMath 2Dplot

Output of `plot([cos, sin], [-4, 4])`

```
mpmath.plot(ctx, f, xlim=[-5, 5], ylim=None, points=200, file=None, dpi=None, singularities=[], axes=None)
```

Shows a simple 2D plot of a function  $f(x)$  or list of functions  $[f_0(x), f_1(x), \dots, f_n(x)]$  over a given interval specified by `xlim`. Some examples:

---

```
plot(lambda x: exp(x)*li(x), [1, 4])
plot([cos, sin], [-4, 4])
plot([fresnels, fresnelc], [-4, 4])
plot([sqrt, cbrt], [-4, 4])
plot(lambda t: zeta(0.5+t*j), [-20, 20])
plot([floor, ceil, abs, sign], [-5, 5])
```

---

Points where the function raises a numerical exception or returns an infinite value are removed from the graph. Singularities can also be excluded explicitly as follows (useful for removing erroneous vertical lines):

---

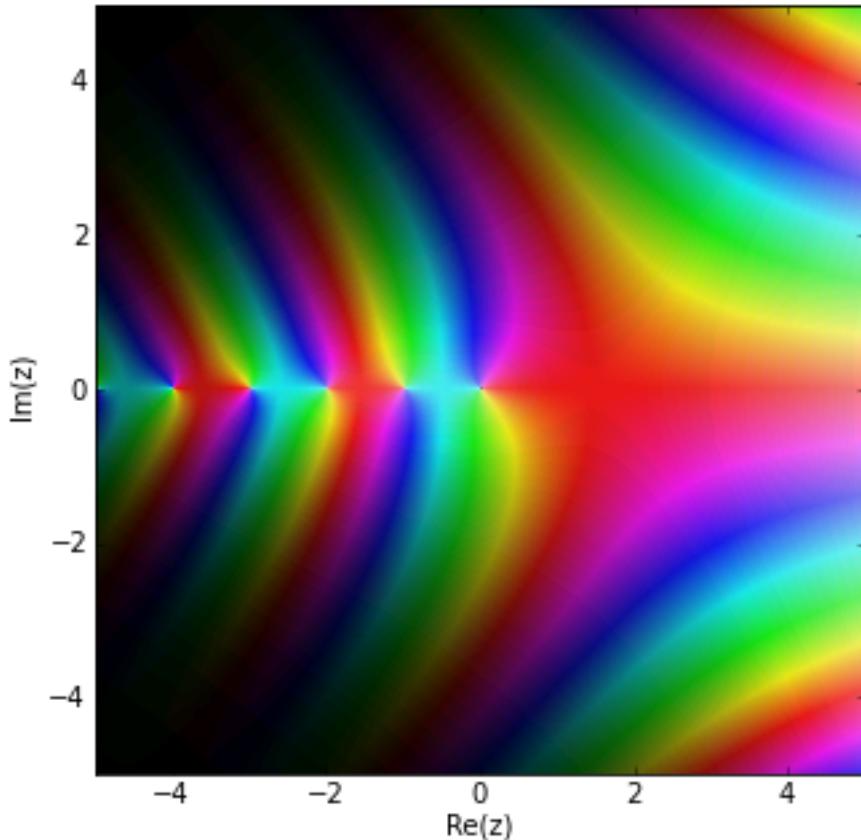
```
plot(cot, ylim=[-5, 5]) # bad
plot(cot, ylim=[-5, 5], singularities=[-pi, 0, pi]) # good
```

---

For parts where the function assumes complex values, the real part is plotted with dashes and the imaginary part is plotted with dots.

Note: This function requires matplotlib (pylab).

### D.1.2.2 Complex function plots



**Figure D.2:** MpMath cplot

Output of `fp.cplot(fp.gamma, points=100000)`

`mpmath.cplot(ctx, f, re=[-5, 5], im=[-5, 5], points=2000, color=None, verbose=False, file=None, dpi=None, axes=None)`

Plots the given complex-valued function  $f$  over a rectangular part of the complex plane specified by the pairs of intervals  $re$  and  $im$ . For example:

---

```
cplot(lambda z: z, [-2, 2], [-10, 10])
cplot(exp)
cplot(zeta, [0, 1], [0, 50])
```

---

By default, the complex argument (phase) is shown as color (hue) and the magnitude is shown as brightness. You can also supply a custom color function (color). This function should take a complex number as input and return an RGB 3-tuple containing floats in the range 0.0-1.0.

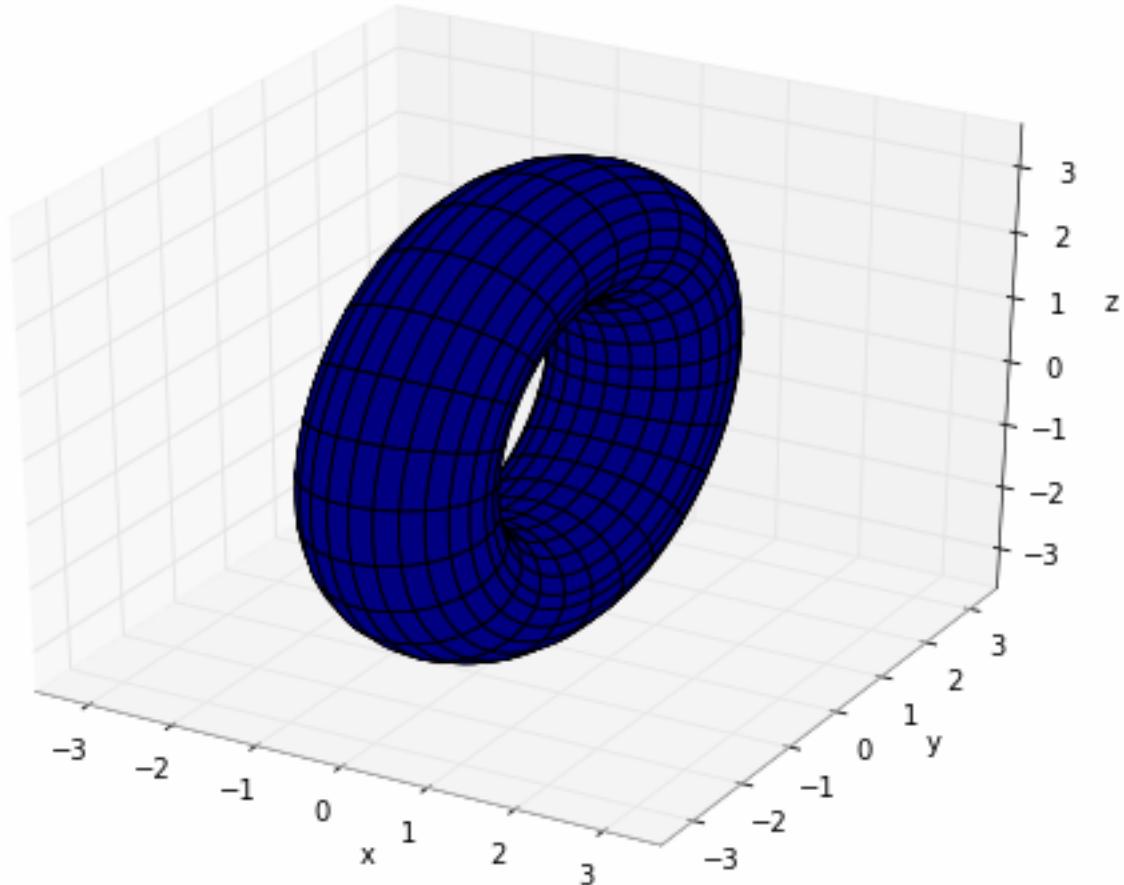
To obtain a sharp image, the number of points may need to be increased to 100,000 or thereabout. Since evaluating the function that many times is likely to be slow, the 'verbose' option is useful to display progress.

Note: This function requires matplotlib (pylab).

### D.1.2.3 3D surface plots

Output of splot for the donut example.

```
mpmath.splot(ctx, f, u=[-5, 5], v=[-5, 5], points=100, keep_aspect=True, wireframe=False, file=None, dpi=None, axes=None)
```



**Figure D.3:** MpMath Surface plot

Plots the surface defined by  $f$ .

If  $f$  returns a single component, then this plots the surface defined by  $z = f(x, y)$  over the rectangular domain with  $x = u$  and  $y = v$ .

If  $f$  returns three components, then this plots the parametric surface  $x, y, z = f(u, v)$  over the pairs of intervals  $u$  and  $v$ .

For example, to plot a simple function:

---

```
>>> from mpmath import *
>>> f = lambda x, y: sin(x+y)*cos(y)
>>> splot(f, [-pi,pi], [-pi,pi])
```

---

Plotting a donut:

---

```
>>> r, R = 1, 2.5
>>> f = lambda u, v: [r*cos(u), (R+r*sin(u))*cos(v), (R+r*sin(u))*sin(v)]
```

---

---

```
>>> splot(f, [0, 2*pi], [0, 2*pi])
```

Note: This function requires matplotlib (pylab) 0.98.5.3 or higher.

### D.1.3 Using the C-API

This section describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can not only define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see [The Python Tutorial](#). The [Python Language Reference](#) gives a more formal definition of the language. The [Python Standard Library](#) documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate [Python/C API Reference Manual](#).

#### . Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such extension modules can do two things that can't be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header "Python.h". The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

Note: The C extension interface is specific to CPython, and extension modules do not work on other Python implementations. In many cases, it is possible to avoid writing C extensions and preserve portability to other implementations. For example, if your use case is calling C library functions or system calls, you should consider using the `ctypes` module or the `cffi` library rather than writing custom C code. These modules let you write Python code to interface with C code and are more portable between implementations of Python than writing and compiling a C extension module.

As an example for an extension module which provides additional functionality in multi-precision computing, see the documentation on `gmpy2` (section [??](#)).

### D.1.4 Interfaces to the C family of languages

#### D.1.4.1 Windows, GNU/Linux, Mac OSX: GNU Compiler Collection

The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

Originally named the GNU C Compiler, because it only handled the C programming language, GCC 1.0 was released in 1987 and the compiler was extended to compile C++ in December of

that year.[1] Front ends were later developed for Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others.[3]

As well as being the official compiler of the unfinished GNU operating system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including Linux and the BSD family. A port to RISC OS has also been developed extensively in recent years. There is also an old (3.0) port of GCC to Plan9, running under its ANSI/POSIX Environment (APE).[4] GCC is also available for Microsoft Windows operating systems and for the ARM processor used by many portable devices.

For further information on the GNU Compiler Collection, see [Wikipedia: GCC](#) (the text above has been copied from this reference), or the [GCC Homepage](#).

#### D.1.4.2 Windows: MSVC

Microsoft Visual C++ (often abbreviated as MSVC or VC++) is a commercial (free version available), integrated development environment (IDE) product from Microsoft for the C, C++, and C++/CLI programming languages. It features tools for developing and debugging C++ code, especially code written for the Microsoft Windows API, the DirectX API, and the Microsoft .NET Framework.

Although the product originated as an IDE for the C programming language, the compiler's support for that language conforms only to the original edition of the C standard, dating from 1989. The later revisions of the standard, C99 and C11, are not supported.[41] According to Herb Sutter, the C compiler is only included for "historical reasons" and is not planned to be further developed. Users are advised to either use only the subset of the C language that is also valid C++, and then use the C++ compiler to compile their code, or to just use a different compiler such as Intel C++ Compiler or the GNU Compiler Collection instead.[42]

For further information on Microsoft Visual C++, see [Wikipedia: MSVC](#) (the text above has been copied from this reference), or the [MSVC Homepage](#).

The following C file makes a number of direct calls into Python, passing arguments as strings and receiving a result as string:

---

```
#include "stdafx.h"
#include "CallPython.h"

int main(int argc, const char *argv[])
{
long ResultLong = SetSpecialValue_Long(2,3,4);
printf( "This is a long: %d\n", ResultLong);

double ResultDouble = SetSpecialValue_Double(2.0,3.0,4.0);
printf( "This is a double: %f\n", ResultDouble);

const char *sLong2[] = {"TestLong", "111", "3", "1432"};
MyPythonFunction(4, sLong2);

const char *sDouble2[] = {"TestDouble", "fff", "13.5", "265.34"};
MyPythonFunction(4, sDouble2);
```

```

const char *sString2[] = {"TestStringFunc", "sss", "3", "2"};
MyPythonFunction(4, sString2);

const char *sString3[] = {"TestStringMpf2", "3", "2.456"};
MyPythonFunctionString2(3, sString3);

char buffer[1600]; // 1600 bytes allocated here on the stack.
int size0a = MyPythonFunctionStringReturn(3, sString3, buffer, sizeof(buffer));
printf("New New0a %s\n", buffer); // prints "Mar"
//printf("Length of string: %ld\n", size0a);

char buffer0[1600]; // 1600 bytes allocated here on the stack.
int size0 = MyPythonFunctionStringReturn00("TestStringMpf0", buffer0,
    sizeof(buffer0));
printf("New New0 %s\n", buffer0); // prints "Mar"
//printf("Length of string0: %ld\n", size0);

char buffer1[1600]; // 1600 bytes allocated here on the stack.
int size1 = MyPythonFunctionStringReturn01("TestStringMpf1", "3", buffer1,
    sizeof(buffer1));
printf("New New1 %s\n", buffer1); // prints "Mar"
//printf("Length of string: %ld\n", size1);

char buffer2[1600]; // 1600 bytes allocated here on the stack.
int size2 = MyPythonFunctionStringReturn02("TestStringMpf2", "3", "2.456", buffer2,
    sizeof(buffer2));
printf("New New2 %s\n", buffer2); // prints "Mar"
//printf("Length of string: %ld\n", size2);

ClosePy();
return 0;
}

```

The header file CallPython.h looks like this:

---

```

#pragma warning(disable: 4244)

#ifndef CALLPYTHON_EXPORTS
#define MPNUMC_DLL_IMPORTEXPORT __declspec(dllexport)
#else
#define MPNUMC_DLL_IMPORTEXPORT __declspec(dllimport)
#endif

#ifndef __cplusplus
extern "C" {
#endif

MPNUMC_DLL_IMPORTEXPORT long SetSpecialValue_Long(long m, long n, long what);
MPNUMC_DLL_IMPORTEXPORT double SetSpecialValue_Double(double m, double n, double
    what);
MPNUMC_DLL_IMPORTEXPORT int CallPythonFunction(int argc, const char *argv[]);

```

```

MPNUMC_DLL_IMPORTEXPORT int MyPythonFunction(int argc, const char *argv[]);
MPNUMC_DLL_IMPORTEXPORT int MyPythonFunctionString(int argc, const char *argv[]);
MPNUMC_DLL_IMPORTEXPORT int MyPythonFunctionString2(int argc, const char *argv[]);
MPNUMC_DLL_IMPORTEXPORT int MyPythonFunctionStringReturn(int argc, const char
    *argv[], char* buffer, int buffersize);

MPNUMC_DLL_IMPORTEXPORT int MyPythonFunctionStringReturn00(const char* FuncName,
    char* buffer, int buffersize);
MPNUMC_DLL_IMPORTEXPORT int MyPythonFunctionStringReturn01(const char* FuncName,
    const char* Arg01, char* buffer, int buffersize);
MPNUMC_DLL_IMPORTEXPORT int MyPythonFunctionStringReturn02(const char* FuncName,
    const char* Arg01, const char* Arg02, char* buffer, int buffersize);

MPNUMC_DLL_IMPORTEXPORT void ClosePy();
#endif __cplusplus
}
#endif

```

---

The C file CallPython.cpp (which produces the dynamic link library) looks like this:

---

```

#define _CRT_SECURE_NO_WARNINGS
#include "CallPython.h"
#include "stdafx.h"
#include<stdio.h>
#include <Python.h>

long SetSpecialValue_Long(long m, long n, long what)
{
    return (m + n + 1) * what;
}

double SetSpecialValue_Double(double m, double n, double what)
{
    return (m + n) * what;
}

int MyPythonFunctionStringReturn00(const char* FuncName, char* buffer, int buffersize)
{
    PyObject *pFunc;
    PyObject *pValue;
    Py_ssize_t size;
    PyObject *pModule= GetPythonModule2();
    pFunc = PyObject_GetAttrString(pModule, FuncName);
    if (pFunc && PyCallable_Check(pFunc)) {
        pValue = PyObject_CallObject(pFunc, NULL);
        if (pValue != NULL) {
            strncpy(buffer, PyUnicode_AsUTF8AndSize(pValue, &size), buffersize-1);
            Py_DECREF(pValue);
        }
    }
}

```

```
Py_XDECREF(pFunc);
return size;
}

int MyPythonFunctionStringReturn01(const char* FuncName, const char* Arg01, char*
buffer, int buffersize)
{
PyObject *pFunc, *pArgs, *pValue;
PyObject *pModule= GetPythonModule2();
Py_ssize_t size;
pFunc = PyObject_GetAttrString(pModule, FuncName);
if (pFunc && PyCallable_Check(pFunc)) {
pArgs = PyTuple_New(1);
pValue = PyUnicode_FromString(Arg01);
PyTuple_SetItem(pArgs, 0, pValue);

pValue = PyObject_CallObject(pFunc, pArgs);
Py_DECREF(pArgs);
if (pValue != NULL) {
strncpy(buffer, PyUnicode_AsUTF8AndSize(pValue, &size), buffersize-1);
Py_DECREF(pValue);
}
}
Py_XDECREF(pFunc);
return size;
}

int MyPythonFunctionStringReturn02(const char* FuncName, const char* Arg01, const
char* Arg02, char* buffer, int buffersize)
{
PyObject *pFunc, *pArgs, *pValue;
PyObject *pModule= GetPythonModule2();
Py_ssize_t size;

pFunc = PyObject_GetAttrString(pModule, FuncName);

if (pFunc && PyCallable_Check(pFunc)) {
pArgs = PyTuple_New(2);
pValue = PyUnicode_FromString(Arg01);
PyTuple_SetItem(pArgs, 0, pValue);

pValue = PyUnicode_FromString(Arg02);
PyTuple_SetItem(pArgs, 1, pValue);

pValue = PyObject_CallObject(pFunc, pArgs);
Py_DECREF(pArgs);
if (pValue != NULL) {
strncpy(buffer, PyUnicode_AsUTF8AndSize(pValue, &size), buffersize-1);
Py_DECREF(pValue);
}
}
```

```
}

Py_XDECREF(pFunc);
return size;
}

void ClosePy()
{
PyObject *pModule;
pModule = GetPythonModule2();
Py_DECREF(pModule);
Py_Finalize();
}
```

---

### D.1.5 Cython: C extensions for the Python language

[Cython] is a programming language that makes writing C extensions for the Python language as easy as Python itself. It aims to become a superset of the [Python] language which gives it high-level, object-oriented, functional, and dynamic programming. Its main feature on top of these is support for optional static type declarations as part of the language. The source code gets translated into optimized C/C++ code and compiled as Python extension modules. This allows for both very fast program execution and tight integration with external C libraries, while keeping up the high programmer productivity for which the Python language is well known.

The primary Python execution environment is commonly referred to as CPython, as it is written in C. Other major implementations use Java (Jython [Jython]), C# (IronPython [IronPython]) and Python itself (PyPy [PyPy]). Written in C, CPython has been conducive to wrapping many external libraries that interface through the C language. It has, however, remained non trivial to write the necessary glue code in C, especially for programmers who are more fluent in a high-level language like Python than in a close-to-the-metal language like C.

Originally based on the well-known Pyrex [Pyrex], the Cython project has approached this problem by means of a source code compiler that translates Python code to equivalent C code. This code is executed within the CPython runtime environment, but at the speed of compiled C and with the ability to call directly into C libraries. At the same time, it keeps the original interface of the Python source code, which makes it directly usable from Python code. These two-fold characteristics enable CythonâŽs two major use cases: extending the CPython interpreter with fast binary modules, and interfacing Python code with external C libraries.

While Cython can compile (most) regular Python code, the generated C code usually gains major (and sometime impressive) speed improvements from optional static type declarations for both Python and C types. These allow Cython to assign C semantics to parts of the code, and to translate them into very efficient C code. Type declarations can therefore be used for two purposes: for moving code sections from dynamic Python semantics into static-and-fast C semantics, but also for directly manipulating types defined in external libraries. Cython thus merges the two worlds into a very broadly applicable programming language.

### D.1.6 A Windows-specific interface: using COM

Example for using the library

---

```
#Enable COM support
from win32com.client import Dispatch

#Load the mpNumerics library
mp = Dispatch("mpNumerics.mp_Lib")

#Set Floating point type to MPFR with 60 decimal digits precision
mp.FloatingPointType = 3
mp.Prec10 = 60

#Assign values to x1 and x2
x1 = mp.Real(4.5)
x2 = mp.Real(1.21)

#Calculate x3 = x1 / x2
x3 = x1.Div(x2)
```

```
#Print the value of x3
print (x3.Str())
```

---

Example for using Excel

---

```
#Enable COM support
from win32com.client import Dispatch

#Load the Excel library
xl = Dispatch("Excel.Application")
xl.Visible = 1
xl.Workbooks.Add()
xl.Cells(1,1).Value = "Hello442"
print("From Python")
```

---

## D.2 R (Statistical System)

R is a free software programming language and a software environment for statistical computing and graphics. The R language is widely used among statisticians and data miners for developing statistical software[2][3] and data analysis.[3] Polls and surveys of data miners are showing R's popularity has increased substantially in recent years.[4][5][6]

R is an implementation of the S programming language combined with lexical scoping semantics inspired by Scheme. S was created by John Chambers while at Bell Labs. R was created by Ross Ihaka and Robert Gentleman[7] at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team, of which Chambers is a member. R is named partly after the first names of the first two R authors and partly as a play on the name of S.[8]

R is a GNU project.[9][10] The source code for the R software environment is written primarily in C, Fortran, and R.[11] R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems. R uses a command line interface; however, several graphical user interfaces are available for use with R.

R provides a wide variety of statistical and graphical techniques, including linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, and others. R is easily extensible through functions and extensions, and the R community is noted for its active contributions in terms of packages. There are some important differences, but much code written for S runs unaltered. Many of R's standard functions are written in R itself, which makes it easy for users to follow the algorithmic choices made. For computationally intensive tasks, C, C++, and Fortran code can be linked and called at run time. Advanced users can write C, C++[12] or Java[13] code to manipulate R objects directly.

R is highly extensible through the use of user-submitted packages for specific functions or specific areas of study. Due to its S heritage, R has stronger object-oriented programming facilities than most statistical computing languages. Extending R is also eased by its lexical scoping rules.[14]

Another strength of R is static graphics, which can produce publication-quality graphs, including mathematical symbols. Dynamic and interactive graphics are available through additional packages.[15]

R has its own LaTeX-like documentation format, which is used to supply comprehensive documentation, both on-line in a number of formats and in hard copy.

R is an interpreted language; users typically access it through a command-line interpreter. If a user types "2+2" at the R command prompt and presses enter, the computer replies with "4", as shown below:

```
i 2+2
[1] 4
```

Like many other languages, R supports matrix arithmetic. R's data structures include scalars, vectors, matrices, data frames (similar to tables in a relational database) and lists.[16] R's extensible object-system includes objects for (among others): regression models, time-series and geo-spatial coordinates.

R supports procedural programming with functions and, for some functions, object-oriented programming with generic functions. A generic function acts differently depending on the type of arguments passed to it. In other words, the generic function dispatches the function (method)

specific to that type of object. For example, R has a generic `print()` function that can print almost every type of object in R with a simple `"print(objectname)"` syntax.

Although mostly used by statisticians and other practitioners requiring an environment for statistical computation and software development, R can also operate as a general matrix calculation toolbox - with performance benchmarks comparable to GNU Octave or MATLAB.[17]

For further information on R, see [Wikipedia: R](#) (the text above has been copied from this reference), or the [R Homepage](#).

A popular GUI for R is [Rstudio](#).

Within RStudio:

Tools - Install Packages.

Type RD in the dialogue box.

RDCOMClient should appear in the drop down box.

Select RDCOMClient and click on Install.

Needs to be done separately for 32 bit and 64 bit.

R contains packages which provide interfaces to GMP (<http://mulcyber.toulouse.inra.fr/projects/gmp>) and MPFR (<http://rmpfr.r-forge.r-project.org/>).

Book recommendation: [Adler \(2012\)](#).

Book recommendation: [Verzani \(2011\)](#).

Book recommendation: [Chang \(2012\)](#).

Example for using the library

---

```
#Enable COM support
require("RDCOMClient")

#Load the mpNumerics library
mp = COMCreate("mpNumerics.mp_Lib")

#Set Floating point type to MPFR with 60 decimal digits precision
mp[["Prec10"]] = 160
mp[["FloatingPointType"]] = 3

#Assign values to x1 and x2
x1 = mp$Real("4.5")
x2 = mp$Real("1.1")

#Perform arithmetic operations
x3 = x1$Plus(x2)
x4 = x1$Div(x2)

#Display output
mp[["Prec10"]]
x3$Str()
x4$Str()
```

---

Example for using Excel

---

```

#Load Library
require("rcom")

#Create instance of Excel
xlApp = comCreateObject("Excel.Application")

#Add 1 workbook and make it visible
wb = xlApp[["Workbooks"]]$Add()
xlApp[["Visible"]] = TRUE

#Display the name of the 1st worksheet
ws = wb[["Worksheets", 1]]
wsname = ws[["Name"]]
wsname

#Assign values to a range
mrange = ws[["Range", "A1:B10"]]
mrange[["Value"]] = 10.3

#Display the values of the range
d = mrange[["Value"]]
d
$
```

---

Within RStudio, on the menu bar, click Tools -> Install packages.  
 In the textbox "Packages" enter "RMpfr" and click on "Install".  
 This should install the RMpfr and GMP packages.

---

```

#Load Library
require("Rmpfr")

n1.25 <- mpfr(5, precBits = 256)/4
n1.25

n1.25 ^ c(1:7, 20, 30)

exp(n1.25)

getGroupMembers("Math")

showMethods(classes = "mpfr")

showMethods(classes = "mpfrArray")
```

---

# Appendix E

## Acknowledgements

### E.1 Contributors to libraries used in the numerical routines

#### E.1.1 Contributors to GMP

The following text has been copied from the GMP manual (5.1.2):

”Torbjörn Granlund wrote the original GMP library and is still the main developer. Code not explicitly attributed to others, was contributed by Torbjörn. Several other individuals and organizations have contributed GMP. Here is a list in chronological order on first contribution:

Gunnar Sjödin and Hans Riesel helped with mathematical problems in early versions of the library.

Richard Stallman helped with the interface design and revised the first version of this manual.

Brian Beuning and Doug Lea helped with testing of early versions of the library and made creative suggestions.

John Amanatides of York University in Canada contributed the function `mpz_probab_prime_p`.

Paul Zimmermann wrote the REDC-based `mpz_powm` code, the Schönhage-Strassen FFT multiply code, and the Karatsuba square root code. He also improved the Toom3 code for GMP 4.2. Paul sparked the development of GMP 2, with his comparisons between bignum packages. The ECMNET project Paul is organizing was a driving force behind many of the optimizations in GMP 3. Paul also wrote the new GMP 4.3 nth root code (with Torbjörn).

Ken Weber (Kent State University, Universidade Federal do Rio Grande do Sul) contributed now defunct versions of `mpz_gcd`, `mpz_divexact`, `mpn_gcd`, and `mpn_bdivmod`, partially supported by CNPq (Brazil) grant 301314194-2.

Per Bothner of Cygnus Support helped to set up GMP to use Cygnus' configure. He has also made valuable suggestions and tested numerous intermediary releases.

Joachim Hollman was involved in the design of the `mpf` interface, and in the `mpz` design revisions for version 2.

Bennet Yee contributed the initial versions of `mpz_jacobi` and `mpz_legendre`.

Andreas Schwab contributed the files `mpn/m68k/lshift.S` and `mpn/m68k/rshift.S` (now in `.asm`

form).

Robert Harley of Inria, France and David Seal of ARM, England, suggested clever improvements for population count. Robert also wrote highly optimized Karatsuba and 3-way Toom multiplication functions for GMP 3, and contributed the ARM assembly code.

Torsten Ekedahl of the Mathematical department of Stockholm University provided significant inspiration during several phases of the GMP development. His mathematical expertise helped improve several algorithms.

Linus Nordberg wrote the new configure system based on autoconf and implemented the new random functions.

Kevin Ryde worked on a large number of things: optimized x86 code, m4 asm macros, parameter tuning, speed measuring, the configure system, function inlining, divisibility tests, bit scanning, Jacobi symbols, Fibonacci and Lucas number functions, printf and scanf functions, perl interface, demo expression parser, the algorithms chapter in the manual, `gmpasm-mode.el`, and various miscellaneous improvements elsewhere.

Kent Boortz made the Mac OS 9 port.

Steve Root helped write the optimized alpha 21264 assembly code.

Gerardo Ballabio wrote the `gmpxx.h` C++ class interface and the C++ `istream` input routines.

Jason Moxham rewrote `mpz_fac_ui`.

Pedro Gimeno implemented the Mersenne Twister and made other random number improvements.

Niels Möller wrote the sub-quadratic GCD, extended GCD and jacobi code, the quadratic Hensel division code, and (with Torbjörn) the new divide and conquer division code for GMP 4.3. Niels also helped implement the new Toom multiply code for GMP 4.3 and implemented helper functions to simplify Toom evaluations for GMP 5.0. He wrote the original version of `mpn_mulmod_bnm1`, and he is the main author of the mini-gmp package used for gmp bootstrapping.

Alberto Zanoni and Marco Bodrato suggested the unbalanced multiply strategy, and found the optimal strategies for evaluation and interpolation in Toom multiplication.

Marco Bodrato helped implement the new Toom multiply code for GMP 4.3 and implemented most of the new Toom multiply and squaring code for 5.0. He is the main author of the current `mpn_mulmod_bnm1` and `mpn_mullo_n`. Marco also wrote the functions `mpn_invert` and `mpn_invertappr`. He is the author of the current combinatorial functions: binomial, factorial, multifactorial, primorial.

David Harvey suggested the internal function `mpn_bdiv_dbm1`, implementing division relevant to Toom multiplication. He also worked on fast assembly sequences, in particular on a fast AMD64 `mpn_mul_basecase`. He wrote the internal middle product functions `mpn_mulmid_basecase`, `mpn_toom42_mulmid`, `mpn_mulmid_n` and related helper routines.

Martin Boij wrote `mpn_perfect_power_p`.

Marc Glisse improved `gmpxx.h`: use fewer temporaries (faster), specializations of `numeric_limits` and `common_type`, C++11 features (move constructors, explicit bool conversion, UDL), make the conversion from `mpq_class` to `mpz_class` explicit, optimize operations where one argument is a small compile-time constant, replace some heap allocations by stack allocations. He also fixed

the eofbit handling of C++ streams, and removed one division from `mpq/aors.c`.

(This list is chronological, not ordered after significance. If you have contributed to GMP but are not listed above, please tell `gmp-devel@gmplib.org` about the omission!)

The development of floating point functions of GNU MP 2, were supported in part by the ESPRIT-BRA (Basic Research Activities) 6846 project POSSO (POlynomial System SOLving).

The development of GMP 2, 3, and 4 was supported in part by the IDA Center for Computing Sciences.

Thanks go to Hans Thorsen for donating an SGI system for the GMP test system environment.”

## E.1.2 Contributors to MPFR

The following text has been copied from the MPFR manual (3.1.2):

”The main developers of MPFR are Guillaume Hanrot, Vincent Lefèvre, Patrick Péliſſier, Philippe Théveny and Paul Zimmermann.

Sylvie Boldo from ENS-Lyon, France, contributed the functions `mpfr_agm` and `mpfr_log`. Sylvain Chevillard contributed the `mpfr_ai` function.

David Daney contributed the hyperbolic and inverse hyperbolic functions, the base-2 exponential, and the factorial function.

Alain Delplanque contributed the new version of the `mpfr_get_str` function.

Mathieu Dutour contributed the functions `mpfr_acos`, `mpfr_asin` and `mpfr_atan`, and a previous version of `mpfr_gamma`.

Laurent Fousse contributed the `mpfr_sum` function.

Emmanuel Jeandel, from ENS-Lyon too, contributed the generic hypergeometric code, as well as the internal function `mpfr_exp3`, a first implementation of the sine and cosine, and improved versions of `mpfr_const_log2` and `mpfr_const_pi`.

Ludovic Meunier helped in the design of the `mpfr_erf` code.

Jean-Luc Rémy contributed the `mpfr_zeta` code.

Fabrice Rouillier contributed the `mpfr_xxx_z` and `mpfr_xxx_q` functions, and helped to the Microsoft Windows porting.

Damien Stehlé contributed the `mpfr_get_ld_2exp` function.

We would like to thank Jean-Michel Muller and Joris van der Hoeven for very fruitful discussions at the beginning of that project, Torbjörn Granlund and Kevin Ryde for their help about design issues, and Nathalie Revol for her careful reading of a previous version of this documentation. In particular Kevin Ryde did a tremendous job for the portability of MPFR in 2002-2004.

The development of the MPFR library would not have been possible without the continuous support of INRIA, and of the LORIA (Nancy, France) and LIP (Lyon, France) laboratories. In particular the main authors were or are members of the PolKA, Spaces, Cacao and Caramel project-teams at LORIA and of the Arénaire and AriC project-teams at LIP.

This project was started during the Fiable (reliable in French) action supported by INRIA, and continued during the AOC action. The development of MPFR was also supported by a grant (202F0659 00 MPN 121) from the Conseil Régional de Lorraine in 2002, from INRIA by an "associate engineer" grant (2003-2005), an "opération de développement logiciel" grant (2007-2009), and the post-doctoral grant of Sylvain Chevillard in 2009-2010. The MPFR-MPC workshop in June 2012 was partly supported by the ERC grant ANTICS of Andreas Enge."

### E.1.3 Contributors to FLINT

The following text has been copied from the FLINT manual (2.4.3):

xxxx

### E.1.4 Contributors to ARB

The following text has been copied from the FLINT manual (2.4.3):

xxxx

### E.1.5 Contributors to MPFRC++

The main developer of MPFRC++ is Pavel Holoborodko.

Contributors: Dmitriy Gubanov, Konstantin Holoborodko, Brian Gladman, Helmut Jarausch, Fokko Beekhof, Ulrich Mutze, Heinz van Saanen, Pere Constans, Peter van Hoof, Gael Guennebaud, Tsai Chia Cheng, Alexei Zubanov, Jauhien Piatlicki, Victor Berger, John Westwood.

### E.1.6 Contributors to Eigen

The following statement is copied from the Eigen Homepage:

"The Eigen project was started by Benoît Jacob (founder) and Gaël Guennebaud (guru). Many other people have since contributed their talents to help make Eigen successful. Here's an alphabetical list: (note to contributors: do add yourself!)

Philip Avery: Fix bug and add functionality to AutoDiff module

Abraham Bachrach: Added functions for cwise min/max with a scalar

Sebastien Barthelemy: Fix EIGEN\_INITIALIZE\_MATRICES\_BY\_NAN

Carlos Becker: Wrote some of the pages of the tutorial

David Benjamin: Artwork: the owls

Cyrille Berger: Fix error in logic of installation script

Armin Berres: Lots of fixes (compilation warnings and errors)

Jose Luis Blanco: Build fixes for MSVC and AMD64, correction in docs

Mark Borgerding: FFT module

Romain Bossart: Updates to Sparse solvers

Kolja Brix: Added documentation to Householder module, fixes for ARPACK wrapper and KroneckerProduct

Gauthier Brun: Making a start with a divide-and-conquer SVD implementation

Thomas Capricelli: Migration to mercurial, Non-linear optimization and numerical differentiation, cron-job to update the online dox

Nicolas Carre: Making a start with a divide-and-conquer SVD implementation

Jean Ceccato: Making a start with a divide-and-conquer SVD implementation

Andrew Coles: Fixes (including a compilation error)r  
Marton Danoczy: MSVC compilation fix, support for ARM NEON with Clang 3.0 and LLVM-GCC  
Jeff Dean: Fix in vectorized square root for small arguments  
Christian Ehrlicher: MSVC compilation fix  
Daniel Gomez Ferro: Improvements in Sparse and in matrix product  
Rohit Garg: Vectorized quaternion and cross products, improved integer product  
Mathieu Gautier: QuaternionMap and related improvements  
Anton Gladky: Visual Studio 2008 and GCC 4.6 compilation fixes  
Stuart Glaser: Prevent allocations in LU decomposition  
Marc Glisse: C++11 compilation issues (suffices for literals)  
Frederic Gosselin: Improve filter for hidden files in CMake  
GaÃ±l Guennebaud: Core developer  
Philippe Hamelin: Allow CMake project to be included in another project  
Marcus D. Hanwell: CMake improvements. Marcus is a developer at Kitware!  
David Harmon: Arpack support module  
Chen-Pang He: Many improvements to MatrixFunctions and KroneckerProduct modules  
Hauke Heibel: Extended matrix functions, STL compatibility, Splines, CMake improvements, and more ...  
Christoph Hertzberg: Quaternions, shifts for Cholmod, bug fixes, lots of user support on forums and IRC  
Pavel Holoborodko: Multi-precision support with MPFR C++  
Tim Holy: Improvements to tutorial, LDLT update and downdate  
Intel: Back-end to Intel Math Kernel Library (MKL)  
Trevor Irons: Square root for complex numbers, fix compile errors and mistake in docs  
BenoÃ©t Jacob : Core developer  
Bram de Jong: Improvement to benchmark suite  
Kiboom Kim: Implement \*= /= \* / operations for VectorwiseOp  
Claas KÃhler: Improvements to Fortran and FFTW in CMake  
Alexey Korepanov: Add RealQZ class  
Igor Krivenko: Properly cast constants when using non-standard scalars  
Marijn Kruisselbrink: CMake fixes  
Moritz Lenz: Allow solving transposed problem with SuperLU  
Sebastian Lippner: MSVC compilation support  
Daniel Lowenberg: Add SparseView class  
David J. Luitz: Bug fix for sparse \* dense matrix product  
Angelos Mantzaflaris: Fix to allow IncompleteLUT to be used with MPFR  
D J Marcin: Fix operator & precedence bug  
Konstantinos A. Margaritis: AltiVec and ARM NEON vectorization  
Ricard Marxer: Reverse, redux improvements, the count() method, some dox  
Vincenzo Di Massa: CMake fix  
Christian Mayer: Early code review and input in technical/design discussions  
Frank Meier-DÃrnberg: MSVC compatibility fixes  
Keir Mierle: LDLT decomposition and other improvements, help with MPL relicensing  
Laurent Montel: CMake improvements. Laurent is (with Alexander) one of the CMake gurus at KDE!  
Eamon Nerbonne: Compilation fixes for win32  
Alexander Neundorf: CMake improvements. Alexander is (with Laurent) one of the CMake gurus at KDE!

Jason Newton: Componentwise tangent functions  
Jitse Niesen: Matrix functions, large improvements in the Eigenvalues module and in the docs, and more ...  
Desire Nuentsa: Many improvements to Sparse module: SparseLU, SparseQR, ILUT, PaStiX-Support, etc  
Jan Oberländer: Compatibility with termios.h  
Jos van den Oever: Compilation fix  
Michael Olbrich: Early patches, including the initial loop meta-unroller  
Simon Pilgrim: Optimizations for NEON  
Bjorn Piltz: Visual C compilation fix  
Benjamin Piwowarski: Add conservativeResize() for sparse matrices  
Zach Ploskey: Copy-editing of tutorial  
Giacomo Po: MINRES iterative solver  
Sergey Popov: Fix bug in SelfAdjointEigenSolver  
Manoj Rajagopalan: Introduce middleRows() / middleCols(), bug fix for nonstandard numeric types  
Stjepan Rajko: MSVC compatibility fix  
Jure Repinc: CMake fixes  
Kenneth Frank Riddile: Lots of Windows/MSVC compatibility fixes, handling of alignment issues  
Adolfo Rodriguez: Prevent allocations in matrix decompositions  
Peter Román: Support for SuperLU's ILU factorization  
Oliver Ruepp: Bug fix in sparse matrix product with row-major matrices  
Radu Bogdan Rusu: Fix compilation warning  
Guillaume Saupin: Skyline matrices  
Michael Schmidt: Fix in assembly when identifying CPU  
Jakob Schwendner: Test for unaligned quaternions  
Martin Senst: Bug fix for empty matrices  
Benjamin Schindler: gdb pretty printers  
Michael Schmidt: Compilation fix connected to min/max  
Dennis Schridde: New typedefs like AlignedBox3f  
Jakob Schwendner: Benchmark for Geometry module  
Sameer Sheorey: Fix gdb pretty printer for variable-size matrices  
Andy Somerville: Functions to get intersection between two ParametrizedLines  
Alex Stapleton: Help with tough C++ questions  
Adam Szalkowski: Bug fix in MatrixBase::makeHouseholder()  
Adolfo Rodriguez: Tsouroukisdissian Version of JacobiSVD that pre-allocates its resources  
Piotr Trojanek: QCC compilation fixes  
Anthony Truchet: Bugfix in QTransform and QMatrix support  
James Richard Tyrer: CMake fix  
Rhys Ulerich: Pkg-config support, improved GDB pretty-printer  
Ingmar Vanhassel: CMake fix  
Scott Wheeler: Documentation improvements  
Urs Wolfer: Fixed a serious warning  
Manuel Yguel: Bug fixes, work on inverse-with-check, the Polynomial module  
Pierre Zoppitelli: Making a start with a divide-and-conquer SVD implementation

Eigen is also using code that we copied from other sources. They are acknowledged in our sources and in the Mercurial history, but let's also mention them here:

Intel Corporation SSE code for 4x4 matrix inversion taken from here. Tim Davis AMD re-ordering simplicial sparse Cholesky factorization adapted from SuiteSparse Julien Pommier SSE implementation of exp,log,cos,sin math functions from GMM++ Yousef Saad IncompleteLUT preconditioner coming from ITSOL Minpack authors Algorithms for non linear optimization.

Special thanks to Tuxfamily for the wonderful quality of their services, and the GCC Compile Farm Project that gives us access to many various systems including ARM NEON. ”

### E.1.7 Contributors to Boost Multiprecision

The main authors of Boost Multiprecision are John Maddock and Christopher Kormanyos.

The Acknowledgements section states:

”This library would not have happened without:

Christopher Kormanyos’ C++ decimal number code.

Paul Bristow for patiently testing, and commenting on the library.

All the folks at GMP, MPFR and libtommath, for providing the ”guts” that makes this library work.

”The Art Of Computer Programming”, Donald E. Knuth, Volume 2: Seminumerical Algorithms, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), xiv+762pp.  
ISBN 0-201-89684-2”

### E.1.8 Contributors to Boost Math

The main authors of the Boost Math Toolkit are Paul A. Bristow, Hubert Holin, Christopher Kormanyos, Bruno Lalande, John Maddock, Johan RÃ©de, Benjamin Sobotta, Gautam Sewani, Thijs van den Berg, Daryle Walker, and Xiaogang Zhang.

The Credits and Acknowledgements section states:

”Hubert Holin started the Boost.Math library. The Quaternions, Octonions, inverse hyperbolic functions, and the sinus cardinal functions are his.

Daryle Walker wrote the integer gcd and lcm functions.

John Maddock started the special functions, the beta, gamma, erf, polynomial, and factorial functions are his, as is the ”Toolkit” section, and many of the statistical distributions.

Paul A. Bristow threw down the challenge in A Proposal to add Mathematical Functions for Statistics to the C++ Standard Library to add the key math functions, especially those essential for statistics. After JM accepted and solved the difficult problems, not only numerically, but in full C++ template style, PAB implemented a few of the statistical distributions. PAB also tirelessly proof-read everything that JM threw at him (so that all remaining editorial mistakes are his fault).

Xiaogang Zhang worked on the Bessel functions and elliptic integrals for his Google Summer of Code project 2006.

Bruno Lalande submitted the ”compile time power of a runtime base” code.

Johan RÃ©de wrote the optimised floating-point classification and manipulation code, and nonfinite facets to permit C99 output of infinities and NaNs. (nonfinite facets were not added until

Boost 1.47 but had been in use with Boost.Spirit). This library was based on a suggestion from Robert Ramey, author of Boost.Serialization. Paul A. Bristow expressed the need for better handling of Input & Output of NaN and infinity for the C++ Standard Library and suggested following the C99 format.

Antony Polukhin improved lexical cast avoiding stringstream so that it was no longer necessary to use a global C99 facet to handle nonfinites.

Håkan Ardu, Boris Gubenko, John Maddock, Markus Schäufel and Olivier Verdier tested the floating-point library and Martin Bonner, Peter Dimov and John Maddock provided valuable advice.

Gautam Sewani coded the logistic distribution as part of a Google Summer of Code project 2008.

M. A. (Thijs) van den Berg coded the Laplace distribution. (Thijs has also threatened to implement some multivariate distributions).

Thomas Mang requested the inverse gamma and chi squared distributions for Bayesian applications and helped in their implementation, and provided a nice example of their use.

Professor Nico Temme for advice on the inverse incomplete beta function.

Victor Shoup for NTL, without which it would have much more difficult to produce high accuracy constants, and especially the tables of accurate values for testing.

We are grateful to Joel Guzman for helping us stress-test his Boost.Quickbook program used to generate the html and pdf versions of this document, adding several new features en route.

Plots of the functions and distributions were prepared in W3C standard Scalable Vector Graphic (SVG) format using a program created by Jacob Voynko during a Google Summer of Code (2007). From 2012, the latest versions of all Internet Browsers have support for rendering SVG (with varying quality). Older versions, especially (Microsoft Internet Explorer (before IE 9) lack native SVG support but can be made to work with Adobe's free SVG viewer plugin). The SVG files can be converted to JPEG or PNG using Inkscape.

We are also indebted to Matthias Schabel for managing the formal Boost-review of this library, and to all the reviewers - including Guillaume Melquiond, Arnaldur Gylfason, John Phillips, Stephan Tolsdorf and Jeff Garland - for their many helpful comments.

Thanks to Mark Coleman and Georgi Boshnakov for spot test values from Wolfram Mathematica, and of course, to Eric Weisstein for nurturing Wolfram MathWorld, an invaluable resource.

The Skew-normal distribution and Owen's t function were written by Benjamin Sobotta.”

### E.1.9 Contributors to Boost Random

The main authors of the Boost Random are Jens Maurer and Steven Watanabe.

The History and Acknowledgements section states:

”In November 1999, Jeet Sukumaran proposed a framework based on virtual functions, and later sketched a template-based approach. Ed Brey pointed out that Microsoft Visual C++ does not support in-class member initializations and suggested the enum workaround. Dave Abrahams highlighted quantization issues.

The first public release of this random number library materialized in March 2000 after extensive discussions on the boost mailing list. Many thanks to Beman Dawes for his original `min_rand` class, portability fixes, documentation suggestions, and general guidance. Harry Erwin sent a header file which provided additional insight into the requirements. Ed Brey and Beman Dawes wanted an iterator-like interface.

Beman Dawes managed the formal review, during which Matthias Troyer, Csaba Szepesvari, and Thomas Holenstein gave detailed comments. The reviewed version became an official part of boost on 17 June 2000.

Gary Powell contributed suggestions for code cleanliness. Dave Abrahams and Howard Hinnant suggested to move the basic generator templates from namespace `boost::detail` to `boost::random`.

Ed Brey asked to remove superfluous warnings and helped with `uint64_t` handling. Andreas Scherer tested with MSVC. Matthias Troyer contributed a lagged Fibonacci generator. Michael Stevens found a bug in the copy semantics of `normal_distribution` and suggested documentation improvements.”

### E.1.10 Contributors to Boost Odeint

The main authors of the Boost Odeint are Karsten Ahnert and Mario Mulansky.

The History and Acknowledgements section states:

Acknowledgments

Steven Watanabe for managing the Boost review process.

All people who participated in the `odeint` review process on the Boost mailing list.

Paul Bristow for helping with the documentation.

The Google Summer Of Code (GSOC) program for funding and Andrew Sutton for supervising us during the GSOC and for lots of useful discussions and feedback about many implementation details..

Joachim Faulhaber for motivating us to participate in the Boost review process and many detailed comments about the library.

All users of `odeint`. They are the main motivation for our efforts.

Contributors

Andreas Angelopoulos implemented the sparse matrix implicit Euler stepper using the MTL4 library.

Rajeev Singh implemented the stiff Van der Pol oscillator example.

Sylwester Arabas improved the documentation.

Denis Demidov provided the adaption to the VexCL and Viennacl libraries.

Christoph Koke provided improved binders.

Lee Hodgkinson provided the black hole example.

Michael Morin fixed several typos in the documentation and the source code comments.

### E.1.11 Contributors to NLOpt

The main author of NLOpt is Steven G. Johnson.

The Acknowledgements section states:

"We are grateful to the many authors who have published useful optimization algorithms implemented in NLOpt, especially those who have provided free/open-source implementations of their algorithms.

Please cite these authors if you use their code or the implementation of their algorithm in NLOpt. See the documentation for the appropriate citation for each of the algorithms in NLOpt — please see the Citing NLOpt information."

# Appendix F

## Licenses

### F.1 GNU Licenses

#### F.1.1 GNU General Public License, Version 2

GNU LIBRARY GENERAL PUBLIC LICENSE  
Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.  
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.] Preamble The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, 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 this service 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 make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming

the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

## TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

â€“a) The modified work must itself be a software library. â€“b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change. â€“c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License. â€“d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful. (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License. However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library".

The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

â€“a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.) â€“b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution. â€“c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place. â€“d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy. For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

â€“a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above. â€“b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), 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 distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library 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 Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions

are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

#### NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "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 LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY 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 LIBRARY (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 LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

#### END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey 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 library's name and an idea of what it does. Copyright (C) year name of author  
This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library 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 Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA. Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

signature of Ty Coon, 1 April 1990 Ty Coon, President of Vice That's all there is to it!

## F.1.2 GNU Library General Public License, Version 2

### GNU LIBRARY GENERAL PUBLIC LICENSE Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.] Preamble The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, 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 this service 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 make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended

to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

#### TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

âA a) The modified work must itself be a software library. âA b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change. âA c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License. âA d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful. (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are

not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License. However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

â€“a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.) â€“b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution. â€“c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place. â€“d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy. For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

â€“a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above. â€“b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason

(not limited to patent issues), 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 distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library 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 Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

#### NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "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 LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR RE-

DISTRIBUTE THE LIBRARY 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 LIBRARY (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 LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

#### END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey 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 library's name and an idea of what it does. Copyright (C) year name of author  
This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library 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 Library General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA. Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

signature of Ty Coon, 1 April 1990 Ty Coon, President of Vice That's all there is to it!

### F.1.3 GNU Lesser General Public License, Version 3

GNU LESSER 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.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the object code with a copy of the GNU GPL and this license document.

#### 4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d) Do one of the following:
  - 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
  - 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
  - e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

#### 5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

#### 6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser 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 Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

## F.1.4 GNU General Public License, Version 3

### 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: (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 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 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.

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 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, 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.

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; 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 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 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 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.

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, 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>>

## F.1.5 GNU Free Documentation License, Version 1.3

GNU Free Documentation License  
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 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 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 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), 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.

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 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.

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 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.

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.

## F.2 Other Licenses

### F.2.1 Mozilla Public License, Version 2.0

#### Mozilla Public License Version 2.0

##### 1. Definitions

###### 1.1. "Contributor"

means each individual or legal entity that creates, contributes to the creation of, or owns Covered Software.

###### 1.2. "Contributor Version"

means the combination of the Contributions of others (if any) used by a Contributor and that particular Contributor's Contribution.

###### 1.3. "Contribution"

means Covered Software of a particular Contributor.

###### 1.4. "Covered Software"

means Source Code Form to which the initial Contributor has attached the notice in Exhibit A, the Executable Form of such Source Code Form, and Modifications of such Source Code Form, in each case including portions thereof.

###### 1.5. "Incompatible With Secondary Licenses"

means

a.that the initial Contributor has attached the notice described in Exhibit B to the Covered Software; or

b.that the Covered Software was made available under the terms of version 1.1 or earlier of the License, but not also under the terms of a Secondary License.

###### 1.6. "Executable Form"

means any form of the work other than Source Code Form.

###### 1.7. "Larger Work"

means a work that combines Covered Software with other material, in a separate file or files, that is not Covered Software.

###### 1.8. "License"

means this document.

###### 1.9. "Licensable"

means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently, any and all of the rights conveyed by this License.

###### 1.10. "Modifications"

means any of the following:

a.any file in Source Code Form that results from an addition to, deletion from, or modification of the contents of Covered Software; or

b.any new file in Source Code Form that contains any Covered Software.

###### 1.11. "Patent Claims" of a Contributor

means any patent claim(s), including without limitation, method, process, and apparatus claims, in any patent Licensable by such Contributor that would be infringed, but for the grant of the License, by the making, using, selling, offering for sale, having made, import, or transfer of either its Contributions or its Contributor Version.

###### 1.12. "Secondary License"

means either the GNU General Public License, Version 2.0, the GNU Lesser General Public License, Version 2.1, the GNU Affero General Public License, Version 3.0, or any later versions of those licenses.

###### 1.13. "Source Code Form"

means the form of the work preferred for making modifications.

###### 1.14. "You" (or "Your")

means an individual or a legal entity exercising rights under this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

## 2. License Grants and Conditions

### 2.1. Grants

Each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license:

- a.under intellectual property rights (other than patent or trademark) Licensable by such Contributor to use, reproduce, make available, modify, display, perform, distribute, and otherwise exploit its Contributions, either on an unmodified basis, with Modifications, or as part of a Larger Work; and
- b.under Patent Claims of such Contributor to make, use, sell, offer for sale, have made, import, and otherwise transfer either its Contributions or its Contributor Version.

### 2.2. Effective Date

The licenses granted in Section 2.1 with respect to any Contribution become effective for each Contribution on the date the Contributor first distributes such Contribution.

### 2.3. Limitations on Grant Scope

The licenses granted in this Section 2 are the only rights granted under this License. No additional rights or licenses will be implied from the distribution or licensing of Covered Software under this License. Notwithstanding Section 2.1(b) above, no patent license is granted by a Contributor:

- a. for any code that a Contributor has removed from Covered Software; or
- b. for infringements caused by: (i) Your and any other third party's modifications of Covered Software, or (ii) the combination of its Contributions with other software (except as part of its Contributor Version); or

c.under Patent Claims infringed by Covered Software in the absence of its Contributions.

This License does not grant any rights in the trademarks, service marks, or logos of any Contributor (except as may be necessary to comply with the notice requirements in Section 3.4).

### 2.4. Subsequent Licenses

No Contributor makes additional grants as a result of Your choice to distribute the Covered Software under a subsequent version of this License (see Section 10.2) or under the terms of a Secondary License (if permitted under the terms of Section 3.3).

### 2.5. Representation

Each Contributor represents that the Contributor believes its Contributions are its original creation(s) or it has sufficient rights to grant the rights to its Contributions conveyed by this License.

### 2.6. Fair Use

This License is not intended to limit any rights You have under applicable copyright doctrines of fair use, fair dealing, or other equivalents.

### 2.7. Conditions

Sections 3.1, 3.2, 3.3, and 3.4 are conditions of the licenses granted in Section 2.1.

## 3. Responsibilities

### 3.1. Distribution of Source Form

All distribution of Covered Software in Source Code Form, including any Modifications that You create or to which You contribute, must be under the terms of this License. You must inform recipients that the Source Code Form of the Covered Software is governed by the terms of this License, and how they can obtain a copy of this License. You may not attempt to alter or restrict the recipients' rights in the Source Code Form.

### 3.2. Distribution of Executable Form

If You distribute Covered Software in Executable Form then:

- a.such Covered Software must also be made available in Source Code Form, as described in Section 3.1, and You must inform recipients of the Executable Form how they can obtain a copy of such Source Code

Form by reasonable means in a timely manner, at a charge no more than the cost of distribution to the recipient; and

b. You may distribute such Executable Form under the terms of this License, or sublicense it under different terms, provided that the license for the Executable Form does not attempt to limit or alter the recipients' rights in the Source Code Form under this License.

### 3.3. Distribution of a Larger Work

You may create and distribute a Larger Work under terms of Your choice, provided that You also comply with the requirements of this License for the Covered Software. If the Larger Work is a combination of Covered Software with a work governed by one or more Secondary Licenses, and the Covered Software is not Incompatible With Secondary Licenses, this License permits You to additionally distribute such Covered Software under the terms of such Secondary License(s), so that the recipient of the Larger Work may, at their option, further distribute the Covered Software under the terms of either this License or such Secondary License(s).

### 3.4. Notices

You may not remove or alter the substance of any license notices (including copyright notices, patent notices, disclaimers of warranty, or limitations of liability) contained within the Source Code Form of the Covered Software, except that You may alter any license notices to the extent required to remedy known factual inaccuracies.

### 3.5. Application of Additional Terms

You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Software. However, You may do so only on Your own behalf, and not on behalf of any Contributor. You must make it absolutely clear that any such warranty, support, indemnity, or liability obligation is offered by You alone, and You hereby agree to indemnify every Contributor for any liability incurred by such Contributor as a result of warranty, support, indemnity or liability terms You offer. You may include additional disclaimers of warranty and limitations of liability specific to any jurisdiction.

## 4. Inability to Comply Due to Statute or Regulation

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Software due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be placed in a text file included with all distributions of the Covered Software under this License. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

## 5. Termination

5.1. The rights granted under this License will terminate automatically if You fail to comply with any of its terms. However, if You become compliant, then the rights granted under this License from a particular Contributor are reinstated (a) provisionally, unless and until such Contributor explicitly and finally terminates Your grants, and (b) on an ongoing basis, if such Contributor fails to notify You of the non-compliance by some reasonable means prior to 60 days after You have come back into compliance. Moreover, Your grants from a particular Contributor are reinstated on an ongoing basis if such Contributor notifies You of the non-compliance by some reasonable means, this is the first time You have received notice of non-compliance with this License from such Contributor, and You become compliant prior to 30 days after Your receipt of the notice.

5.2. If You initiate litigation against any entity by asserting a patent infringement claim (excluding declaratory judgment actions, counter-claims, and cross-claims) alleging that a Contributor Version directly or indirectly infringes any patent, then the rights granted to You by any and all Contributors for the Covered Software under Section 2.1 of this License shall terminate.

5.3. In the event of termination under Sections 5.1 or 5.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or Your distributors under this License prior to termination shall survive termination.

## 6. Disclaimer of Warranty

Covered Software is provided under this License on an "as is" basis, without warranty of any kind, either expressed, implied, or statutory, including, without limitation, warranties that the Covered Software is free of defects, merchantable, fit for a particular purpose or non-infringing. The entire risk as to the quality and performance of the Covered Software is with You. Should any Covered Software prove defective in any respect, You (not any Contributor) assume the cost of any necessary servicing, repair, or correction. This disclaimer of warranty constitutes an essential part of this License. No use of any Covered Software is authorized under this License except under this disclaimer.

## 7. Limitation of Liability

Under no circumstances and under no legal theory, whether tort (including negligence), contract, or otherwise, shall any Contributor, or anyone who distributes Covered Software as permitted above, be liable to You for any direct, indirect, special, incidental, or consequential damages of any character including, without limitation, damages for lost profits, loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses, even if such party shall have been informed of the possibility of such damages. This limitation of liability shall not apply to liability for death or personal injury resulting from such party's negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to You.

## 8. Litigation

Any litigation relating to this License may be brought only in the courts of a jurisdiction where the defendant maintains its principal place of business and such litigation shall be governed by laws of that jurisdiction, without reference to its conflict-of-law provisions. Nothing in this Section shall prevent a party's ability to bring cross-claims or counter-claims.

## 9. Miscellaneous

This License represents the complete agreement concerning the subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not be used to construe this License against a Contributor.

## 10. Versions of the License

### 10.1. New Versions

Mozilla Foundation is the license steward. Except as provided in Section 10.3, no one other than the license steward has the right to modify or publish new versions of this License. Each version will be given a distinguishing version number.

### 10.2. Effect of New Versions

You may distribute the Covered Software under the terms of the version of the License under which You originally received the Covered Software, or under the terms of any subsequent version published by the license steward.

### 10.3. Modified Versions

If you create software not governed by this License, and you want to create a new license for such software, you may create and use a modified version of this License if you rename the license and remove any references to the name of the license steward (except to note that such modified license differs from this License).

### 10.4. Distributing Source Code Form that is Incompatible With Secondary Licenses

If You choose to distribute Source Code Form that is Incompatible With Secondary Licenses under the terms of this version of the License, the notice described in Exhibit B of this License must be attached.

## Exhibit A - Source Code Form License Notice

This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, You can obtain one at <http://mozilla.org/MPL/2.0/>.

If it is not possible or desirable to put the notice in a particular file, then You may include the notice in a location (such as a LICENSE file in a relevant directory) where a recipient would be likely to look for such a notice.

You may add additional accurate notices of copyright ownership.

Exhibit B - "Incompatible With Secondary Licenses" Notice

This Source Code Form is "Incompatible With Secondary Licenses", as defined by the Mozilla Public License, v. 2.0.

### **F.2.2 Boost Software License, Version 1.0**

Boost Software License Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### F.2.3 MIT License

#### MIT License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# **Part IX**

## **Back Matter**

# Bibliography

- Abdel-Aty, S.H. 1954. Approximate formulae for the percentage points and the probability integral of the noncentral  $\chi^2$  distribution. *Biometrika*, **41**, 538–540. 676
- Abramowitz, M., & Stegun, I.A. 1970. *Handbook of Mathematical Functions*. New York: Dekker. Available as <http://www.math.sfu.ca/~cbm/aands/>. 126, 196, 572, 666
- Adler, Joseph. 2012. *R in a Nutshell*. 2nd edn. O'Reilly Media. Online resource: <http://shop.oreilly.com/product/0636920022008.do>. 825
- Agresti, Alan. 2010. *Analysis of ordinal categorical data*. 2nd edn. John Wiley & Sons. 734
- Ahnert, Karsten, & Mulansky, Mario. 2013 (June). *Boost Odeint Library*. <http://www.boost.org/libs/odeint/>. Documentation available as [http://www.boost.org/doc/libs/1\\_54\\_0/libs/numeric/odeint/doc/html/index.html](http://www.boost.org/doc/libs/1_54_0/libs/numeric/odeint/doc/html/index.html). 615
- Akahira, M., & Torigoe, N. 1998. A new higher order approximation to a percentage point of the distribution of the sample correlation coefficient. *J. Japan Statist. Soc*, **28**(1), 45–57. Available at [http://www.tulips.tsukuba.ac.jp/dspace/bitstream/2241/118439/1/JJSS\\_28-1.pdf](http://www.tulips.tsukuba.ac.jp/dspace/bitstream/2241/118439/1/JJSS_28-1.pdf). 654
- Akahira, M., Sato, M., & Torigoe, N. 1995. On the new approximation to non-central t-distributions. *J. Japan Statist. Soc*, **25**(1), 1–18. Available at [http://www.tulips.tsukuba.ac.jp/dspace/bitstream/2241/118436/1/JJSS\\_25-1.pdf](http://www.tulips.tsukuba.ac.jp/dspace/bitstream/2241/118436/1/JJSS_25-1.pdf). 661, 686, 687, 689, 695, 759, 767
- Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. Du, Greenbaum, A., Hammarling, S., McKenney, A., & Sorensen, D. 1999. *LAPACK Users' Guide*. 3rd edn. Society for Industrial and Applied Mathematics (SIAM). LAPACK Homepage: <http://www.netlib.org/lapack>. 365
- Anderson, T. W. 2003. *An Introduction to Multivariate Statistical Analysis*. 3 edn. John Wiley & Sons, Inc. 635, 652
- András, S., & Baricz, Á. 2008. Properties of the probability density function of the non-central chi-squared distribution. *Journal of Mathematical Analysis and Applications*, **346**(2), 395 – 402. Available at <http://www.sciencedirect.com/science/article/pii/S0022247X08005696>. 673
- Arndt, J. 2011. *Matters Computational - Ideas, Algorithms, Source Code*. Springer. Available from <http://www.jjj.de/fxt/#fxtbook>. 491
- Arnold, H. J. 1965. Small sample power for the one sample Wilcoxon test for non-normal shift alternatives. *Ann. Stat. Math*, **36**(6), 1767–1778. Available at <http://projecteuclid.org/euclid.aoms/1177699805>. 736

- Barker, V.A., Blackford, S., Dongarra, J., Croz, J. Du, Hammarling, S., Marinova, M., Wasniewski, J., & Yalamov, P. 2001. *LAPACK95 Users' Guide*. 1st edn. Society for Industrial and Applied Mathematics (SIAM). LAPACK Homepage: <http://www.netlib.org/lapack>. 365
- Benton, Denise, & Krishnamoorthy, K. 2003. Computing discrete mixtures of continuous distributions: noncentral chisquare, noncentral t and the distribution of the square of the sample multiple correlation coefficient. *Computational Statistics & Data Analysis*, **14**, 249–267. Available as <http://www.ucs.louisiana.edu/~kxk4695/CSDA-03.pdf>. 604, 610, 657, 681, 686
- Bernstein, Dennis S. 2009. *Matrix mathematics: theory, facts, and formulas*. 2nd edn. Princeton University Press. 366
- Birgin, E. G., & Martínez, J. M. 2008. Improving ultimate convergence of an augmented Lagrangian method. *Optimization Methods and Software*, **23**(2), 177–195. 509
- Björck, Å., & Hammarling, S. 1983. A Schur method for the square root of a matrix. *Linear Algebra Appl.*, **52/53**, 127–140. 472
- Blomquist, F. 2005. *Automatische a priori Fehlerabschätzungen zur Entwicklung optimaler Algorithmen und Intervallfunktionen in C-XSC (in German)*. Available from [http://www.math.uni-wuppertal.de/~xsc/literatur/a\\_priori.pdf](http://www.math.uni-wuppertal.de/~xsc/literatur/a_priori.pdf). 360
- Blomquist, F., Hofschuster, W., & Krämer, W. 2008a. A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range. In: Cuyt, Annie, Krämer, Walter, Luther, Wolfram, & Markstein, Peter (eds), *Numerical Validation in Current Hardware Architectures*. Dagstuhl Seminar Proceedings, no. 08021. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. Available from <http://drops.dagstuhl.de/opus/volltexte/2008/1445>. 360
- Blomquist, F., Hofschuster, W., & Krämer, W. 2008b. *Real and Complex Staggered (Interval) Arithmetics with Wide Exponent Range (in German)*. Preprint BUGHW-WRSWT 2008/1, Universität Wuppertal, 2008, Available at [http://www2.math.uni-wuppertal.de/~xsc/preprints/prep\\_08\\_1.pdf](http://www2.math.uni-wuppertal.de/~xsc/preprints/prep_08_1.pdf). 360
- Blomquist, F., Hofschuster, W., & Krämer, W. 2012. *Umfangreiche C-XSC-Langzahlpakete für beliebig genaue reelle und komplexe Intervallrechnung*. Preprint 2012/2, Universität Wuppertal, 2012 (in german). Online resource: [http://www2.math.uni-wuppertal.de/~xsc/xsc/cxsc\\_software.html#mpfr-mpfi](http://www2.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html#mpfr-mpfi). The manual is available from [http://www2.math.uni-wuppertal.de/~xsc/preprints/prep\\_12\\_2.pdf](http://www2.math.uni-wuppertal.de/~xsc/preprints/prep_12_2.pdf). 360
- Bortz, J., Lienert, G. A., & Boehnke, K. 1990. *Distribution Free Methods in Biostatistics (German)*. Springer-Verlag Inc. 748
- Box, G.E.P. 1949. A general distribution theory for a class of likelihood criteria. *Biometrika*, **36**, 317–346. 635, 636
- Box, M. J. 1965. A new method of constrained optimization and a comparison with other methods. *Computer J.*, **8**(1), 42–52. 505
- Brent, R. 1972. *Algorithms for Minimization without Derivatives*. 1st edn. Prentice-Hall (Reprinted by Dover, 2002.). 505

- Brent, R. P. 1992a. On the Periods of Generalized Fibonacci Recurrences. *Computer Sciences Laboratory Australian National University*. 514
- Brent, R. P. 1992b. Uniform random number generators for supercomputers. *Proc. of Fifth Australian Supercomputer Conference, Melbourne*, 704–706. 514
- Bristow, Paul A., Holin, Hubert, Kormanyos, Christopher, Lalande, Bruno, Maddock, John, RÃ©de, Johan, Sobotta, Benjamin, Sewani, Gautam, van den Berg, Thijs, Walker, Daryle, , & Zhang, Xiaogang. 2013 (June). *Boost Math Toolkit Library*. <http://www.boost.org/libs/math/>. Documentation available as [http://www.boost.org/doc/libs/1\\_54\\_0/libs/math/doc/html/index.html](http://www.boost.org/doc/libs/1_54_0/libs/math/doc/html/index.html). 3, 517, 610, 685, 686
- Broda, S.A., & Paolella, M. 2007. Saddlepoint Approximations for the Doubly Noncentral t Distribution. *Computational Statistics & Data Analysis*, 51, 2907–2918. Postprint available as [http://www.zora.uzh.ch/16956/2/broda\\_dnct\\_07V.pdf](http://www.zora.uzh.ch/16956/2/broda_dnct_07V.pdf). 686, 692
- Butler, R.W. 2007. *Saddlepoint Approximations with Applications*. 1st edn. Cambridge University Press. 675
- Butler, R.W., & Paolella, M.S. 2002. Calculating the density and distribution function for the singly and doubly noncentral F. *Statistics and Computing*, 12, 9–16. Working paper (1999) available at <http://129.82.101.115/statresearch/stattechreports/Technical%20Reports/1999/99-6%20Butler%20Paolella.pdf>. 698, 703, 704, 705
- Butler, R.W., & Wood, A.T.A. 2002. Laplace approximations for hypergeometric functions with matrix arguments. *Ann. Statist.*, 30, 1155–1177. Available at (permanent URL): <http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid-aos/1031689021>. 230, 236, 626, 627, 717, 718, 727
- Butler, R.W., & Wood, A.T.A. 2003. Laplace approximation for Bessel functions of matrix argument. *J. Comput. Appl. Math.*, 155(2), 359–382. 628
- Butler, R.W., & Wood, A.T.A. 2005. Approximation of power in multivariate analysis. *Statistics and Computing*, 15(4), 281–287. 712, 715, 716, 725
- Carlson, B. C., & Gustafson, J. L. 1994. Asymptotic Approximations for Symmetric Elliptic Integrals. *Siam Journal on Mathematical Analysis*, 25. 263, 533
- Carlson, B.C. 1995. Numerical computation of real or complex elliptic integrals. *Numerical Algorithms*, 10(1), 13–26. 263, 533
- Casagrande, J. T., Pike, M. C., & Smith, P. G. 1978. The Power Function of the "Exact" Test for Comparing Two Binomial Distributions. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 27(2), 176–180. Article Stable URL: <http://www.jstor.org/stable/2346945>. 563
- Chang, Winston. 2012. *R Graphics Cookbook - Practical Recipes for Visualizing Data*. 1st edn. O'Reilly Media. Online resource: <http://shop.oreilly.com/product/0636920023135.do>. 825
- Chase, P.J. 1970. Permutations of a set with repetitions. *Communications of the ACM*, 13, 368–369. 744

- Chattamvelli, R., & Jones, M. C. 1995. Recurrence relations for noncentral density, distribution functions and inverse moments. *Journal of Statistical Computation and Simulation*, **52**(3), 289–299. [700](#), [707](#)
- Cheney, Ward, & Kincaid, David. 2008. *Numerical Mathematics and Computing*. 6th edn. Thomson Brooks/Cole. [543](#)
- Cheng, Peter H., & Meng, Cliff Y.K. 1995. A new formula for tail probabilities of dunnett's t with unequal sample sizes. *Communications in Statistics - Theory and Methods*, **24**(2), 523–532. [667](#)
- Chernick, Michael R. 2008. *Bootstrap methods : a guide for practitioners and researchers*. 2nd edn. John Wiley & Sons. [543](#)
- Chiani, M. 2012. Distribution of the largest eigenvalue for real Wishart and Gaussian random matrices and a simple approximation for the Tracy-Widom distribution. *ArXiv e-prints*, Sept. Available at <http://adsabs.harvard.edu/abs/2012arXiv1209.3394C>. [710](#)
- Chiani, M. 2014. Distribution of the largest root of a matrix for Roy's test in multivariate analysis of variance. *ArXiv e-prints*, Jan. Available at <http://adsabs.harvard.edu/abs/2014arXiv1401.3987C>. [710](#), [711](#)
- Chou, Youn-Min. 1985. New representations for the doubly noncentral  $F$ -distribution and derived distribution. *Communications in Statistics - Theory and Methods*, **14**(3), 527–534. [674](#), [697](#), [704](#)
- Cohen, J. D. 1988. Noncentral Chi-Square: Some Observations on Recurrence. *The American Statistician*, **42**, 120–122. [677](#)
- Conlon, M., & Thomas, R. G. 1993. Algorithm AS 280: The Power Function for Fisher's Exact Test. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, **42**(1), 258–260. Article Stable URL: <http://www.jstor.org/stable/2347431>. [563](#)
- Conn, A. R., Gould, N. I. M., & Toint, Ph. L. 1991. A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds. *SIAM J. Numer. Anal.*, **28**(2), 545–572. [509](#)
- Conover, W. J., & Iman, Ronald L. 1981. Rank Transformations as a Bridge Between Parametric and Nonparametric Statistics. *The American Statistician*, **35**(3), 124–129. [748](#)
- Corless, R. M., Gonnet, G. H., Hare, D. E. G., Jeffrey, D. J., & Knuth, D. E. 1996. On the Lambert W Function. *Pages 329–359 of: Advances in Computational Mathematics*. [182](#)
- Cuyt, A.A.M., Verdonk, B., Becuve, S., & Kuterna, P. 2001. A remarkable example of catastrophic cancellation unraveled. *Computing*, **66**(3), 309–320. [6](#)
- Cuyt, A.A.M., Petersen, V., Verdonk, B., Waadeland, H., & Jones, W.B. 2008. *Handbook of Continued Fractions for Special Functions*. Springer. [517](#)
- Daniels, H. E. 1954. Saddlepoint Approximations in Statistics. *Annals of Mathematical Statistics*, **25**, 631–650. [633](#)
- Daniels, H. E. 1987. Tail Probability Approximations. *International Statistical Review*, **55**, 37–48. [633](#)

- David, H. A., Lachenbruch, P. A., & Brandis, H. P. 1972. The Power Function of Range and Studentized Range Tests in Normal Samples. *Biometrika*, **59**(1), 161–168. Article Stable URL: <http://www.jstor.org/stable/2334627>. 668
- David, S. T., Kendall, M. G., & Stuart, A. 1951. Some Questions of Distribution in the Theory of Rank Correlation. *Biometrika*, **38**(1/2), 131–140. Article Stable URL: <http://www.jstor.org/stable/2332322>. 742, 743
- Davies, Ph., & Higham, N. J. 2003. A Schur-Parlett algorithm for computing matrix functions. *SIAM J. Matrix Anal. Appl.*, **25**, 464–485. 479
- Davis, A. W. 1968. A System of Linear Differential Equations for the Distribution of Hotelling's Generalized  $T_0^2$ . *The Annals of Mathematical Statistics*, **39**(3), 815–832. Available at <http://dx.doi.org/10.1214/aoms/1177698313>. 631, 641, 719, 722
- Davis, A. W. 1970a. On the Null Distribution of the Sum of the Roots of a Multivariate Beta Distribution. *The Annals of Mathematical Statistics*, **41**(5), 1557–1562. 631
- Davis, A. W. 1970b. Further applications of a differential equation for Hotelling's generalized  $T_0^2$ . *Annals of the Institute of Statistical Mathematics*, **22**(1), 77–87. Available at <http://dx.doi.org/10.1007/BF02506325>. 631, 642, 719, 722
- Davis, A. W. 1971. Percentile approximations for a class of likelihood ratio criteria. *Biometrika*, **58**, 349–356. 635
- Dembo, R. S., & Steihaug, T. 1982. Truncated Newton algorithms for large-scale optimization. *Math. Programming*, **26**, 190–212. 508
- Di Bucchianico, A., & van de Wiel, M.A. 2005. Exact null distributions of distribution-free quadratic t-sample statistics. *Journal of Statistical Planning and Inference*, **127**(1–2), 1–21. 745
- DiDonato, A.R., & Morris, A.H. 1986. Computation of the Incomplete Gamma Function Ratios and their Inverse. *ACM TOMS*, **12**, 377–393. Fortran source: ACM TOMS 13 (1987) pp. 318–319; available from <http://netlib.org/toms/654>. 166
- DiDonato, Armido R., & Morris, Jr., Alfred H. 1992. Algorithm 708: Significant digit computation of the incomplete beta function ratios. *ACM Trans. Math. Softw.*, **18**(3), 360–373. 523
- Ding, C. G. 1992. Algorithm AS 275: Computing the non-central  $\chi^2$  distribution function. *Applied Statistics*, **41**, 478–482. 606, 674
- Ding, C. G. 1996. On the computation of the distribution of the square of the sample multiple correlation coefficient. *Computational Statistics & Data Analysis*, **22**(4), 345 – 350. 657
- Divine, George, Kapke, Alissa, Havstad, Suzanne, & Joseph, Christine L. M. 2010. Exemplary data set sample size calculation for Wilcoxon–Mann–Whitney tests. *Statistics in Medicine*, **29**(1), 108–115. 729
- Dongarra, J.J., DuCroz, J., Hammarling, S., & Hanson, R. 1988. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, **14**, 1–32. BLAS Homepage: <http://www.netlib.org/blas/>. 365

- Dongarra, J.J., Duff, I., DuCroz, J., & Hammarling, S. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, **16**, 1–28. BLAS Homepage: <http://www.netlib.org/blas/>. 365
- Dormand, J.R., & Prince, P.J. 1980. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, **6**(1), 19 – 26. 618
- Dubinov, A.E., & Dubinova, A.A. 2008. Exact integral-free expressions for the integral Debye functions. *Technical Physics Letters*, **34**(12), 999–1001. Available at <http://link.springer.com/article/10.1134/S106378500812002X#page-1>. 288
- Dunnett, Charles W. 1955. A Multiple Comparison Procedure for Comparing Several Treatments with a Control. *Journal of the American Statistical Association*, **50**, 1096–1121. 667
- Eastlake, D., Crocker, S., & Schiller, J. 1994. Randomness Recommendations for Security. *Network Working Group, RFC 1750*. 511
- Edelman, Alan, & Murakami, H. 1995. Polynomial Roots from Companion Matrix Eigenvalues. *Math. Comput.*, **64**(210), 763–776. Available as <http://www.ams.org/journals/mcom/1995-64-210/S0025-5718-1995-1262279-2/S0025-5718-1995-1262279-2.pdf>. 489
- Enge, Andreas, Gastineau, Mickaël, Théveny, Philippe, & Zimmermann, Paul. 2012 (July). *mpc — A library for multiprecision complex arithmetic with exact rounding*. 1.0 edn. INRIA. <http://mpc.multiprecision.org/>. The MPC manual is available from <http://www.multiprecision.org/mpc/download/mpc-1.0.1.pdf>. 3
- Fan, Chunpeng, & Zhang, Donghui. 2012. A Note on Power and Sample Size Calculations for the Kruskal-Wallis Test for Ordered Categorical Data. *Journal of Biopharmaceutical Statistics*, **22**(6), 1162–1173. PMID: 23075015. 745
- Fan, Chunpeng, Zhang, Donghui, & Zhang, Cun-Hui. 2011. On Sample Size of the Kruskal-Wallis Test with Application to a Mouse Peritoneal Cavity Study. *Biometrics*, **67**(1), 213–224. 745
- Fellingham, S. A., & Stoker, D. J. 1964. An Approximation for the Exact Distribution of the Wilcoxon Test for Symmetry. *Journal of the American Statistical Association*, **59**(307), 899–905. Article Stable URL: <http://www.jstor.org/stable/2283109>. 735
- Feuersänger, Christian. 2014. *Manual for Package pgfplots: 2D/3D Plots in LATEX, Version 1.11*. Available at <http://sourceforge.net/projects/pgfplots>. 7
- Fisher, Sir Ronald A., & Cornish, E. A. 1960. The Percentile Points of Distributions Having Known Cumulants. *Technometrics*, **2**, 209–225. 632
- Forrey, Robert C. 1997. Computing the Hypergeometric Function. *Journal of Computational Physics*, **137**(1), 79 – 100. PDF document and Fortran code available from: <http://physics.bk.psu.edu/pub/papers/hyper.pdf>, and <http://physics.bk.psu.edu/codes.html>. 227
- Fousse, Laurent, Hanrot, Guillaume, Lefèvre, Vincent, Péllissier, Patrick, & Zimmermann, Paul. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, **33**(2), 13:1–13:15. Online resource: <http://www.mpfr.org/>. The MPFR manual is available from <http://www.mpfr.org/mpfr-current/mpfr.pdf>, and a description of the algorithms from <http://www.mpfr.org/algorithms.pdf>. 3

- Fujikoshi, Y. 1973. Asymptotic formulas for the non-null distributions of three statistics for multivariate linear hypothesis. *Annals of the Institute of Statistical Mathematics*, **25**, 423–436. 714
- Gablonsky, J. M., & Kelley, C. T. 2001. A locally-biased form of the DIRECT algorithm. *J. Global Optimization*, **21**(1), 27–37. 501
- Gansky, Stuart, Huynh, Quang, & Kapila, Yvonne L. 2001. Power Analysis for Friedman's Test with an Application to Molecular Biology. *Proceedings of the Annual Meeting of the American Statistical Association, August 5-9, Abstract #301336*. Article Stable URL: <http://www.amstat.org/sections/SRMS/Proceedings/y2001/Proceed/00450.pdf>. 747
- Genz, A., & Bretz, F. 2002. Methods for the Computation of Multivariate t-Probabilities. *Journal of Computational and Graphical Statistics*, **11**(1), 950–971. Online resource: <http://www.math.wsu.edu/faculty/genz/homepage>. 669
- Genz, A., & Bretz, F. 2009. *Computation of Multivariate Normal and t Probabilities, Lecture Notes in Statistics* 195. Springer-Verlag, New York. 669
- Ghazi, Kaveh R., Lefèvre, Vincent, Théveny, Philippe, & Zimmermann, Paul. 2010. Why and How to Use Arbitrary Precision. *Computing in Science and Engineering*, **12**(3), 62–65. Preprint available at <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.212.6321&rep=rep1&type=pdf>. 6
- Gil, Amparo, Segura, Javier, & Temme, Nico M. 2007. *Numerical methods for special functions*. SIAM. 517
- Gil, Amparo, Segura, Javier, & Temme, NicoM. 2011. Basic Methods for Computing Special Functions. Pages 67–121 of: Simos, Theodore E. (ed), *Recent Advances in Computational and Applied Mathematics*. Springer Netherlands. Preprint available from [http://www.researchgate.net/publication/229032140\\_Basic\\_Methods\\_for\\_Computing\\_Special\\_Functions/file/d912f5093e6adecd38.pdf](http://www.researchgate.net/publication/229032140_Basic_Methods_for_Computing_Special_Functions/file/d912f5093e6adecd38.pdf). 517
- Gleser, L. J. 1976. A canonical representation for the noncentral Wishart distribution useful for simulation. *Journal of the American Statistical Association*, 690–695. 557, 679
- Goldberg, David. 1991. What Every Computer Scientist Should Know About Floating Point Arithmetic. *ACM Computing Surveys*, **23**(1), 5–48. Extended and edited reprint from <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.6768>. 5
- Golhar, M. B. 1972. The errors of first and second kinds in Welch-Aspin's solution of the Behrens-Fisher problem. *Journal of Statistical Computation and Simulation*, **1**(3), 209–224. 576
- Golub, Gene H., & Van Loan, Charles F. 1996. *Matrix Computations*. 3rd edn. The John Hopkins University Press. 366, 387, 428, 429, 447, 458, 463
- Good, Ph. I. 2005. *Permutation, Parametric, and Bootstrap Tests of Hypotheses (Springer Series in Statistics)*. Springer-Verlag. 734
- Govindarajulu, Z. 2007. *Nonparametric Inference*. World Scientific Publishing Co. Pte. Ltd., Singapore. 734

- Granlund, Torbjörn, & the GMP development team. 2013. *GNU MP: The GNU Multiple Precision Arithmetic Library*. 5.1.2 edn. Online resource: <http://gmplib.org/>. The GMP manual is available from <http://gmplib.org/gmp-man-5.1.2.pdf>. 3, 336
- Grantham, Jon. 2001. Frobenius pseudoprimes. *Mathematics of Computation*, **70**, 873–891. Available at <http://www.ams.org/journals/mcom/2001-70-234/S0025-5718-00-01197-2/home.html>. 351
- Gudmundsson, S. 1998. *Parallel Global Optimization*. M.Phil. thesis, Technical University of Denmark. 503
- Guennebaud, Gaël, Jacob, Benoît, et al. 2010. *Eigen v3, a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms*. Online resource: <http://eigen.tuxfamily.org>. 3, 365
- Guenther, William C. 1974. Sample Size Formulas for Some Binomial Type Problems. *Technometrics*, **16**, 465–467. 564
- Guenther, William C. 1977. Desk Calculation of Probabilities for the Distribution of the Sample Correlation Coefficient. *The American Statistician*, **31**, 45–48. 649
- Gupta, R. D., & Richards, D. 1979. Calculation of zonal polynomials of 3 x 3 positive definite symmetric matrices. *Annals of the Institute of Statistical Mathematics*, **31**, 207–213. 625
- Gurland, J. 1968. A relatively simple form of the distribution of the multiple correlation coefficient. *Journal of the Royal Statistical Society, Series B, Methodological*, **30**, 276–283. 657
- Gurland, J., & Milton, R. 1970. Further Consideration of the Distribution of the Multiple Correlation Coefficient. *Journal of the Royal Statistical Society. Series B (Methodological)*, **32**(3), 381–394. Article Stable URL: <http://www.jstor.org/stable/2984364>. 658
- Hairer, E., & Wanner, G. 2010. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. 2nd edn. Springer, Berlin. 615
- Hairer, E., Wanner, G., & Lubich, C. 2006. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd edn. Springer-Verlag GmbH. 615
- Hairer, E., Nørsett, S. P., & Wanner, G. 2009. *Solving Ordinary Differential Equations I: Nonstiff Problems*. 2nd edn. Springer, Berlin. 615
- Hammer, R., Hocks, M., Ratz, D., & Kulisch, U. 1995. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems Theory, Algorithms, and Programs*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. Online resource: <http://link.springer.com/book/10.1007/978-3-642-79651-7/page/1>. 360
- Hardy D.W., Richman F., Walker C.L. 2009. *Applied Algebra: Codes, Ciphers and Discrete Algorithms (Discrete Mathematics and Its Applications)*. 2nd edn. Chapman and Hall/CRC. 337
- Harkness, W. L., & Katz, L. 1964. Comparison of the Power Functions for the Test of Independence in 2 x 2 Contingency Tables. *The Annals of Mathematical Statistics*, **35**(3), 1115–1127. Article Stable URL: <http://www.jstor.org/stable/2238241>. 563

- Hart, W. B. 2010. Fast Library for Number Theory: An Introduction. *Pages 88–91 of: Proceedings of the Third International Congress on Mathematical Software*. ICMS'10. Berlin, Heidelberg: Springer-Verlag. <http://flintlib.org>. 3
- Harter, H. L. 1960. Tables of Range and Studentized Range. *The Annals of Mathematical Statistics*, **31**(4), 1122–1147. Article Stable URL: <http://www.jstor.org/stable/2237810>. 668
- Hayes, Brian. 2003. A Lucid Interval. *American Scientist*, **91**(6), 484–488. Available at: <http://www.cs.utep.edu/interval-comp/hayes.pdf>. 360
- Hellekalek, P. 1995. Inversive pseudorandom number generators: concepts, results and links. *Pages 255–262 of: Alexopoulos, C., Kang, K., Lilegdon, W.R., , & Goldsman, D. (eds), Proceedings of the 1995 Winter Simulation Conference*. Available from <ftp://random.mat.sbg.ac.at/pub/data/wsc95.ps>. 513
- Hendrix, E. M. T., Ortigosa, P. M., , & García, I. 2001. On success rates for controlled random search. *J. Global Optim.* **21**, 239–263. 502
- Higham, N. J. 1987. Computing real square roots of a real matrix. *Linear Algebra Appl.*, **88/89**, 405–430. 471
- Higham, N. J. 2005. The scaling and squaring method for the matrix exponential revisited. *SIAM J. Matrix Anal. Appl.*, **26**, 1179–1193. 473
- Higham, N. J. 2008. *Functions of Matrices: Theory and Computation*. 1st edn. Society for Industrial and Applied Mathematics (SIAM). ISBN 978-0-898716-46-7. 475
- Higham, N. J., & Lin, L. 2011. A Schur-Padé algorithm for fractional powers of a matrix. *SIAM J. Matrix Anal. Appl.*, **32**(3), 1056–1078. 476
- Higham, Nicholas J. 2002. *Accuracy and Stability of Numerical Algorithms*. 1st edn. SIAM. Online resource: <http://www.maths.manchester.ac.uk/~higham/asna/>. 5
- Higham, Nicholas J. 2009. *Accuracy and Stability of Numerical Algorithms - Presentation given at 3rd Many-core and ReconiñAgurable Supercomputing Network Workshop QueenâŽs University, Belfast, January 15-16, 2009*. Available as <http://cpc.cs.qub.ac.uk/MRSN/higham.pdf>. 5
- Hirotsu, C. 1979. An F Approximation and its Application. *Biometrika*, **66**(3), 577–584. Article Stable URL: <http://www.jstor.org/stable/2335178>. 672
- Hochberg, Y., & Tamhane, A. C. 1987. *Multiple comparison procedures*. New York, NY, USA: John Wiley & Sons, Inc. 668
- Hofschuster, W. 2000. *Zur Berechnung von Funktionswerteinschließungen bei speziellen Funktionen der mathematischen Physik*. Ph.D. thesis, Universität Karlsruhe. Available online at: <http://digibib.ubka.uni-karlsruhe.de/volltexte/documents/1493>. 360
- Hofschuster, W., & Krämer, W. 2004. C-XSC 2.0: A C++ Library for Extended Scientific Computing. *Pages 15–35 of: Alt, R., Frommer, A., Kearfott, R.B., & Luther, W. (eds), Numerical Software with Result Verification, Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg. Online resource: <http://www2.math.uni-wuppertal.de/~xsc/xsc-sprachen.html>.

A preprint is available from [http://www2.math.uni-wuppertal.de/~xsc/preprints/prep\\_03\\_5.pdf](http://www2.math.uni-wuppertal.de/~xsc/preprints/prep_03_5.pdf). 6, 360

Holoborodko, P. 2008-2012. *MPFR C++: A high-performance C++ interface for the MPFR library*. Online resource: <http://www.holoborodko.com/pavel/mpfr/>. 3

Hotelling, Harold. 1953. New Light on the Correlation Coefficient and Its Transforms. *Journal of the Royal Statistical Society, Series B, Methodological*, **15**, 193–225. 648, 649, 650, 653

Iman, R.L., Quade, D., & Alexander, D.A. 1975. Exact probability levels for the Kruskal Wallis test. *Selected Tables in Mathematical Statistics*, **3**, 329–384. 742, 743

James, A. T. 1968. Calculation of Zonal Polynomial Coefficients by Use of the Laplace-Beltrami Operator. *The Annals of Mathematical Statistics*, **39**(5), 1711–1718. 625

James, Alan T. 1964. Distributions of Matrix Variates and Latent Roots Derived from Normal Samples. *The Annals of Mathematical Statistics*, **35**(2), 475–501. 625

James, F. 1994. RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher. *Computer Physics Communications*, **79**, 111–114. 514

Jensen, J.L. 1992. The modified signed likelihood statistic and saddlepoint approximations. *Annals of Mathematical Statistics*, **79**, 693–703. 633

Johansson, F. 2013. *ARB: a C library for arbitrary-precision floating-point ball arithmetic*. Version 2.0.0, <http://fredrikj.net/arb/>. 3

Johnson, N. L., Kotz, S., & Balakrishnan, N. 1994.. *Continuous Univariate Distributions, Volume 1*. 2nd edn. Wiley & Sons, Inc., New York-London-Sydney-Toronto. 543

Johnson, N. L., Kotz, S., & Balakrishnan, N. 1995.. *Continuous Univariate Distributions, Volume 2*. 2nd edn. Wiley & Sons, Inc., New York-London-Sydney-Toronto. 543, 652, 660

Johnson, Steven G. 2012. *NLOPT v2.3: a free/open-source library for nonlinear optimization*. Online resource: <http://ab-initio.mit.edu/wiki/index.php/NLopt>. 500

Johnstone, I. M., & Nadler, B. 2013. Roy's Largest Root Test Under Rank-One Alternatives. *ArXiv e-prints*, Oct. Available at <http://adsabs.harvard.edu/abs/2013arXiv1310.6581J>. 710

Jones, D. R., Perttunen, C. D., & Stuckmann, B. E. 1993. Lipschitzian optimization without the lipschitz constant. *J. Optimization Theory and Applications*, **79**, 157. 501

Joye, M., & Quisquater, J.-J. 1996. Efficient computation of full Lucas sequences. *Electronics Letters*, **32**(6), 537–538. Available at <http://joye.site88.net/papers/JQ96lucas.pdf>. 354

Kaelo, P., & Ali, M. M. 2006. Some variants of the controlled random search algorithm for global optimization. *J. Optim. Theory Appl.*, **130**(2), 27–37. 502

Kahan, W. 1997. *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*. Tech. rept. Elect. Eng. & Computer Science, University of California, Berkeley, CA. Available as <http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>. 328, 329

Kammler, D. W. 2008. *A First Course in Fourier Analysis*. 1st edn. Cambridge University Press. 491

- Klatte, R., Kulisch, U., Neaga, M., Ratz, D., & Ullrich, Ch. 1991. *PASCAL-XSC Language Reference with Examples*. Springer Berlin Heidelberg. Available at: <http://www2.math.uni-wuppertal.de/~xsc/literatur/PXSCENGL.pdf>. 360
- Klotz, J. 1963. Small sample power and efficiency for the one sample Wilcoxon and normal scores tests. *Ann. Stat. Math.*, **34**(2), 624–632. Available at <http://projecteuclid.org/euclid.aoms/1177704175>. 736
- Knüsel, L., & Michalk, J. 1987. Asymptotic expansion of the power function of the two-sample binomial test with and without randomization. *Metrika*, **34**(1), 31–44. Available at: <http://rd.springer.com/article/10.1007%2FBF02613128#>. 563
- Knuth, Donald E. 1981. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 2nd edn. Addison-Wesley Professional. Online resource: <http://www-cs-faculty.stanford.edu/~knuth/taocp.html>. 513
- Knuth, Donald E. 1997. *Art of Computer Programming, Volume 1: Fundamental Algorithms, Volume 2: Seminumerical Algorithms*. 3rd edn. Addison-Wesley Professional. Online resource: <http://www-cs-faculty.stanford.edu/~knuth/taocp.html>. 512
- Kocherlakota, K., & Kocherlakota, S. 1991. On the doubly noncentral t distribution. *Communications in Statistics - Simulation and Computation*, **20**(1), 23–31. 691, 692
- Koev, P., & Edelman, A. 2006. The efficient evaluation of the hypergeometric function of a matrix argument. *Mathematics of Computation*, **75**, 833–846. Article Stable URL: <http://math.mit.edu/~plamen/files/hyper.pdf>, Software available at <http://math.mit.edu/~plamen/software/mhgref.html>. 626, 627
- Kolassa, John E. 1995. A comparison of size and power calculations for the wilcoxon statistic for ordered categorical data. *Statistics in Medicine*, **14**(14), 1577–1581. Available at <http://onlinelibrary.wiley.com/doi/10.1002/sim.4780141408/abstract>. 733
- Koning, A. J., & Does, R. J. M. M. 1988. Statistical Algorithms: Algorithm AS 234: Approximating the Percentage Points of Simple Linear Rank Statistics with Cornish-Fisher Expansions. *Applied Statistics*, **37**(2), 278–284. 742, 743
- Koziol, J. A. 1982. Orthogonal Components of the Kruskal-Wallis and Related K-sample Linear Rank Statistics. *Biometrical Journal*, **24**(5), 445–455. 745, 747
- Kraft, D. 1988. *A software package for sequential quadratic programming*. Technical Report DFVLR-FB 88-28. Institut für Dynamik der Flugsysteme, Oberpfaffenhofen. 507
- Kraft, D. 1994. Algorithm 733: TOMP-Fortran modules for optimal control calculations. *ACM Transactions on Mathematical Software*, **20**(3), 262–281. 507
- Krah, Stefan. 2012. *mpdecimal: a package for correctly-rounded arbitrary precision decimal floating point arithmetic*. 2.3 edn. Online resource: <http://www.byttereef.org/mpdecimal/index.html>. 3, 26
- Krämer, W., Kulisch, U., & Lohner, R. 1994. *Numerical Toolbox for Verified Computing II: Advanced Numerical Problems*. Draft Manuscript. Preprint (1994) available from <http://www2.math.uni-wuppertal.de/~xsc/literatur/tb2.pdf>. 360

- Krämer, W., Kulisch, U., & Lohner, R. 2006. *Numerical Toolbox for Verified Computing II: Advanced Numerical Problems*. SpringerVerlag. Pascal Version Preprint (1994) available from <http://www2.math.uni-wuppertal.de/~xsc/literatur/tb2.pdf>. 360
- Krämer, W., Zimmer, M., & Hofschuster, W. 2012. Using C-XSC for High Performance Verified Computing. *Pages 168–178 of: Applied Parallel and Scientific Computing*. Lecture Notes in Computer Science, vol. 7134. Springer Berlin Heidelberg. 360
- Kreutzer, W. 1986. *System Simulation: Programming Styles and Languages (International Computer Science Series)*. 1st edn. Addison-Wesley. 513
- Kucherenko, S., & Sytsko, Y. 2005. Application of deterministic low-discrepancy sequences in global optimization. *Computational Optimization and Applications*, **30**, 297–318. 502
- Kulp, R. W., & Nagarsenker, B. N. 1984. An Asymptotic Expansion of the Nonnull Distribution of Wilks Criterion for Testing the Multivariate Linear Hypothesis. *The Annals of Statistics*, **12**(4), 1576–1583. Available at <http://projecteuclid.org/euclid-aos/1176346816>. 712
- Lai, Yongzeng. 2006. Efficient computations of multivariate normal distributions with applications to finance. *Pages 522–527 of: Proceedings of the 17th IASTED international conference on Modelling and simulation*. MS'06. Anaheim, CA, USA: ACTA Press. 666
- Lange, Kenneth. 2010. *Numerical Analysis for Statisticians*. 2nd edn. Springer Science+Business Media. 543
- Lawson, C., Hanson, R., Kincaid, D., & Krogh, F. 1979. Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, **5**, 308–325. BLAS Homepage: <http://www.netlib.org/blas/>. 365
- L'Ecuyer, P. 1988. Efficient and Portable Combined Random Number Generators. *Communications of the ACM*, **31**(6), 742–749. 513
- L'Ecuyer, P. 1996. Maximally Equidistributed Combined Tausworthe Generators. *Mathematics of Computation*, **65**(213), 203–213. 513
- Lee, Y.-S., & Lin, T.-K. 1992. Algorithm AS 269: High Order Cornish-Fisher Expansion. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, **41**(1), 233–240. Article Stable URL: <http://www.jstor.org/stable/2347649>. 631, 632, 742, 743
- Lee, Yoong-Sin. 1971a. Asymptotic Formulae for the Distribution of a Multivariate Test Statistic: Power Comparisons of Certain Multivariate Tests. *Biometrika*, **58**, 647–651. 712
- Lee, Yoong-Sin. 1971b. Distribution of the Canonical Correlations and Asymptotic Expansions for Distributions of Certain Independent Test Statistics. *The Annals of Mathematical Statistics*, **42**, 526–537. Available at <http://projecteuclid.org/euclid.aoms/1177693403>. 712, 715, 718
- Lee, Yoong-Sin. 1971c. Some Results on the Sampling Distribution of the Multiple Correlation Coefficient. *Journal of the Royal Statistical Society, Series B, Methodological*, **33**, 117–130. 657, 658, 660, 661
- Lee, Yoong-Sin. 1972a. Some Results on the Distribution of Wilks's Likelihood-ratio Criterion. *Biometrika*, **59**, 649–664. 712

- Lee, Yoong-Sin. 1972b. Tables of Upper Percentage Points of the Multiple Correlation Coefficient. *Biometrika*, **59**, 175–189. [658](#)
- Leimkuhler, B., & Reich, S. 2005. *Simulating Hamiltonian Dynamics*. 1st edn. Cambridge University Press. [615](#)
- Lewis, P.A., Goodman, A.S., & Miller, J.M. 1969. A pseudo-random number generator for the System/360. *IBM Systems Journal*, **8**(2), 136–146. [513](#)
- Ling, Robert F., & Pratt, John W. 1984. The accuracy of Peizer approximations to the hypergeometric distribution, with comparisons to some other approximations. *JASA. Journal of the American Statistical Association*, **79**, 49–60. [563](#)
- Liu, D. C., & Nocedal, J. 1989. On the limited memory BFGS method for large scale optimization. *Math. Programming*, **45**, 503–528. [508](#)
- Luescher, M. 1994. A portable high-quality random number generator for lattice field theory calculations. *Computer Physics Communications*, **79**, 100–110. [514](#)
- Lugannani, Robert, & Rice, Stephen. 1980. Saddle Point Approximation for the Distribution of the Sum of Independent Random Variables. *Advances in Applied Probability*, **12**(2), 475–490. [633](#)
- Luschny, Peter. 2012. *Approximation Formulas for the Factorial Function*. Web. Accessed June 3, 2012. Available at <http://www.luschny.de/math/factorial/approx/SimpleCases.html>. [517](#)
- Mack, G.A., & Skillings, J.H. 1980. A Friedman type rank test for main effects in a two factor ANOVA. *JASA*, **75**, 947–951. [747](#)
- Maddock, John, & Kormanyos, Christopher. 2013 (June). *Boost Multiprecision Library*. <http://www.boost.org/libs/multiprecision/>. Documentation available as [http://www.boost.org/doc/libs/1\\_54\\_0/libs/multiprecision/doc/html/index.html](http://www.boost.org/doc/libs/1_54_0/libs/multiprecision/doc/html/index.html). [517](#)
- Madsen, K., Zertchaninov, S., & Zilinskas, A. 1998. *Global Optimization using Branch-and-Bound*. Unpublished Manuscript. [503](#)
- Mathai, A.M., Saxena, R. K., & Haubold, H. J. 2010. *The H-Function: Theory and Applications*. 1 edn. Springer New York Dordrecht Heidelberg London. [643](#)
- Matsumoto, M., & Nishimura, T. 1998. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation: Special Issue on Uniform Random Number Generation*, **8**(1), 3–30. Available as <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf>. [514](#)
- Maurer, Jens, & Watanabe, Steven. 2013 (June). *Boost Random Number Library*. <http://www.boost.org/libs/random/>. Documentation available as [http://www.boost.org/doc/libs/1\\_54\\_0/doc/html/boost\\_random.html](http://www.boost.org/doc/libs/1_54_0/doc/html/boost_random.html). [3, 511](#)
- McKinney, Wes. 2012. *Python for Data Analysis*. O'Reilly Media. Online resource: <http://shop.oreilly.com/product/0636920023784.do>. [811](#)
- McLachlan, R. I. 1995. On the numerical integration of ordinary differential equations by symmetric composition methods. *SIAM J. Sci. Comput.*, **16**(1), 151–168. [615](#)

- McLaren, M. L. 1976. Algorithm AS 94: Coefficients of the Zonal Polynomials. *Applied Statistics*, **25**(1), 82–87. [625](#)
- Mehrotra, Devan V., Lu, Xiaomin, & Li, Xiaoming. 2010. Rank-Based Analyses of Stratified Experiments: Alternatives to the van Elteren Test. *The American Statistician*, **64**(2), 121–130. [729](#)
- Milton, R. C. 1970. *Rank Order Probabilities. Two-Sample Normal Shift Alternatives*. Wiley & Sons, Inc., New York-London-Sydney-Toronto. [669](#), [730](#), [736](#), [744](#)
- Moler, C., & Stewart, G. 1973. An Algorithm for Generalized Matrix Eigenvalue Problems. *SIAM J. Numer. Anal.*, **10**(2), 241–256. [463](#)
- Monahan, John F. 2011. *Numerical methods of statistics*. 2nd edn. Cambridge University Press. [543](#)
- Moore R. E., Cloud M. J., Kearfott R. B. 2009. *Introduction to Interval Analysis*. Philadelphia: Society for Industrial and Applied Mathematics. [360](#)
- Moré, J. J., Garbow, B. S., & Hillstrom, K. E. 1980. *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74, Argonne, Ill. Available as <http://www.mcs.anl.gov/~more/ANL8074a.pdf> (Chapter 1-3) and <http://www.mcs.anl.gov/~more/ANL8074b.pdf> (Chapter 4). [497](#)
- Mudholkar, G. S., Chaubey, Y. P., & Lin, C.-C. 1975. Some Approximations for the Noncentral-F Distribution. *Technometrics*, **18**(3), 351–358. [700](#)
- Mudholkar, G. S., Chaubey, Y. P., & Lin, C.-C. 1976. Approximations for the doubly noncentral-F distribution. *Communications in Statistics - Theory and Methods*, **5**(1), 49–63. [704](#)
- Muirhead, R. J. 1982. *Aspects of Multivariate Statistical Theory*. John Wiley and Sons. [625](#), [660](#), [682](#), [717](#), [718](#), [725](#), [727](#)
- Muller, K. E., LaVange, L. M., Ramey, S L., & Ramey, C. T. 1992. Power Calculations for General Linear Multivariate Models Including Repeated Measures Applications. *J Am Stat Assoc.*, **87**, 1209–1226. [712](#)
- Muller, Keith E. 2001. Computing the confluent hypergeometric function,  $M(a,b,x)$ . *Numerische Mathematik*, **90**(1), 179–196. [227](#)
- Murakami, H., & Kamakura, T. 2009. A Saddlepoint Approximation to a Nonparametric Test for Ordered Alternatives. *Journal of the Japan Statistical Society*, **39**(2), 143–153. [741](#)
- Nagarsenker, B. N., & Suniaga, J. 1983. Distributions of a Class of Statistics Useful in Multivariate Analysis. *Journal of the American Statistical Association*, **78**(382), 472–475. [644](#)
- Nagarsenker, P. B. 1984. On Bartlett's Test for Homogeneity of Variances. *Biometrika*, **71**(2), 405–407. Article Stable URL: <http://www.jstor.org/stable/2336260>. [641](#)
- Narula, S.C. 1978. Orthogonal polynomial regression for unequal spacing and frequencies. *Journal of Quality Technology*, **10**, 170–179. [667](#)
- Nelder, J. A., & Mead, R. 1965. A simplex method for function minimization. *The Computer Journal*, **7**, 308–313. [505](#)

- Neuhäuser, Markus, & Hothorn, Ludwig A. 1998. An Analogue of Jonckheere's Trend Test for Parametric and Dichotomous Data. *Biometrical Journal*, **40**(1), 11–19. [741](#)
- Nocedal, J. 1980. Updating quasi-Newton matrices with limited storage. *Math. Comput.*, **35**, 773–782. [508](#)
- Noether, G.E. 1967. *Elements of nonparametric statistics*. SIAM series in applied mathematics. Wiley. [734](#)
- Noether, Gottfried E. 1987. Sample Size Determination for Some Common Nonparametric Tests. *Journal of the American Statistical Association*, **82**, 645–647. [737](#)
- Noether, Gottfried E., & Dueker, Marilyn. 1990. *Introduction to Statistics: the Nonparametric Way*. Springer-Verlag Inc. [734](#)
- O'Brien, R. G., & Shieh, G. 1992. *Pragmatic, Unifying Algorithm Gives Power Probabilities for Common F Tests of the Multivariate General Linear Hypothesis*. Poster presented at the American Statistical Association Meetings, Boston, Statistical Computing Section. Also, paper in review, downloadable in PDF form from <http://www.bio.ri.ccf.org/UnifyPow>. [712](#)
- Odeh, Robert E. 1986. *Confidence limits on the correlation coefficient*. In: *Selected Tables in Mathematical Statistics, Vol. 10*. American Mathematical Society. [647](#), [648](#), [650](#)
- Odeh, Robert E., & Owen, D. B. 1980. *Tables for Normal Tolerance Limits, Sampling Plans, and Screening*. Marcel Dekker Inc. [759](#)
- Olver, F.W.J., Lozier, D.W., Boisvert, R.F., & Clark, C.W. 2010. *NIST Handbook of Mathematical Functions*. 1 edn. Cambridge. Online resource: NIST Digital Library of Mathematical Functions <http://dlmf.nist.gov/>. [126](#), [270](#), [517](#), [537](#), [625](#), [626](#), [627](#), [628](#)
- Ong, S. H., & Lee, P. A. 1979. The Non-central Negative Binomial Distribution. *Biometrical Journal. Journal of Mathematical Methods in Biosciences. [Continues: Biometrische Zeitschrift. Zeitschrift für mathematische Methoden in den Biowissenschaften]*, **21**, 611–628. [567](#)
- Owen, D. B. 1956. Tables for computing bivariate normal probabilities. *Ann. Math. Statist.*, **27**, 1075–1090. Available as [http://projecteuclid.org/DPubS/Repository/1.0/Disseminate?view=body&id=pdf\\_1&handle=euclid.aoms/1177728074](http://projecteuclid.org/DPubS/Repository/1.0/Disseminate?view=body&id=pdf_1&handle=euclid.aoms/1177728074). [614](#), [665](#), [666](#)
- Owen, D. B. 1965. A Special Case of a Bivariate Non-central  $t$ -distribution. *Biometrika*, **52**, 437–446. [672](#)
- Owen, D. B. 1968. A Survey of Properties and Applications of the Noncentral  $t$ -distribution. *Technometrics*, **10**, 445–478. [685](#)
- Owen, D. B., & Steck, G. P. 1962. Moments of Order Statistics from the Equicorrelated Multivariate Normal Distribution. *The Annals of Mathematical Statistics*, **33**, 1286–1291. [666](#)
- Paoletta, M. S. 2006. *Fundamental Probability: A Computational Approach*. Wiley & Sons, Inc., New York-London-Sydney-Toronto. [703](#)
- Paoletta, M. S. 2007. *Intermediate Probability: A Computational Approach*. Wiley & Sons, Inc., New York-London-Sydney-Toronto. [703](#)

- Park, S. K., & Miller, K. W. 1988. Random Number Generators: Good ones are hard to find. *Communications of the ACM*, **31**(10), 1192–1201. [513](#)
- Patefield, M., & Tand, D. 2000. Fast and accurate Calculation of OwenâŽs T-Function. *Journal of Statistical Software*. Available as <http://www.jstatsoft.org/v05/i05/paper>. [614](#), [665](#)
- Patnaik, P. B. 1949. The noncentral  $\chi^2$  and  $F$ -distributions and their applications. *Biometrika*, **36**, 202–232. [674](#), [698](#)
- Pearson, J. 2009. Computation of Hypergeometric Functions. *Masters thesis*. Available as [http://people.maths.ox.ac.uk/porterm/research/pearson\\_final.pdf](http://people.maths.ox.ac.uk/porterm/research/pearson_final.pdf). [227](#)
- Pearson, P. B. 1959. Note on an approximation to the distribution of noncentral  $\chi^2$ . *Biometrika*, **46**, 364. [674](#)
- Peizer, David B., & Pratt, John W. 1968. A Normal Approximation for Binomial,  $F$ , Beta, and Other Common, Related Tail Probabilities. I (Ref: P1457-1483). *Journal of the American Statistical Association*, **63**, 1416–1456. [555](#)
- Penev, S., & Raykov, T. 2000. A Wiener Germ Approximation of the Noncentral Chi Square Distribution and of Its Quantiles. *Computational Statistics*, **15**, 219–228. Preprint available at [http://www.researchgate.net/publication/2648631\\_A\\_Wiener\\_Germ\\_Approximation\\_of\\_the\\_Noncentral\\_Chisquare\\_Distribution\\_and\\_of\\_Its\\_Quantiles](http://www.researchgate.net/publication/2648631_A_Wiener_Germ_Approximation_of_the_Noncentral_Chisquare_Distribution_and_of_Its_Quantiles). [675](#)
- Pham-Gia, T. 2008. Exact distribution of the generalized Wilksâ€ statistic and applications. *Journal of Multivariate Analysis*, **99**(8), 1698–1716. Available at <http://www.sciencedirect.com/science/article/pii/S0047259X08000249>. [643](#)
- Pikovsky, A., Rosemblum, M., & Kurths, J. 2001. *Synchronization: A Universal Concept in Nonlinear Sciences*. 1st edn. Cambridge University Press. [615](#)
- Powell, M. J. D. 1994. A direct search optimization method that models the objective and constraint functions by linear interpolation. *Pages 51–67 of: Gomez, S., & Hennart, J.-P. (eds), Advances in Optimization and Numerical Analysis*. Kluwer Academic: Dordrecht. [503](#)
- Powell, M. J. D. 1998. Direct search algorithms for optimization calculations. *Acta Numerica*, **7**, 287–336. [503](#)
- Powell, M. J. D. 2004. The NEWUOA software for unconstrained optimization without derivatives. *Pages 51–67 of: Pardalos, P., & Pillo, G. (eds), Proc. 40th Workshop on Large Scale Nonlinear Optimization (Erice, Italy)*. [504](#)
- Powell, M. J. D. 2009. *The BOBYQA algorithm for bound constrained optimization without derivatives*. technical report NA2009/06. Department of Applied Mathematics and Theoretical Physics, Cambridge England. [504](#)
- Press, W.H., Teukolsky, S.A., Vetterling, W.T., & Flannery, B.P. 2007. *Numerical Recipes: The Art of Scientific Computing*. 3rd edn. Cambridge University Press. Available from <http://www.nr.com/>. [511](#), [517](#), [615](#)
- Price, W. L. 1978. A controlled random search procedure for global optimization. *Pages 71–84 of: Dixon, L. C. W., & Szego, G. P. (eds), Towards Global Optimization 2*. North-Holland Press, Amsterdam. [502](#)

- Price, W. L. 1983. Global optimization by controlled random search. *J. Optim. Theory Appl.*, **40**(3), 333–348. [502](#)
- Pugh, G.R. 2004. An Analysis of the Lanczos Gamma Approximation. *PhD thesis, The University of British Columbia*. Available as <http://laplace.physics.ubc.ca/ThesesOthers/Phd/pugh.pdf>. [517](#)
- Quade, D. 1979. Using weighted rankings in the analysis of complete blocks with additive block effects. *JASA*, **74**, 680–683. [747](#)
- Revol, Nathalie, & Rouillier, Fabrice. 2005. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, **11**(4), 275–290. Online resource: <https://gforge.inria.fr/projects/mpfi/>. [360](#)
- Richardson, J. A., & Kuester, J. L. 1973. The complex method for constrained optimization. *Commun. ACM*, **16**(8), 487–489. [505](#)
- Rinne, H. 2008. *Taschenbuch der Statistik*. 4 edn. Frankfurt, M. : Deutsch. [543](#), [577](#)
- Rinnooy Kan, A. H. G., & Timmer, G. T. 1987a. Stochastic global optimization methods, part I: clustering methods. *Mathematical Programming*, **39**, 27–56. [502](#)
- Rinnooy Kan, A. H. G., & Timmer, G. T. 1987b. Stochastic global optimization methods, part II: multilevel methods. *Mathematical Programming*, **39**, 57–78. [502](#)
- Robillard, Pierre. 1972. Kendall's S Distribution with Ties in One Ranking. *Journal of the American Statistical Association*, **67**(338), 453–455. [729](#), [741](#)
- Robinson, J. 1980. An Asymptotic Expansion for Permutation Tests with Several Samples. *Ann. Stat.*, **8**(4), 851–864. Available at <http://projecteuclid.org/euclid-aos/1176345078>. [745](#)
- Röhmel, Joachim. 1997. The permutation distribution of the Friedman test. *Computational Statistics & Data Analysis*, **26**(1), 83 – 99. [747](#)
- Rowan, T. 1990. *Functional Stability Analysis of Numerical Algorithms*. Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin. [506](#)
- Ruben, H. 1966. Some new results on the distribution of the sample correlation coefficient. *Journal of the Royal Statistical Society, Series B, Methodological*, **28**, 513–525. [654](#)
- Rump, S.M. 1999. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*. Kluwer Academic Publishers, Dordrecht, 77–104. Online resource: <http://www.ti3.tu-harburg.de/~rump/intlab/>. [360](#)
- Runarsson, T. Ph., & Yao, X. 2000. Stochastic ranking for constrained evolutionary optimization. *IEEE Trans. Evolutionary Computation*, **4**(3), 284–294. [503](#)
- Runarsson, T. Ph., & Yao, X. 2005. Search biases in constrained evolutionary optimization. *IEEE Trans. on Systems, Man, and Cybernetics Part C: Applications and Reviews*, **35**(2), 233–243. [503](#)
- Seber, G.A.F. 2008. *A Matrix Handbook for Statisticians*. 1st edn. John Wiley & Sons, Inc. [366](#)

- Shieh, Gwowen, Jan, Show-Li, & Randles, Ronald H. 2007. Power and sample size determinations for the Wilcoxon signed-rank test. *Journal of Statistical Computation and Simulation*, **77**(8), 717–724. [734](#)
- Shorack, R. A. 1966. Recursive Generation of the Distribution of the Mann-Whitney-Wilcoxon U-Statistic Under Generalized Lehmann Alternatives. *Ann. Math. Statist.*, **37**(1), 284–286. Available at <http://projecteuclid.org/euclid.aoms/1177699621>. [730](#), [745](#)
- Shoup, Victor. 2009. *A Computational Introduction to Number Theory and Algebra*. 2nd edn. Cambridge University Press. Available as <http://shoup.net/ntb/ntb-v2.pdf>. [337](#)
- Siotani, Minoru, Hayakawa, Takeshi, & Fujikoshi, Yasunori. 1985. *Modern Multivariate Statistical Analysis: a Graduate Course and Handbook*. American Sciences Press. [636](#)
- Skillings, J.H. 1980. On the null distribution of Jonckheere's statistic used in two way models for ordered alternatives. *Technometrics*, **22**, 431–436. [741](#)
- Skillings, J.H., & Mack, G.A. 1981. On the Use of a Friedman type statistic in balanced and unbalanced block designs. *Technometrics*, **23**, 171–177. [747](#)
- Snow, B. 1962. The third moment of Kendall's tau in normal samples. *Biometrika*, **49**(1-2), 177–183. [738](#)
- Spurrier, John D. 2003. On the null distribution of the Kruskal-Wallis statistic. *Journal of Nonparametric Statistics*, **15**(6), 685–691. [745](#)
- Steck, G. P., & Owen, D. B. 1962. A Note on the Equicorrelated Multivariate Normal Distribution. *Biometrika*, **49**(1/2), 269–271. Article Stable URL: <http://www.jstor.org/stable/2333495>. [666](#)
- Stockmal, Frank. 1962. Algorithm 114: Generation of partitions with constraints. *Communications of the ACM*, **5**, 434–435. [744](#)
- Stoline, M. R., & Ury, H. K. 1979. Tables of the Studentized Maximum Modulus Distribution and an Application to Multiple Comparisons among Means. *Technometrics*, **21**(1), 269–271. Article Stable URL: <http://www.jstor.org/stable/1268584>. [667](#)
- Stoline, Michael R. 1978. Tables of the Studentized Augmented Range and Applications to Problems of Multiple Comparison. *Journal of the American Statistical Association*, **73**(363), 656–660. [668](#)
- Subrahmaniam, Kathleen, & Subrahmaniam, Kocherlakota. 1983. Some Extensions to Miss F. N. David's Tables of Sample Correlation Coefficient: Distribution Function and Percentiles. *Sankhya, Series B, Indian Journal of Statistics*, **45**, 75–147. [647](#)
- Sun, H. J. 1988a. A FORTRAN subroutine for computing normal general reduction formula for n-variate normal orthant probabilities for dimensions up to nine. *Communication in Statistics - Simulation and Computation*, **17**(1), 1097 – 1111. [668](#)
- Sun, H. J. 1988b. A general reduction formula for n-variate normal orthant probability. *Communication in Statistics - Theory and Methods*, **17**(1), 21 – 39. [668](#)

- Sundrum, R. M. 1953. Moments of the Rank Correlation Coefficient  $\tau$  in the General Case. *Biometrika*, **40**(3/4), 409–420. Article Stable URL: <http://www.jstor.org/stable/2333357>. 739
- Sundrum, R. M. 1954. A Further Approximation to the Distribution of Wilcoxon's Statistic in the General Case. *Journal of the Royal Statistical Society. Series B (Methodological)*, **16**(2), 255–260. Article Stable URL: <http://www.jstor.org/stable/2984051>, a draft manuscript as available at [http://repository.lib.ncsu.edu/dr/bitstream/1840.4/2253/1/ISMS\\_1954\\_89.pdf](http://repository.lib.ncsu.edu/dr/bitstream/1840.4/2253/1/ISMS_1954_89.pdf). 731
- Svanberg, K. 2002. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM J. Optim.*, **12**(2), 487–489. 506
- Tang, J., & Gupta, A.K. 1984. On the distribution of the product of independent beta random variables. *Statistics & Probability Letters*, **2**, 165–168. 643, 644
- Tang, J., & Gupta, A.K. 1986. Exact distribution of certain general statistics in multivariate analysis. *Austral. J. Statist.*, **28**, 197–114. 643, 644
- Tang, J., & Gupta, A.K. 1987. On the type-B integral equation and the distribution of Wilks' statistic for testing independence of several groups of variables. *Statistics*, **18**, 379–387. 644
- Tang, Yongqiang. 2011. Size and power estimation for the Wilcoxon–Mann–Whitney test for ordered categorical data. *Statistics in Medicine*, **30**(29), 3461–3470. 729
- Taylor, D. J., & Muller, K. E. 1995. Computing Confidence Bounds for Power and Sample Size of the General Linear Univariate Model. *Am Stat.*, **49**(1), 43–47. 712
- Temme, N. M. 1979. The asymptotic expansion of the incomplete gamma functions. *SIAM J. Math. Anal.*, **10**, 757–766. 160, 519
- Temme, N. M. 1994. A Set of Algorithms for the Incomplete Gamma Functions. *Probability in the Engineering and Informational Sciences*, **8**(4), 291–307. 160, 519
- Temme, Nico M. 1996. *Special Functions: An Introduction to the Classical Functions of Mathematical Physics*. John Wiley & Sons, Inc. Online resource: <http://onlinelibrary.wiley.com/book/10.1002/9781118032572>. 517
- Tiku, M.L. 1966. A Note on Approximating to the Non-Central F-Distribution. *Biometrika*, **53**, 606–610. 698
- Tiwari, Ram C., & Yang, Jie. 1997. Algorithm AS 318: An Efficient Recursive Algorithm for Computing the Distribution Function and Non-centrality Parameter of the Non-central F-distribution. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, **46**(3), 408–413. 603, 681
- Tong, Y. L. 1990. *The multivariate normal distribution*. Springer-Verlag. 666, 667, 671
- Tretter, M. J., & Walster, G. W. 1975. Central and Noncentral Distributions of Wilks' Statistic in Manova as Mixtures of Incomplete Beta Functions. *Ann. Stat.*, **3**(2), 467–472. Available at <http://projecteuclid.org/euclid-aos/1176343073>. 713

- Tretter, M. J., & Walster, G. W. 1979. Continued Fractions for the Incomplete Beta Function: Additions and Corrections. *Ann. Stat.*, **7**(2), 462–465. Available at <http://projecteuclid.org/euclid-aos/1176344629>. 549
- Upton, Graham J. G. 1982. A Comparison of Alternative Tests for the  $2 \times 2$  Comparative Trial. *Journal of the Royal Statistical Society. Series A (General)*, **145**(1), 86–105. Article Stable URL: <http://www.jstor.org/stable/2981423>. 563
- van de Wiel, M.A. 2004. Exact null distributions of quadratic distribution-free statistics for two-way classification. *Journal of Statistical Planning and Inference*, **120**(1–2), 29 – 40. 745
- van de Wiel, M.A., & Di Bucchianico, A. 2001. Fast computation of the exact null distribution of Spearman's rho and Page's L statistic for samples with and without ties. *Journal of Statistical Planning and Inference*, **92**, 133–145. 742, 743
- van Eeden, C. 1961. A higher order approximation to a percentage point of the non-central t-distribution. *International Statistical Review*, **29**, 4–31. 687
- Van Hauwermeiren, M., & Vose, D. 2009. *A Compendium of Distributions*. [ebook]. Vose Software, Ghent, Belgium. Available from <http://www.vosesoftware.com>. 543
- Verzani, John. 2011. *Getting Started with RStudio - An Integrated Development Environment for R*. 1st edn. O'Reilly Media. Online resource: <http://shop.oreilly.com/product/0636920021278.do>. 825
- Vlcek, J., & Luksan, L. 2006. Shifted limited-memory variable metric methods for large-scale unconstrained minimization. *J. Computational Appl. Math.*, **186**, 365–390. 508
- Wakaki, Hirofumi. 2006. Edgeworth expansion of Wilks' lambda statistic. *Journal of Multivariate Analysis*, **97**(9), 1958 – 1964. Available at <http://www.sciencedirect.com/science/article/pii/S0047259X06000790>. 639, 712
- Walck, C. 2007. *Handbook on Statistical Distributions for experimentalists*. University of Stockholm, Internal Report SUF-PFY/96-01. Available as <http://www.stat.rice.edu/~dobelman/textfiles/DistributionsHandbook.pdf>. 543, 552, 554, 682
- Walster, G. W., & Tretter, M. J. 1980. Exact Noncentral Distributions of Wilks'  $\lambda$  and Wilks' Lawley U Criteria as Mixtures of Incomplete Beta Functions: for Three Tests. *Ann. Stat.*, **8**(6), 1388–1390. Available at <http://projecteuclid.org/euclid-aos/1176345210>. 713
- Wang, Hansheng, Chen, Bin, & Chow, Shein-Chung. 2003. Sample Size Determination Based on Rank Tests in Clinical Trials. *Journal of Biopharmaceutical Statistics*, **13**(4), 735–751. PMID: 14584719. 729, 734, 738
- Wang, S., & Gray, H.L. 1993. Approximating tail probabilities of noncentral distributions. *Computational Statistics & Data Analysis*, **15**(3), 343 – 352. Available at: <http://www.sciencedirect.com/science/article/pii/016794739390261Q>. 603, 673, 681, 684, 688, 697
- Wedner, Stefan. 2000. *Verifizierte Bestimmung singulärer Integrale - Quadratur und Kubatur*. Ph.D. thesis, Universität Karlsruhe. Available online at: <http://digibib.ubka.uni-karlsruhe.de/volltexte/documents/3175>. 360

- Wikipedia contributors. 2013. *Pareto distribution* — Wikipedia, The Free Encyclopedia. Available at [http://en.wikipedia.org/w/index.php?title=Pareto\\_distribution&oldid=559561732](http://en.wikipedia.org/w/index.php?title=Pareto_distribution&oldid=559561732). 594
- Wilkening, J. 2008. *gmpfrxx : A C++ interface for GMP and MPFR*. Online resource: <http://math.berkeley.edu/~wilken/code/gmpfrxx/>. 3
- Wilks, S. S. 1932. Certain Generalizations in the Analysis of Variance. *Biometrika*, **24**, 471–494. 643
- Winterbottom, A. 1979. Cornish-Fisher Expansions for Confidence Limits. *Communications in Statistics, Part B* – *Simulation and Computation*, **41**, 69–79. 677
- Winterbottom, A. 1980. Estimation for the Bivariate Normal Correlation Coefficient Using Asymptotic Expansions. *Communications in Statistics, Part B* – *Simulation and Computation [Split from: @J(CommStat)]*, **9**, 599–609. 651, 655
- Witkovsky, V. 2013. A Note on Computing Extreme Tail Probabilities of the Noncentral T Distribution with Large Noncentrality Parameter. *ArXiv e-prints*, June. Article Stable URL: <http://arxiv.org/abs/1306.5294>, Matlab code available at <http://www.mathworks.de/matlabcentral/fileexchange/41790-nctcdfvw/content/nctcdfVW.m>. 684, 685, 688, 691, 692
- Xu, W., Hou, Y., Hung, Y.S., & Zou, Y. 2013. A comparative analysis of SpearmanâŽs rho and KendallâŽs tau in normal and contaminated normal models. *Signal Processing*, **93**, 261–276. 742, 743
- Yoshida, H. 1990. Construction of higher order symplectic integrators. *Physics Letters A*, **150**(5), 262–268. 615
- Zeng, Zhonggang. 2004. Algorithm 835: MultRoot—a Matlab Package for Computing Polynomial Roots and Multiplicities. *ACM Trans. Math. Softw.*, **30**(2), 218–236. Preprint available at <http://www.neiu.edu/~zzeng/Papers/zrootpak.pdf>, Software avaialbe from <http://www.neiu.edu/~zzeng/multroot.htm>. 489
- Zeng, Zhonggang. 2005. Computing multiple roots of inexact polynomials. *Math. Comput.*, **74**(250). Preprint available at <http://www.neiu.edu/~zzeng/mathcomp/zroot.pdf>. 489
- Zertchaninov, S., & Madsen, K. 1998. *A C++ Programme for Global Optimization*. IMM-REP-1998-04. Department of Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark. 503
- Zhao, Yan D. 2006. Sample size estimation for the van Elteren testâŽa stratified WilcoxonâŽs–MannâŽWhitney test. *Statistics in Medicine*, **25**(15), 2675–2687. 729
- Zhao, Yan D., Rahardja, Dewi, & Mei, Yajun. 2008. Sample Size Calculation for the van Elteren Test Adjusting for Ties. *Journal of Biopharmaceutical Statistics*, **18**(6), 1112–1119. PMID: 18991111. 729
- Zimmermann, H. 1985a. Exact Calculation of permutational distributions for two dependent samples. *Biometrical Journal*, **27**, 349–352. 735
- Zimmermann, H. 1985b. Exact Calculation of permutational distributions for two independent samples. *Biometrical Journal*, **27**, 431–434. 729

# Index

## Multiprecision Functions

–, 54, 329  
\*, 56, 77, 329  
+, 52, 75, 329  
.Div, 58, 330  
.DivInt, 330  
.DotProd, 56, 329  
.EQ, 63, 330  
.GE, 63, 331  
.GT, 63, 331  
.LE, 63, 331  
.LSH, 56, 329  
.LT, 64, 331  
.Minus, 54, 329  
.MinusInt, 329  
.Mod, 60, 330  
.ModInt, 330  
.NE, 64, 332  
.Plus, 52, 75, 329  
.PlusInt, 329  
.Pow, 61, 330  
.PowInt, 330  
.RSH, 58, 330  
.Sorted, 334  
.Times, 56, 329  
.TimesInt, 329  
.TimesMat, 56, 77, 329  
.nint\_distance, 69  
/, 58, 330  
=, 63, 64, 330–332  
^, 61, 330  
abs, 49  
acos, 128  
acosh, 135  
AcoshBoost, 541  
acot, 133  
acoth, 137  
agm, 183  
airyai, 202

AiryAiBoost, 532  
AiryAiDerivativeBoost, 532  
airyaizero, 205  
airybi, 203  
AiryBiBoost, 533  
AiryBiDerivativeBoost, 533  
airybizer0, 205  
almosteq, 64  
altzeta, 275  
angerj, 198  
apery, 83  
appellf1, 250  
appellf2, 251  
appellf3, 251  
appellf4, 252  
arange, 70  
arg, 50  
asin, 126  
asinh, 134  
AsinhBoost, 541  
AssociatedLaguerreBoost, 527  
AssociatedLaguerreLNextBoost, 527  
AssociatedLaguerreMpMath, 226  
AssociatedLegendrePlmBoost, 525  
AssociatedLegendrePlmNextBoost, 526  
atan, 130  
Atan2, 132  
atanh, 136  
AtanhBoost, 542  
backlunds, 282  
barnesg, 155  
bei, 194  
bell, 301  
ber, 194  
bernfrac, 296  
bernoulli, 295  
BernoulliDistBoost, 578  
BernoulliDistInfoBoost, 579  
BernoulliDistInvBoost, 579

BernoulliDistRandomBoost, 580  
 bernpoly, 297  
 besseli, 187  
 BesselIBlack, 529  
 BesselIEmpMath, 184  
 besselj, 184  
 BesselJBoost, 529  
 besseljzero, 190  
 besselk, 188  
 BesselKBoost, 530  
 BesselKeMpMath, 184  
 BesselSphericaljBoost, 530  
 BesselSphericalyBoost, 530  
 bessely, 186  
 BesselYBoost, 529  
 besselyzero, 191  
 beta, 165  
 BetaBoost, 522  
 BetaDist, 550  
 BetaDistInfo, 551  
 BetaDistInv, 550  
 BetaDistRandom, 552  
 betainc, 165  
 BetaProductDist, 643  
 BetaProductDistInfo, 645  
 BetaProductDistInv, 644  
 BetaProductDistRandom, 645  
 bihyper, 248  
 binomial, 150  
 BinomialCoefficientBoost, 522  
 BinomialDist, 552  
 BinomialDistInfo, 553  
 BinomialDistInv, 553  
 BinomialDistRandom, 554  
 BoxDavisDist, 636  
 BoxDavisDistInv, 637  
 CarlsonRCBoost, 534  
 CarlsonRDBoost, 534  
 CarlsonRFBoost, 534  
 CarlsonRJBoost, 535  
 catalan, 83  
 CauchyDistBoost, 580  
 CauchyDistInfoBoost, 581  
 CauchyDistInvBoost, 580  
 CauchyDistRandomBoost, 581  
 cbt, 97  
 CbrtBoost, 540  
 CDist, 555  
 CDistInfo, 556  
 CDistInv, 556  
 CDistRan, 556  
 ceil, 44  
 CentralMomentsToCumulants, 630  
 CentralMomentsToRawMoments, 629  
 chebyt, 219  
 chebyu, 219  
 chi, 174  
 cholesky, 140  
 chop, 64  
 ci, 172  
 clesos, 289  
 clesinlog, 288  
 CompleteLegendreEllint1Boost, 535  
 CompleteLegendreEllint2Boost, 535  
 CompleteLegendreEllint3Boost, 536  
 conj, 51  
 cos, 105  
 cosh, 119  
 cosm, 315  
 CosPiBoost, 539  
 cot, 113  
 coth, 125  
 coulombc, 209  
 coulombf, 208  
 coulombg, 209  
 cplxAsum, 367  
 cplxApxy, 367  
 cplxCubicEquation, 487  
 cplxDecompCholeskyLDLT, 391  
 cplxDecompCholeskyLLT, 387  
 cplxDecompColPivQR, 409  
 cplxDecompFullPivLU, 399  
 cplxDecompFullPivQR, 414  
 cplxDecompJacobiSVD, 420  
 cplxDecompPartialPivLU, 395  
 cplxDecompQR, 405  
 cplxDetCholeskyLDLT, 394  
 cplxDetCholeskyLLT, 389  
 cplxDetColPivQR, 413  
 cplxDetFullPivLU, 404  
 cplxDetFullPivQR, 418  
 cplxDetJacobiSVD, 422  
 cplxDetPartialPivLU, 397  
 cplxDetQR, 408  
 cplxDotc, 366  
 cplxDotu, 366  
 cplxEigenGenherm, 449  
 cplxEigenGenhermv, 449

- cplxEigenHerm, 432  
cplxEigenHermv, 432  
cplxEigenNonsymm, 440  
cplxEigenNonsymmv, 440  
cplxGemm, 376  
cplxGemm, 368  
cplxGeneralPolynomialEquation, 489  
cplxGerc, 372  
cplxGeru, 372  
cplxHankel1Boost, 531  
cplxHankel2Boost, 531  
cplxHankelSph1Boost, 531  
cplxHankelSph2Boost, 532  
cplxHemm, 378  
cplxHemv, 371  
cplxHer, 373  
cplxHer2, 374  
cplxHer2k, 386  
cplxHerk, 384  
cplxInvertCholeskyLDLT, 393  
cplxInvertCholeskyLLT, 389  
cplxInvertColPivQR, 413  
cplxInvertFullPivLU, 404  
cplxInvertFullPivQR, 417  
cplxInvertJacobiSVD, 422  
cplxInvertPartialPivLU, 397  
cplxInvertQR, 408  
cplxMatCos, 482  
cplxMatCosh, 484  
cplxMatExp, 473  
cplxMatGeneralFunction, 479  
cplxMatLog, 474  
cplxMatPow, 476  
cplxMatSin, 480  
cplxMatSinh, 482  
cplxMatSqrt, 471  
cplxNrm2, 367  
cplxPolynomialEvaluation, 485  
cplxQuadraticEquation, 486  
cplxQuarticEquation, 488  
cplxSolveCholeskyLDLT, 393  
cplxSolveCholeskyLLT, 389  
cplxSolveColPivQR, 412  
cplxSolveFullPivLU, 403  
cplxSolveFullPivQR, 417  
cplxSolveJacobiSVD, 421  
cplxSolvePartialPivLU, 396  
cplxSolveQR, 407  
cplxSymm, 377  
cplxSyr2k, 385  
cplxSyrk, 383  
cplxTrmm, 379  
cplxTrmv, 369  
cplxTrsm, 381  
cplxTrsv, 370  
csc, 111  
csch, 124  
CubicEquation, 487  
CumulantsToCentralMoments, 630  
CumulantsToRawMoments, 630  
cyclotomic, 307  
DebyeMpMath, 288  
DecompCholeskyLDLT, 391  
DecompCholeskyLLT, 387  
DecompColPivQR, 409  
DecompFullPivLU, 399  
DecompFullPivQR, 414  
DecompJacobiSVD, 420  
DecompPartialPivLU, 395  
DecompQR, 405  
degree, 82  
degrees, 102  
DetCholeskyLDLT, 393  
DetCholeskyLLT, 389  
DetColPivQR, 413  
DetFullPivLU, 404  
DetFullPivQR, 418  
DetJacobiSVD, 422  
DetPartialPivLU, 397  
DetQR, 408  
digamma, 164  
DigammaBoost, 518  
dilog, 288  
dirichlet, 276  
DirichletBetaMpMath, 276  
DirichletEtam1MpMath, 275  
DirichletLambdaMpMath, 276  
DoubleFactorialBoost, 521  
DoublyNoncentralFDistEx, 703  
DoublyNoncentralFDistInfoEx, 706  
DoublyNoncentralFDistInvEx, 706  
DoublyNoncentralFDistNoncentralityEx, 708  
DoublyNoncentralFDistRanEx, 708  
DoublyNoncentralFDistSampleSizeEx, 709  
DoublyNoncentralTDist, 691  
DoublyNoncentralTDistInfo, 694  
DoublyNoncentralTDistInv, 693  
DoublyNoncentralTDistNoncentrality, 695

DoublyNoncentralTDistRan, 694  
 DoublyNoncentralTDistSampleSize, 695  
 e, 83  
 e1, 169  
 ei, 168  
 eig, 325  
 EigenGenNonsymm, 450  
 EigenGenNonsymmv, 450  
 EigenGensymm, 445  
 EigenGensymmv, 446  
 EigenNonsymm, 435  
 EigenNonsymmv, 435  
 EigenSymm, 428  
 EigenSymmv, 428  
 eigh, 326  
 ellipe, 259  
 ellipef, 259  
 ellipf, 257  
 ellipfun, 270  
 ellipk, 257  
 ellippi, 260  
 ellippif, 261  
 elliprc, 264  
 elliprd, 266  
 elliprf, 263  
 elliprg, 266  
 elliprj, 265  
 erf, 175  
 ErfBoost, 523  
 erfc, 176  
 ErfcBoost, 523  
 ErfcInvBoost, 524  
 erfi, 176  
 erfinv, 177  
 ErfInvBoost, 523  
 euler, 83  
 eulernum, 299  
 eulerpoly, 299  
 exp, 85  
 exp10, 88  
 exp2, 89  
 expint, 169  
 expj, 87  
 expjpi, 87  
 expm, 313  
 expm1, 88  
 Expm1Boost, 540  
 ExponentialDist, 557  
 ExponentialDistInfo, 558  
 ExponentialDistInv, 557  
 ExponentialDistRandom, 558  
 ExponentialIntegralE1Boost, 538  
 ExponentialIntegralEiBoost, 539  
 ExponentialIntegralEnBoost, 539  
 ExtremevalueDistBoost, 582  
 ExtremevalueDistInfoBoost, 583  
 ExtremevalueDistInvBoost, 582  
 ExtremevalueDistRandomBoost, 583  
 fabs, 49  
 fac, 147  
 fac2, 148  
 Factorial, 147  
 FactorialBoost, 521  
 fadd, 52  
 FallingFactorialBoost, 522  
 FDist, 559  
 FDistInfo, 560  
 FDistInv, 559  
 FDistRan, 560  
 fdiv, 59  
 fdot, 57  
 FermiDirac3HalfMpMath, 285  
 FermiDiracHalfMpMath, 285  
 FermiDiracIntMpMath, 285  
 FermiDiracPHalfMpMath, 285  
 ff, 152  
 FFTW\_BACKWARD, 491  
 FFTW\_C2R, 492  
 FFTW\_FORWARD, 491  
 FFTW\_R2C, 492  
 FFTW\_REDFT00, 493  
 FFTW\_REDFT01, 494  
 FFTW\_REDFT10, 493  
 FFTW\_REDFT11, 494  
 FFTW\_RODFT00, 495  
 FFTW\_RODFT01, 495  
 FFTW\_RODFT10, 495  
 FFTW\_RODFT11, 496  
 fib, 293  
 fibonacci, 293  
 floor, 44  
 fmod, 60  
 fmul, 56  
 fneg, 55  
 fprod, 57  
 frac, 45  
 fraction, 70  
 fresnelc, 180

fresnels, 180  
frexp, 47  
fsub, 54  
fsum, 53  
gamma, 156  
GammaDist, 560  
GammaDistInfo, 561  
GammaDistInv, 561  
GammaDistRandom, 562  
gammainc, 159  
Gammap, 624  
GammaPBoost, 519  
GammaPDerivativeBoost, 521  
GammaPDerivativeMpMath, 160  
GammaPinvaBoost, 520  
GammaPinvBoost, 520  
GammaPinvMpMath, 162  
GammaPMpMath, 160  
gammaprod, 157  
GammaQBoost, 519  
GammaQinvaBoost, 520  
GammaQinvBoost, 520  
GammaQinvMpMath, 162  
GammaQMpMath, 160  
gegenbauer, 222  
GeneralizedExponentialIntegralEpMpMath, 170  
GeneralPolynomialEquation, 488  
GeometricDistBoost, 583  
GeometricDistInfoBoost, 584  
GeometricDistInvBoost, 584  
GeometricDistRandomBoost, 584  
glaisher, 83  
grampoint, 282  
hankel1, 193  
hankel2, 193  
harmonic, 164  
hermite, 224  
HermiteHBoost, 528  
HermiteHNextBoost, 528  
hurwitz, 273  
hyp0f1, 228  
hyp1f1, 230  
hyp1f2, 239  
hyp2f0, 232  
hyp2f1, 236  
hyp2f2, 239  
hyp2f3, 240  
hyp3f2, 240  
hyper, 242  
hyper2d, 249  
hypercomb, 243  
hyperfac, 154  
Hypergeometric0F1Matrix, 628  
Hypergeometric0F1RegularizedMpMath, 229  
Hypergeometric1F1Matrix, 627  
Hypergeometric1F1RegularizedMpMath, 231  
Hypergeometric2F1Matrix, 626  
Hypergeometric2F1RegularizedMpMath, 238  
HypergeometricDist, 563  
HypergeometricDistInfo, 564  
HypergeometricDistInv, 564  
HypergeometricDistRandom, 565  
hyperu, 232  
hypot, 97  
HypotBoost, 540  
IBetaMpMath, 167  
IBetaNonNormalizedMpMath, 166  
im, 49  
inf, 83  
intAbs, 346  
intAdd, 337  
intAND, 344  
intBinCoeff, 348  
intCDivQ, 340  
intCDivQ2exp, 340  
intCDivQR, 341  
intCDivR, 341  
intCDivR2exp, 341  
intClearBit, 345  
intComBit, 345  
intComplement, 344  
intDivExact, 339  
intFactorial, 348  
intFDivQ, 341  
intFDivQ2exp, 341  
intFDivQR, 342  
intFDivR, 342  
intFDivR2exp, 342  
intFibonacci, 351  
intFma, 338  
intFms, 338  
intGcd, 349  
intGcdExt, 349  
intHamDist, 344  
intInvertMod, 350  
intIOR, 344  
intIsBpswPrp, 351

intIsEulerPrp, 352  
 intIsExtraStrongLucasPrp, 352  
 intIsFermatPrp, 352  
 intIsFibonacciPrp, 352  
 intIsLucasPrp, 353  
 intIsSelfridgePrp, 353  
 intIsStrongBpswPrp, 353  
 intIsStrongLucasPrp, 353  
 intIsStrongPrp, 354  
 intIsStrongSelfridgePrp, 354  
 intJacobiSymbol, 350  
 intKroneckerSymbol, 351  
 intLcm, 349  
 intLegendreSymbol, 350  
 intLSH, 339  
 intLucas, 351  
 intLucasModU, 355  
 intLucasModV, 355  
 intLucasU, 354  
 intLucasV, 355  
 intMod, 339  
 intMul, 338  
 intNeg, 337  
 intNextprime, 349  
 intPopCount, 346  
 intPow, 347  
 intPowMod, 347  
 intRemoveFactor, 350  
 intRoot, 348  
 intRootRem, 348  
 intRrandomb, 356  
 intRSH, 339  
 intScan0, 345  
 intScan1, 346  
 intSetBit, 345  
 intSgn, 346  
 intSqrt, 347  
 intSqrtRem, 347  
 intSub, 337  
 intTDivQ, 342  
 intTDivQ2exp, 343  
 intTDivQr, 343  
 intTDivR, 343  
 intTDivR2exp, 343  
 intTestBit, 345  
 intUrandomb, 356  
 intUrandomm, 356  
 intXOR, 344  
 InverseChiSquaredDistBoost, 585  
 InverseChiSquaredDistInfoBoost, 586  
 InverseChiSquaredDistInvBoost, 586  
 InverseChiSquaredDistRanBoost, 586  
 InverseGammaDistBoost, 587  
 InverseGammaDistInfoBoost, 588  
 InverseGammaDistInvBoost, 588  
 InverseGammaDistRanBoost, 589  
 InverseGaussianDistBoost, 589  
 InverseGaussianDistInfoBoost, 590  
 InverseGaussianDistInvBoost, 590  
 InverseGaussianDistRanBoost, 591  
 InverseTangentMpMath, 286  
 InvertCholeskyLDLT, 393  
 InvertCholeskyLLT, 389  
 InvertColPivQR, 412  
 InvertFullPivLU, 403  
 InvertFullPivQR, 417  
 InvertJacobiSVD, 421  
 InvertPartialPivLU, 396  
 InvertQR, 407  
 IsCongruent, 356  
 IsCongruent2exp, 357  
 IsDivisible, 357  
 IsDivisible2exp, 358  
 IsEmpty, 361  
 isfinite, 66  
 isnf, 67  
 IsInside, 361  
 isint, 68  
 isnan, 67  
 isnormal, 66  
 IsPerfectPower, 358  
 IsPerfectSquare, 358  
 IsProbablyPrime, 357  
 IsStrictlyInside, 361  
 IsStrictlyNeg, 362  
 IsStrictlyPos, 362  
 j0, 186  
 j1, 186  
 jacobi, 221  
 JacobiCNBoost, 537  
 JacobiDNBoost, 538  
 JacobiSNBoost, 537  
 jtheta, 268  
 kei, 195  
 ker, 194  
 kfrom, 256  
 khinchin, 83  
 kleinj, 272

laguerre, 225  
LaguerreLBoost, 526  
LaguerreLMpMath, 226  
LaguerreLNextBoost, 527  
lambertw, 182  
LaplaceDistBoost, 591  
LaplaceDistInfoBoost, 592  
LaplaceDistInvBoost, 592  
LaplaceDistRanBoost, 592  
ldexp, 47  
legendre, 214  
LegendreChiMpMath, 286  
LegendreEllint1Boost, 536  
LegendreEllint2Boost, 536  
LegendreEllint3Boost, 537  
LegendrePBoost, 525  
LegendrePNextBoost, 525  
LegendreQBoost, 526  
legenp, 215  
legenq, 216  
lerchphi, 284  
LgammaBoost, 517  
li, 171  
linspace, 71  
ln, 90  
LnBetaMpMath, 165  
lnp1, 92  
Lnp1Boost, 539  
LoadDefaultRngState, 512  
log, 90  
log10, 92  
log2, 92  
Logb, 90  
loggamma, 158  
LogisticDistBoost, 593  
LogisticDistInfoBoost, 594  
LogisticDistInvBoost, 593  
LogisticDistRanBoost, 594  
logm, 319  
LogNormalDist, 566  
LognormalDistInfo, 566  
LognormalDistInv, 566  
LognormalRandom, 567  
lommels1, 200  
lommels2, 200  
lu, 144  
lu\_solve, 142  
mag, 68  
mangoldt, 308  
MatCos, 482  
MatCosh, 483  
MatExp, 473  
MatGeneralFunction, 479  
MatLog, 474  
MatPow, 476  
matrix, 72  
MatrixAdd, 76  
MatSin, 480  
MatSinh, 482  
MatSqrt, 471  
MatSymmInverseSqrt, 430  
MatSymmSqrt, 431  
meijerg, 245  
mertens, 83  
mfrom, 255  
mnorm, 139  
mod, 60, 330  
mpc, 48  
nan, 83  
ncdf, 179  
NDist, 570  
NDistInv, 571  
NegativeBinomialDist, 568  
NegativeBinomialDistInfo, 568  
NegativeBinomialDistInv, 568  
NegativeBinomialDistRandom, 569  
nint, 45  
NoncentralBartlettsMDist, 725  
NoncentralBartlettsMDistInv, 726  
NoncentralBartlettsMDistRan, 727  
NoncentralBartlettsMInfo, 726  
NoncentralBetaDistBoost, 603, 681  
NoncentralBetaDistInfoBoost, 604, 682  
NoncentralBetaDistInvBoost, 604, 681  
NoncentralBetaDistRanBoost, 604, 683  
NoncentralCDistBoost, 605  
NoncentralCDistEx, 673  
NoncentralCDistInfoBoost, 606  
NoncentralCDistInfoEx, 676  
NoncentralCDistInvBoost, 606  
NoncentralCDistInvEx, 675  
NoncentralCDistNoncentralityEx, 677  
NoncentralCDistRanBoost, 607  
NoncentralCDistRanEx, 678  
NoncentralCDistSampleSizeEx, 678  
NoncentralFDistBoost, 607, 697  
NoncentralFDistInfoBoost, 608, 700  
NoncentralFDistInvBoost, 608, 699

NoncentralFDistNoncentralityEx, 702  
 NoncentralFDistRanBoost, 609, 701  
 NoncentralFDistSampleSizeEx, 702  
 NoncentralTDistBoost, 610, 684  
 NoncentralTDistInfoBoost, 611, 688  
 NoncentralTDistInvBoost, 610, 687  
 NoncentralTDistNoncentrality, 689  
 NoncentralTDistRanBoost, 611, 688  
 NoncentralTDistSampleSize, 690  
 NoncentralWilksLambdaDist, 712  
 NoncentralWilksLambdaDistInfo, 717  
 NoncentralWilksLambdaDistInv, 717  
 NoncentralWilksLambdaDistRan, 718  
 NonNormalisedGammaPBoost, 519  
 NonNormalisedGammaPMpMath, 161  
 NonNormalisedGammaQBoost, 519  
 NonNormalisedGammaQMpMath, 161  
 norm, 138  
 NormalDistInfo, 571  
 NormalRandom, 572  
 npdf, 179  
 nthroot, 98  
 nzeros, 279  
 ParetoDistBoost, 594  
 ParetoDistInfoBoost, 595  
 ParetoDistInvBoost, 595  
 ParetoDistRanBoost, 596  
 pcf, 211  
 pcfu, 211  
 pcfv, 212  
 pcfw, 213  
 PearsonRhoDist, 647  
 PearsonRhoDistInfo, 652  
 PearsonRhoDistInv, 651  
 PearsonRhoDistNoncentrality, 654  
 PearsonRhoDistRan, 653  
 PearsonRhoDistSampleSize, 655  
 phase, 50  
 phi, 83  
 pi, 82  
 PoissonDist, 572  
 PoissonDistInfo, 573  
 PoissonDistInv, 573  
 PoissonDistRan, 574  
 polar, 48  
 polyexp, 290  
 polygamma, 163  
 polylog, 287  
 PolynomialEvaluation, 485  
 power, 94  
 powm, 321  
 powm1, 95  
 Powm1Boost, 540  
 primepi, 304  
 primepi2, 304  
 primezeta, 291  
 PseudoEigenNonsymm, 435  
 PseudoEigenNonsymmv, 436  
 psi, 163  
 qbarfrom, 255  
 qfac, 310  
 qfrom, 254  
 qgamma, 310  
 qhyper, 312  
 qp, 309  
 qr, 144  
 QuadraticEquation, 486  
 QuarticEquation, 488  
 radians, 102  
 RaleighDistBoost, 596  
 RaleighDistInfoBoost, 597  
 RaleighDistInvBoost, 597  
 RaleighDistRanBoost, 597  
 rand, 70  
 RAsum, 367  
 RawMomentsToCentralMoments, 629  
 RawMomentsToCumulants, 630  
 RAxpy, 367  
 RDot, 366  
 re, 48  
 rect, 48  
 RelativePochhammerMpMath, 151  
 residual, 142  
 rf, 151  
 rgamma, 157  
 RGemm, 374  
 RGemm, 368  
 RGer, 372  
 Rho2Dist, 656  
 Rho2DistInfo, 659  
 Rho2DistInv, 659  
 Rho2DistNoncentrality, 661  
 Rho2DistRan, 660  
 Rho2DistSampleSize, 661  
 riemannr, 305  
 RiemannZetaBoost, 538  
 RisingFactorialBoost, 521  
 RNrm2, 366

root, 98  
RSymm, 377  
RSymv, 371  
RSyr, 373  
RSyr2, 374  
Rsyrik, 385  
Rsyrik, 383  
RTrmm, 379  
RTrmv, 369  
RTrsm, 381  
RTrsv, 370  
SaveDefaultRngState, 512  
schur, 324  
scorergi, 206  
scorerhi, 206  
sec, 109  
sech, 123  
secondzeta, 291  
shi, 174  
si, 172  
siegeltheta, 281  
siegelz, 281  
sign, 50  
sin, 103  
SincaBoost, 541  
sinh, 117  
SinhcaBoost, 541  
sinn, 316  
SinPiBoost, 539  
SkewNormalDistBoost, 612, 663  
SkewNormalDistInfoBoost, 613, 664  
SkewNormalDistInvBoost, 613, 663  
SkewNormalDistRanBoost, 614, 664  
SolveCholeskyLDLT, 392  
SolveCholeskyLLT, 388  
SolveColPivQR, 412  
SolveFullPivLU, 403  
SolveFullPivQR, 417  
SolveJacobiSVD, 421  
SolvePartialPivLU, 396  
SolveQR, 407  
spherharm, 217  
SphericalHarmonicBoost, 528  
sqrt, 96  
sqrtm, 317  
Sqrtp1m1Boost, 540  
square, 93  
stieltjes, 278  
stirling1, 302  
stirling2, 302  
struveh, 196  
struvel, 197  
StudentTTest1, 754  
StudentTTest2i, 761  
StudentTTestPower1, 757  
StudentTTestPower2i, 764  
StudentTTestSampleSize1, 758  
StudentTTestSampleSize2i, 766  
superfac, 153  
svd, 323  
tan, 107  
tanh, 121  
taufrom, 256  
TDist, 574  
TDistInfo, 575  
TDistInv, 575  
TDistRan, 576  
TgammaBoost, 517  
TgammaDeltaRatioBoost, 518  
TgammaRatioBoost, 518  
TOwenBoost, 614, 665  
TriangularDistBoost, 598  
TriangularDistInfoBoost, 599  
TriangularDistInvBoost, 598  
TriangularDistRanBoost, 599  
TricomiGammaMpMath, 161  
twinprime, 83  
UniformDistBoost, 600  
UniformDistInfoBoost, 601  
UniformDistInvBoost, 600  
UniformDistRanBoost, 601  
webere, 198  
WeibullDist, 576  
WeibullDistInfo, 577  
WeibullDistInv, 577  
WeibullDistRandom, 578  
whitm, 234  
whitw, 234  
ZernikeRadialMpMath, 222  
zeta, 273  
zetazero, 279  
Multiprecision Properties  
.Adjoint, 334  
.AsDiagonal, 334  
.Block, 334  
.BottomLeftCorner, 334  
.BottomRightCorner, 334  
.BottomRows, 333

.Col, 333  
.Cols, 332  
.Diagonal, 334  
.FillLinearByStep, 334  
.Item, 333  
.LeftCols, 333  
.MiddleCols, 333  
.MiddleRows, 333  
.Prec10, 333  
.Prec2, 333  
.RightCols, 333  
.Row, 333  
.Rows, 332  
.SetRandomSymmetric, 334  
.Size, 332  
.TopLeftCorner, 334  
.TopRightCorner, 334  
.TopRows, 333  
.TriangularView, 334

## Spreadsheet Procedures

Sampling, 512