

The mpFormulaPy Library and Toolbox Manual

Dietrich Hadler
Helge Hadler
Thomas Hadler

July 2015
Version 0.1, Alpha 3 (Pre-Release)

Original authors of the mpmath documentation: Frederik Johanson and mpmath contributors

Original authors of the Python documentation: the Python development Team

Subsequent modifications: Dietrich Hadler, Helge Hadler and Thomas Hadler

Copyright © 2015 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.

Preface

The mpFormulaPy library provides a comprehensive set of real and complex functions in multiprecision arithmetic. For a subset of functions there is also support for decimal and interval arithmetic.

Additional planned functionality includes integration in Microsoft Excel (Windows) and LibreOffice Calc (Windows, Mac OSX, GNU/Linux), with multiprecision support for the numerical functions of these spreadsheet programs.

mpFormulaPy is based on the Python runtime and the mpmath library, both of which implement multiprecision arithmetic.

The current version is 0.1, alpha1 (pre-release), 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 with Error Bounds.

Functions in this part come optionally with a guaranteed error bound, which can (in principal) be made arbitrarily small. The set of functions includes all real and complex functions that are included in Microsoft Excel and OpenOffice/LibreOffice Calc, including the financial functions.

Part III: Special Functions

Functions in this part include a rich set of real and complex functions with an emphasis on mathematical physics. These functions are available in arbitrary precision, but without guaranteed error bounds.

Part IV: Numerical Calculus

Functions in this part include root-finding and optimization, differentiation and integration, sums, products and limit.

Part V: Application Examples

This is work in progress, with an emphasis on statistical distribution functions.

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

Appendix A describes the interfaces to a number of popular flavors of Python, including CPython (Windows, Mac OSX, GNU/Linux), Jython (Windows, Mac OSX, GNU/Linux), IronPython (Windows), QPython (Android) and Pythonista (iOS).

Appendix B describes the interfaces to popular spreadsheet programs: LibreOffice Calc (Windows, Mac OSX, GNU/Linux) and Microsoft Excel (Windows, Mac OSX).

Windows-specific interfaces to languages with CLR support (e.g. C#, F#, Visual Basic, Matlab) or COM support (e.g. VBA, JScript, Perl, R) are described in appendices C and D.

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

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

Dietrich Hadler

Helge Hadler

Thomas Hadler

Contents

Preface	i
Contents	iii
List of Tables	xxvi
List of Figures	xxvii
I Getting Started	1
1 Introduction	2
1.1 Overview: Features and Setup	2
1.1.1 Features	2
1.1.2 The mpFormulaPy Library	2
1.1.3 The mpFormulaPy 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
2 Tutorials	5
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
2.4	Eval, Options, Tables and Charts	15
3	Python: Built-in numerical types	16
3.1	Truth Value Testing	16
3.2	Boolean Operations: and, or, not	16
3.3	Comparisons	17
3.4	Numeric Types - int, float, complex	17
3.5	Long integers	19
3.5.1	Bitwise Operations on Integer Types	19
3.5.2	Additional Methods on Integer Types	19
3.5.3	Additional Methods on Float	21
3.6	Fractions	23
3.6.1	Properties	24
3.6.2	Methods	24
3.7	Decimals	26
3.7.1	Overview	26
3.7.2	Quick-start Tutorial	27
3.7.3	Decimal objects	30
3.7.4	Methods	32
3.7.5	Context objects	39
3.7.6	Context Methods	41
3.7.7	Constants	47
3.7.8	Rounding modes	48
3.7.9	Signals	48
3.7.10	Floating Point Notes	50
3.7.11	Working with threads	52
3.7.12	Recipes	52
3.7.13	Decimal FAQ	55
II	Functions with Error Bounds	59
4	Basic Usage	60
4.1	Number types	60
4.1.1	Setting the precision	61
4.1.2	Temporarily changing the precision	62
4.1.3	Providing correct input	63
4.1.4	Printing	64
4.1.5	Contexts	65

4.1.6	Common interface	65
4.1.7	Arbitrary-precision floating-point (mp)	67
4.1.8	Arbitrary-precision interval arithmetic (iv)	67
4.1.9	Fast low-precision arithmetic (fp)	70
4.2	Precision and representation issues	72
4.2.1	Precision, error and tolerance	72
4.2.2	Representation of numbers	73
4.2.3	Decimal issues	74
4.2.4	Correctness guarantees	75
4.2.5	Double precision emulation	76
4.3	Conversion of formatted numbers	77
4.3.1	Conversions between Roman and Arabic Numbers	77
4.3.2	Conversions from Binary	77
4.3.3	Conversions from Decimal	78
4.3.4	Conversions from Hexadecimal	79
4.3.5	Conversions from Octal	80
4.3.6	Conversion to and from a Given Base	80
4.4	Conversion and printing	82
4.4.1	convert()	82
4.4.2	nstr()	82
4.5	Rounding	83
4.5.1	Nearest integer: Round(x)	83
4.5.2	Next higher or equal integer: Ceil(x)	83
4.5.3	Next lower or equal integer: Floor(x)	84
4.5.4	Next integer, rounded toward zero: Trunc(x)	86
4.5.5	EVEN(x)	86
4.5.6	ODD(x)	86
4.5.7	Nearest integer	86
4.5.8	Fractional Part	87
4.5.9	ROUNDDOWN(<i>Number, Digits</i>)	88
4.5.10	ROUNDUP(<i>Number, Digits</i>)	88
4.5.11	MROUND(<i>Number, Multiple</i>)	88
4.5.12	QUOTIENT(x, y)	89
4.6	Components of Real and Complex Numbers	90
4.6.1	Number generated from Significand and Exponent: Ldexp(x, y)	90
4.6.2	Significand and Exponent: Frexp(x)	90
4.6.3	Building a Complex Number from Real Components	91
4.6.4	Representations of Complex Numbers	91
4.6.5	Real Component	92
4.6.6	Imaginary Component	92
4.6.7	Absolute Value	93
4.6.8	Argument	94
4.6.9	Sign	95
4.6.10	Conjugate	95
4.7	Arithmetic operations	97

4.7.1	Addition and Sum	97
4.7.2	Sums and Series	98
4.7.3	Substraction	101
4.7.4	Negation	102
4.7.5	Multiplication	103
4.7.6	Products	104
4.7.7	Multiplication by multiples of 2 (LSH)	106
4.7.8	Division	107
4.7.9	Division by multiples of 2 (RSR)	108
4.7.10	Modulo	108
4.7.11	Power	109
4.8	Logical Operators	110
4.8.1	Bitwise AND	110
4.8.2	Bitwise Inclusive OR	110
4.8.3	Bitwise Exclusive OR	110
4.9	Comparison Operators and Sorting	111
4.9.1	Equal	111
4.9.2	Greater or equal	111
4.9.3	Greater than	111
4.9.4	Less or equal	111
4.9.5	Less than	112
4.9.6	Not equal	112
4.9.7	Tolerances and approximate comparisons	112
4.10	Properties of numbers	115
4.10.1	Testing for special values	115
4.10.2	Testing for integers	117
4.10.3	Approximating magnitude and precision	118
4.11	Number generation	120
4.11.1	Random numbers	120
4.11.2	Fractions	120
4.11.3	Ranges	121
4.12	Matrices	123
4.12.1	Basic methods	123
4.12.2	Vectors	125
4.12.3	Other	125
4.12.4	Transposition	126
4.12.5	Matrix Properties	126
4.12.6	Addition	127
4.12.7	Multiplication	128
4.12.8	Python-specific convenience functions	129
5	Elementary Functions	133
5.1	Constants	133
5.1.1	Mathematical constants	133
5.1.2	Special values	134

5.2	Exponential and Logarithmic Functions	136
5.2.1	Exponential Function $e^z = \exp(z)$	136
5.2.2	Exponential Function $10^z = \exp_{10}(z)$	140
5.2.3	Exponential Function $2^z = \exp_2(z)$	140
5.2.4	Natural logarithm $\ln(x) = \log_e(x)$	141
5.2.5	Common (decadic) logarithm $\log_{10}(z)$	143
5.2.6	Binary logarithm $\log_2(z)$	144
5.2.7	Auxiliary Function $\ln(1 + x)$	144
5.3	Roots and Power Functions	145
5.3.1	Square: z^2	145
5.3.2	Power Function	146
5.3.3	Square Root: \sqrt{z}	149
5.3.4	Auxiliary Function $\sqrt{x^2 + y^2}$	150
5.3.5	Cube Root: $\sqrt[3]{x}, n = 2, 3, \dots$	151
5.3.6	Nth Root: $\sqrt[n]{z}, n = 2, 3, \dots$	151
5.4	Trigonometric Functions	156
5.4.1	Trigonometric functions: overview	156
5.4.2	Conversion between Degrees and radians	156
5.4.3	SQRTPI	157
5.4.4	Sine: $\sin(z)$	158
5.4.5	Cosine: $\cos(z)$	160
5.4.6	Tangent: $\tan(z)$	162
5.4.7	Secant: $\sec(z) = 1/\cos(z)$	164
5.4.8	Cosecant: $\csc(z) = 1/\sin(z)$	166
5.4.9	Cotangent: $\cot(z) = 1/\tan(z)$	168
5.4.10	Sinc function	170
5.4.11	Trigonometric functions with modified argument	170
5.5	Hyperbolic Functions	172
5.5.1	Hyperbolic Sine: $\sinh(z)$	172
5.5.2	Hyperbolic Cosine: $\cosh(z)$	174
5.5.3	Hyperbolic Tangent: $\tanh(z)$	176
5.5.4	Hyperbolic Secant: $\operatorname{sech}(z) = 1/\cosh(z)$	178
5.5.5	Hyperbolic Cosecant: $\operatorname{csch}(z) = 1/\sinh(z)$	180
5.5.6	Hyperbolic Cotangent: $\operatorname{coth}(z) = 1/\tanh(z)$	182
5.6	Inverse Trigonometric Functions	184
5.6.1	Arcsine: $\operatorname{asin}(z)$	184
5.6.2	Arccosine: $\operatorname{acos}(z)$	186
5.6.3	Arctangent: $\operatorname{atan}(z)$	188
5.6.4	Arc-tangent, version with 2 arguments: $\operatorname{atan2}(x, y)$	190
5.6.5	Arccotangent: $\operatorname{acot}(z)$	191
5.6.6	$\operatorname{asec}(x)$	191
5.6.7	$\operatorname{acsc}(x)$	192
5.7	Inverse Hyperbolic Functions	193
5.7.1	Inverse Hyperbolic Sine: $\operatorname{asinh}(z)$	193
5.7.2	Inverse Hyperbolic Cosine: $\operatorname{acosh}(z)$	194
5.7.3	Inverse Hyperbolic Tangent: $\operatorname{atanh}(z)$	195

5.7.4	Inverse Hyperbolic Cotangent: $\text{acoth}(z)$	196
5.7.5	$\text{asech}(x)$	197
5.7.6	$\text{acsch}(x)$	197
5.8	Elementary Functions of Mathematical Physics	198
5.8.1	Bessel Function $J_\nu(x)$	198
5.8.2	Bessel Function $Y_\nu(x)$	198
5.8.3	Bessel Function $I_\nu(x)$	198
5.8.4	Bessel Function $K_\nu(x)$	199
5.8.5	Complementary Error Function	199
5.8.6	Gamma function $\Gamma(x)$	200
5.8.7	Beta Function $B(a,b)$	201
5.9	Factorial and Related Functions	202
5.9.1	Factorial	202
5.9.2	Double Factorial	202
5.9.3	Binomial Coefficient, Combinations	202
5.9.4	Multinomial	203
5.9.5	Permutations	203
5.9.6	Greatest Common Divisor (GCD)	203
5.9.7	Least Common Multiple (LCM)	204
6	Linear Algebra	205
6.1	Multiple Linear Regression	205
6.1.1	Determinant	205
6.1.2	Inverse	205
6.1.3	LinEst	206
6.1.4	TREND	206
6.2	Exponential Growth Curves	207
6.2.1	LogEst	207
6.2.2	Growth	207
6.3	Norms	208
6.4	Decompositions	210
6.5	Linear Equations	212
6.6	Matrix Factorization	214
7	Distribution Functions	216
7.1	Introduction to Distribution Functions	216
7.1.1	Continuous Distribution Functions	216
7.1.2	Discrete Distribution Functions	216
7.1.3	Commonly Used Function Types	217
7.2	Beta-Distribution	223
7.2.1	Definition	223
7.2.2	Density and CDF	223

7.2.3	Quantiles	224
7.2.4	Properties	225
7.2.5	Random Numbers	226
7.3	Binomial Distribution	228
7.3.1	Density and CDF	228
7.3.2	Quantiles	229
7.3.3	Properties	230
7.3.4	Random Numbers	231
7.4	Chi-Square Distribution	232
7.4.1	Definition	232
7.4.2	Density and CDF	232
7.4.3	Quantiles	234
7.4.4	Properties	236
7.4.5	Random Numbers	237
7.4.6	Wishart Matrix	237
7.5	Exponential Distribution	238
7.5.1	Density and CDF	238
7.5.2	Quantiles	239
7.5.3	Properties	239
7.5.4	Random Numbers	240
7.6	Fisher's F-Distribution	241
7.6.1	Definition	241
7.6.2	Density and CDF	241
7.6.3	Quantiles	242
7.6.4	Properties	245
7.6.5	Random Numbers	245
7.7	Gamma (and Erlang) Distribution	247
7.7.1	Density and CDF	247
7.7.2	Quantiles	248
7.7.3	Properties	249
7.7.4	Random Numbers	249
7.8	Hypergeometric Distribution	251
7.8.1	Definition	251
7.8.2	Density and CDF	251
7.8.3	Quantiles	252
7.8.4	Properties	253
7.8.5	Random Numbers	253
7.9	Lognormal Distribution	254
7.9.1	Definition	254
7.9.2	Density and CDF	254
7.9.3	Quantiles	255
7.9.4	Properties	256
7.9.5	Random Numbers	257
7.10	Negative Binomial Distribution	258

7.10.1	Density and CDF	258
7.10.2	Quantiles	259
7.10.3	Properties	259
7.10.4	Random Numbers	260
7.11	Normal Distribution	261
7.11.1	Definition	261
7.11.2	Density and CDF	261
7.11.3	Quantiles	263
7.11.4	Properties	264
7.11.5	Random Numbers	265
7.12	Poisson Distribution	266
7.12.1	Definition	266
7.12.2	Density and CDF	266
7.12.3	Quantiles	267
7.12.4	Properties	267
7.12.5	Random Numbers	268
7.13	Student's t-Distribution	269
7.13.1	Definition	269
7.13.2	Density and CDF	269
7.13.3	Quantiles	271
7.13.4	Properties	272
7.13.5	Random Numbers	273
7.13.6	Behrens-Fisher Problem	273
7.14	Weibull Distribution	274
7.14.1	Density and CDF	274
7.14.2	Quantiles	275
7.14.3	Properties	275
7.14.4	Random Numbers	276
8	Statistical Functions	277
8.1	Frequencies and Percentages	277
8.1.1	Count, CountA	277
8.1.2	Frequency and 1D Histogram	278
8.2	Transformations	280
8.2.1	Linear Transformations (CONVERT)	280
8.2.2	Standardization	282
8.2.3	Trimming and Winsorizing	282
8.3	Sums, Means, Moments and Cumulants	284
8.3.1	Sum	284
8.3.2	Arithmetic Mean	285
8.3.3	Geometric Mean	286
8.3.4	Harmonic Mean	286
8.3.5	Variance	286
8.3.6	Standard Deviation	288

8.3.7	Average Deviation	289
8.3.8	Sum of Squares of Deviations (DEVSQ)	289
8.3.9	Skewness	290
8.3.10	Kurtosis	291
8.4	Min, Max, Median, Percentiles	292
8.4.1	Minimum	292
8.4.2	Maximum	292
8.4.3	Median	292
8.4.4	Mode	293
8.4.5	<i>K</i> -th Largest Number	293
8.4.6	<i>K</i> -th Smallest Number	294
8.4.7	Percentile	294
8.4.8	PercentRank	295
8.4.9	Quartile	295
8.4.10	Rank	296
8.4.11	Prob	297
8.5	Summary Tables of Aggregates	298
8.5.1	SUBTOTAL	298
8.5.2	AGGREGATE	298
8.6	Confidence intervals and tests	300
8.6.1	Confidence Interval for the mean, with known variance	300
8.6.2	Confidence Interval for the mean, with unknown variance	300
8.6.3	Gauss z-Tests	301
8.6.4	Student's t-Test, 2 samples	301
8.6.5	F-Test (variances of 2 independent samples)	302
8.6.6	Anova: Single Factor	302
8.6.7	Anova: Two Factors (with or without replication)	303
8.6.8	Chi-Square-Test (Homogeneity)	303
8.7	Covariance and Correlation	305
8.7.1	Covariance	305
8.7.2	Correlation	305
8.7.3	Fisher's z-transform	306
8.8	Linear Regression	308
8.8.1	INTERCEPT	308
8.8.2	SLOPE	308
8.8.3	Forecast	308
8.8.4	SteXY	309
8.9	Database related functions	310
8.9.1	DGET	310
8.9.2	DPRODUCT	310
8.9.3	DCount, DCountA	310
8.9.4	DSum	311
8.9.5	DAverage	311
8.9.6	Variance	312
8.9.7	Standard Deviation	312

8.9.8	Minimum	313
8.9.9	Maximum	313

III Special Functions 314

9 Factorials and gamma functions	315
9.1 Factorials	315
9.1.1 Factorial	315
9.1.2 Double factorial	316
9.2 Binomial coefficient	318
9.3 Pochhammer symbol, Rising and falling factorials	319
9.3.1 Relative Pochhammer symbol	319
9.3.2 Rising factorial	319
9.3.3 Falling factorial	320
9.4 Super- and hyperfactorials	321
9.4.1 Superfactorial	321
9.4.2 Hyperfactorial	322
9.4.3 Barnes G-function	323
9.5 Gamma functions	324
9.5.1 Gamma function	324
9.5.2 Reciprocal of the gamma function	325
9.5.3 The product / quotient of gamma functions	325
9.5.4 The log-gamma function	326
9.5.5 Generalized incomplete gamma function	327
9.5.6 Derivative of the normalised incomplete gamma function	328
9.5.7 Normalised incomplete gamma functions	328
9.5.8 Non-Normalised incomplete gamma functions	329
9.5.9 Tricomi's entire incomplete gamma function	329
9.5.10 Inverse normalised incomplete gamma functions	330
9.6 Polygamma functions and harmonic numbers	331
9.6.1 Polygamma function	331
9.6.2 Digamma function	332
9.6.3 Harmonic numbers	332
9.7 Beta Functions	333
9.7.1 Beta function $B(a, b)$	333
9.7.2 Beta function	333
9.7.3 Logarithm of $B(a, b)$	333
9.7.4 Generalized incomplete beta function	333
9.7.5 Non-Normalised incomplete beta functions	334
9.7.6 Normalised incomplete beta functions	335

10 Exponential integrals and error functions	336
10.1 Exponential integrals	336
10.1.1 Exponential integral Ei	336
10.1.2 Exponential integral E1	337
10.1.3 Generalized exponential integral En	337
10.1.4 Generalized Exponential Integrals Ep	338
10.2 Logarithmic integral	339
10.2.1 logarithmic integral li	339
10.3 Trigonometric integrals	340
10.3.1 cosine integral ci	340
10.3.2 sine integral si	340
10.4 Hyperbolic integrals	342
10.4.1 hyperbolic cosine integral chi	342
10.4.2 hyperbolic sine integral shi	342
10.5 Error functions	343
10.5.1 Error Function	343
10.5.2 Complementary Error Function	344
10.5.3 Imaginary Error Function	344
10.5.4 Inverse Error Function	345
10.6 The normal distribution	347
10.6.1 The normal probability density function	347
10.6.2 The normal cumulative distribution function	347
10.7 Fresnel integrals	348
10.7.1 Fresnel sine integral	348
10.7.2 Fresnel cosine integral	348
10.8 Other Special Functions	350
10.8.1 Lambert W function	350
10.8.2 Arithmetic-geometric mean	351
11 Bessel functions and related functions	352
11.1 Bessel functions	352
11.1.1 Exponentially scaled Bessel function $I_{\nu,e}(x)$	352
11.1.2 Exponentially scaled Bessel function $K_{\nu,e}(x)$	352
11.1.3 Bessel function of the first kind	352
11.1.4 Bessel function of the second kind	354
11.1.5 Modified Bessel function of the first kind	355
11.1.6 Modified Bessel function of the second kind	356
11.2 Bessel function zeros	358
11.2.1 Zeros of the Bessel function of the first kind	358
11.2.2 Zeros of the Bessel function of the second kind	359
11.3 Hankel functions	361
11.3.1 Hankel function of the first kind	361

11.3.2	Hankel function of the second kind	361
11.4	Kelvin functions	362
11.4.1	Kelvin function ber	362
11.4.2	Kelvin function bei	362
11.4.3	Kelvin function ker	362
11.4.4	Kelvin function kei	363
11.5	Struve Functions	364
11.5.1	Struve function H	364
11.5.2	Modified Struve function L	365
11.6	Anger-Weber functions	366
11.6.1	Anger function J	366
11.6.2	Weber function E	366
11.7	Lommel functions	368
11.7.1	First Lommel function s	368
11.7.2	Second Lommel function S	368
11.8	Airy and Scorer functions	370
11.8.1	Airy function Ai	370
11.8.2	Airy function Bi	371
11.8.3	Zeros of the Airy function Ai	373
11.8.4	Zeros of the Airy function Bi	373
11.8.5	Scorer function Gi	374
11.8.6	Scorer function Hi	374
11.9	Coulomb wave functions	376
11.9.1	Regular Coulomb wave function	376
11.9.2	Irregular Coulomb wave function	377
11.9.3	Normalizing Gamow constant	377
11.10	Parabolic cylinder functions	379
11.10.1	Parabolic cylinder function D	379
11.10.2	Parabolic cylinder function U	379
11.10.3	Parabolic cylinder function V	380
11.10.4	Parabolic cylinder function W	381
12	Orthogonal polynomials	382
12.1	Legendre functions	382
12.1.1	Legendre polynomial	382
12.1.2	Associated Legendre function of the first kind	383
12.1.3	Associated Legendre function of the second kind	384
12.1.4	Spherical harmonics	385
12.2	Chebyshev polynomials	387
12.2.1	Chebyshev polynomial of the first kind	387
12.2.2	Chebyshev polynomial of the second kind	387
12.3	Jacobi and Gegenbauer polynomials	389
12.3.1	Jacobi polynomial	389

12.3.2	Zernike Radial Polynomials	390
12.3.3	Gegenbauer polynomial	390
12.4	Hermite and Laguerre polynomials	392
12.4.1	Hermite polynomials	392
12.4.2	Laguerre polynomials	393
12.4.3	Laguerre Polynomials	394
12.4.4	Associated Laguerre Polynomials	394
13	Hypergeometric functions	395
13.1	Confluent Hypergeometric Limit Function	396
13.1.1	Confluent Hypergeometric Limit Function	396
13.1.2	Regularized Confluent Hypergeometric Limit Function	397
13.2	Kummer's Confluent Hypergeometric Functions and related functions	398
13.2.1	Kummer's Confluent Hypergeometric Function of the first kind	398
13.2.2	Kummer's Regularized Confluent Hypergeometric Function	399
13.2.3	Kummer's Confluent Hypergeometric Function of the second kind	400
13.2.4	Hypergeometric Function 2F0	400
13.3	Whittaker functions M and W	402
13.3.1	Whittaker function M	402
13.3.2	Whittaker function W	402
13.4	Gauss Hypergeometric Function	404
13.4.1	Gauss Hypergeometric Function	404
13.4.2	Gauss Regularized Hypergeometric Function	406
13.5	Additional Hypergeometric Functions	407
13.5.1	Hypergeometric Function 1F2	407
13.5.2	Hypergeometric Function 2F2	407
13.5.3	Hypergeometric Function 2F3	408
13.5.4	Hypergeometric Function 3F2	408
13.6	Generalized hypergeometric functions	410
13.6.1	Generalized hypergeometric function pFq	410
13.6.2	Weighted combination of hypergeometric functions	411
13.7	Meijer G-function	413
13.8	Bilateral hypergeometric series	416
13.9	Hypergeometric functions of two variables	417
13.9.1	Generalized 2D hypergeometric series	417
13.9.2	Appell F1 hypergeometric function	418
13.9.3	Appell F2 hypergeometric function	419
13.9.4	Appell F3 hypergeometric function	419
13.9.5	Appell F4 hypergeometric function	420
14	Elliptic functions	422
14.1	Elliptic arguments	422

14.2	Legendre elliptic integrals	425
14.2.1	Complete elliptic integral of the first kind	425
14.2.2	Incomplete elliptic integral of the first kind	425
14.2.3	Complete elliptic integral of the second kind	427
14.2.4	Incomplete elliptic integral of the second kind	427
14.2.5	Complete elliptic integral of the third kind	428
14.2.6	Incomplete elliptic integral of the third kind	429
14.3	Carlson symmetric elliptic integrals	431
14.3.1	Symmetric elliptic integral of the first kind, RF	431
14.3.2	Degenerate Carlson symmetric elliptic integral of the first kind, RC	432
14.3.3	Symmetric elliptic integral of the third kind, RJ	433
14.3.4	Symmetric elliptic integral of the second kind, RD	434
14.3.5	Completely symmetric elliptic integral of the second kind, RG	434
14.4	Jacobi theta functions	436
14.5	Jacobi elliptic functions	438
14.6	Klein j-invariant	440
15	Zeta functions, L-series and polylogarithms	441
15.1	Riemann and Hurwitz zeta functions	441
15.2	Dirichlet L-series	443
15.2.1	Dirichlet eta function	443
15.2.2	Dirichlet $\eta(s) - 1$	443
15.2.3	Dirichlet Beta Function	444
15.2.4	Dirichlet Lambda Function	444
15.2.5	Dirichlet L-function	444
15.3	Stieltjes constants	446
15.4	Zeta function zeros	447
15.5	Riemann-Siegel Z function and related functions	449
15.5.1	Riemann-Siegel Z	449
15.5.2	Riemann-Siegel theta function	449
15.5.3	Gram point (Riemann-Siegel Z function)	450
15.5.4	Backlunds function	450
15.6	Lerch transcendent and related functions	452
15.6.1	Fermi-Dirac integrals of integer order	453
15.6.2	Fermi-Dirac integral $F_{-1/2}(x)$	453
15.6.3	Fermi-Dirac integral $F_{1/2}(x)$	453
15.6.4	Fermi-Dirac integral $F_{3/2}(x)$	453
15.6.5	Legendre Chi-Function	454
15.6.6	Inverse Tangent Integral	454
15.7	Polylogarithms and Clausen functions	455
15.7.1	Polylogarithm	455
15.7.2	Dilogarithm Function	456

15.7.3	Debye Functions	456
15.7.4	Clausen sine function	456
15.7.5	Clausen cosine function	457
15.7.6	Polyexponential function	458
15.8	Zeta function variants	459
15.8.1	Prime zeta function	459
15.8.2	Secondary zeta function	459
16	Number-theoretical, combinatorial and integer functions	461
16.1	Fibonacci numbers	461
16.2	Bernoulli numbers and polynomials	463
16.2.1	Bernoulli numbers	463
16.2.2	Bernoulli polynomials	465
16.3	Euler numbers and polynomials	467
16.3.1	Euler numbers	467
16.3.2	Euler polynomials	467
16.4	Bell numbers and polynomials	469
16.5	Stirling numbers	470
16.5.1	Stirling number of the first kind	470
16.5.2	Stirling number of the second kind	470
16.6	Prime counting functions	472
16.6.1	Exact prime counting function	472
16.6.2	Prime counting function interval	472
16.6.3	Riemann R function	473
16.7	Miscellaneous functions	475
16.7.1	Cyclotomic polynomials	475
16.7.2	von Mangoldt function	476
17	q-functions	477
17.1	q-Pochhammer symbol	477
17.2	q-gamma and factorial	478
17.2.1	q-gamma	478
17.2.2	q-factorial	478
17.3	Hypergeometric q-series	480
18	Matrix functions	481
18.1	Matrix exponential	481
18.2	Matrix cosine	483
18.3	Matrix sine	484

18.4	Matrix square root	485
18.5	Matrix logarithm	487
18.6	Matrix power	489
19	Eigensystems and related Decompositions	491
19.1	Singular value decomposition	491
19.2	The Schur decomposition	492
19.3	The eigenvalue problem	493
19.4	The symmetric eigenvalue problem	494
IV	Numerical Calculus	495
20	Polynomials	496
20.1	Polynomial evaluation	496
20.2	Polynomial roots	497
21	Root-finding and optimization	499
21.1	Root-finding	499
21.2	Solvers	503
21.2.1	Secant	503
21.2.2	Newton	503
21.2.3	MNewton	503
21.2.4	Halley	503
21.2.5	Muller	503
21.2.6	Bisection	503
21.2.7	Illinois	504
21.2.8	Pegasus	504
21.2.9	Anderson	504
21.2.10	Ridder	504
21.2.11	ANewton	504
21.2.12	MDNewton	504
22	Sums, products, limits and extrapolation	506
22.1	Summation	506
22.1.1	One-dimensional Summation	506
22.1.2	Two-dimensional Summation	512
22.1.3	Three-dimensional Summation	513
22.1.4	Euler-Maclaurin formula	514
22.1.5	Abel-Plana formula	515
22.2	Products	517

22.3	Limits	520
22.4	Extrapolation	522
22.4.1	Richardson's algorithm	522
22.4.2	Shanks' algorithm	523
22.4.3	Levin's algorithm	525
22.4.4	Cohen's algorithm	529
23	Differentiation	531
23.1	Numerical derivatives	531
23.1.1	One-dimensional Differentiation	531
23.1.2	Sequence of derivatives	533
23.2	Composition of derivatives	534
23.2.1	Differentiation of products of functions	534
23.2.2	Differentiation of the exponential of functions	534
23.3	Fractional derivatives	535
24	Numerical integration (quadrature)	537
24.1	Standard quadrature	537
24.1.1	One-dimensional Integration	537
24.1.2	Two-dimensional Integration	540
24.1.3	Three-dimensional Integration	541
24.1.4	Oscillatory integrals	542
24.2	Main Quadrature rules	545
24.2.1	TanhSinh	545
24.2.2	Gauss-Legendre	546
24.3	Additional Quadrature rules	547
24.3.1	Gauss-Legendre	547
24.3.2	Gauss-Hermite	547
24.3.3	Gauss-Laguerre	547
25	Ordinary differential equations	548
25.1	Solving the ODE initial value problem	548
26	Function approximation	551
26.1	Taylor series	551
26.2	Pade approximation	552
26.3	Chebyshev approximation	553
26.4	Fourier series	555

27 Number identification	557
27.1 Constant recognition	557
27.1.1 Examples	558
27.1.2 Finding approximate solutions	559
27.1.3 Symbolic processing	560
27.1.4 Miscellaneous issues and limitations	561
27.2 Algebraic identification	562
27.2.1 Examples	562
27.2.2 Non-algebraic numbers	563
27.3 Integer relations (PSLQ)	564
27.3.1 Examples	564
27.3.2 Machin formulas	565
V Application Examples	566
28 Date, Time and Financial Functions	567
28.1 Date and Time: Conversions from Serial Number	567
28.1.1 Serial Number to Second	567
28.1.2 Serial Number to Minute	568
28.1.3 Serial Number to Hour	568
28.1.4 Serial Number to Day of the Month	568
28.1.5 Number of days between two dates	569
28.1.6 Serial Number to Month	569
28.1.7 Serial Number to Year	569
28.1.8 Serial Number to a Day of the Week	569
28.1.9 Serial Number to Calendar Week	570
28.2 Date and Time: Conversions to Serial Number	571
28.2.1 Serial Number of a particular Date	571
28.2.2 Serial Number of Easter Sunday	571
28.2.3 Date as Text to Serial Number	571
28.2.4 Serial Number of Months before or after Start Date	572
28.2.5 Serial Number of the last day of the months	572
28.2.6 Serial Number of the current date and time	573
28.2.7 Serial Number of a particular Time	573
28.2.8 Time as Text to Serial Number	573
28.2.9 Serial Number of today's Date	573
28.2.10 Serial Number of Date +/- n Workdays	574
28.2.11 Serial Number of Date +/- n Workdays, international	574
28.3 Date and Time: Calculations	576
28.3.1 Number of whole workdays between two dates	576
28.3.2 Number of whole workdays between two dates, international	577
28.3.3 Year Fraction representing whole days between 2 Dates	578
28.4 Coupons	579
28.4.1 Days from Beginning to Settlement Date	579

28.4.2	Days in Coupon Period containing the Settlement Date	580
28.4.3	Days from Settlement Date to next Coupon Date	580
28.4.4	Next Coupon Date after the Settlement Date	580
28.4.5	Coupons payable between Settlement and Maturity Date	581
28.4.6	Previous Coupon Date before the Settlement Date	581
28.4.7	Macauley Duration for an assumed par Value of 100	581
28.4.8	Modified Macauley Duration	582
28.5	Securities	583
28.5.1	Accrued Interest	583
28.5.2	Accrued Interest at Maturity	584
28.5.3	Discount Rate for a Security	584
28.5.4	Interest Rate for a fully invested Security	585
28.5.5	Price of a Security having an odd first Period	585
28.5.6	Yield of a Security that has an odd first Period	586
28.5.7	Price of a Security having an odd last Coupon	587
28.5.8	Yield of a Security that has an odd last Period	587
28.5.9	Price of a Security that pays periodic Interest	588
28.5.10	Price of a discounted Security	589
28.5.11	Price of a Security that pays Interest at Maturity	589
28.5.12	Amount received at Maturity for a fully invested Security	590
28.5.13	Yield on a Security that pays periodic Interest	590
28.5.14	Annual Yield for a discounted Security	591
28.5.15	Annual Yield of a Security that pays Interest at Maturity	591
28.6	Treasury Bills	592
28.6.1	Bond-equivalent Yield for a Treasury bill	592
28.6.2	Price for a Treasury bill	592
28.6.3	Yield for a Treasury bill	592
28.7	Depreciation Functions	594
28.7.1	Depreciation of an Asset	594
28.7.2	Straight-Line Depreciation of an Asset	594
28.7.3	Sum-of-Years' Digits Depreciation of an Asset	595
28.7.4	Fixed Declining Balance Method	595
28.7.5	Variable Declining Balance	596
28.7.6	Depreciation for each accounting period	596
28.7.7	Depreciation using a depreciation coefficient	597
28.8	Annuity Functions	598
28.8.1	Future Value	598
28.8.2	Present Value	598
28.8.3	Payment	599
28.8.4	Number of periods	599
28.8.5	Number of periods required	600
28.8.6	Interest Rate	600
28.8.7	Interest Payment	601
28.8.8	Principal Payment	601
28.8.9	Cumulative Interest Paid	601
28.8.10	Cumulative Principal Paid	602

28.8.11	Effective Annual Interest Rate	602
28.8.12	Nominal Annual Interest Rate	602
28.8.13	FV Schedule, variable Compound Interest Rates	603
28.8.14	Interest paid during a specific Period of an Investment	603
28.9	Cash-Flow Functions	604
28.9.1	Internal Rate of Return	604
28.9.2	Calc: Rate of Return	604
28.9.3	Modified Internal Rate of Return	605
28.9.4	Net Present Value	605
28.9.5	Internal Rate of Return, non-periodic Schedule	606
28.9.6	Net Present Value, non-periodic Schedule	606
28.10	Conversion	608
28.10.1	Price as a fraction into a price as decimal	608
28.10.2	Price as a decimal into a price as fraction	608

VI Appendices 609

A Python	610	
A.1	Overview	610
A.2	CPython	611
A.2.1	Downloading and installing CPython 2.7	611
A.2.2	Plotting	612
A.2.3	Using the C-API	616
A.2.4	Interfaces to the C family of languages	616
A.2.5	Cython: C extensions for the Python language	622
A.2.6	A Windows-specific interface: using COM	622
A.3	PyPy: a fast alternative Python interpreter	624
A.3.1	Installation	624
A.4	Jython: Python for the Java Virtual Machine	625
A.4.1	Installing Java	626
A.4.2	Installing Eclipse with support for JPython	626
A.5	IronPython: Python for .NET Framework and Mono	627
A.6	Pythonista: Python for iOS	628
A.7	QPython: Python for Android	629
A.8	Brython: a Python to JavaScript compiler	630

B LibreOffice Calc and Microsoft Excel 631

B.1	LibreOffice Calc	631
B.1.1	Windows	631
B.1.2	Mac OSX	632
B.1.3	GNU/Linux	633

B.1.4	Controlling LibreOffice using sockets or named pipes	633
B.2	Microsoft Excel	635
B.2.1	Windows	635
B.2.2	Mac OSX	635
B.3	Spreadsheet functions implemented in mpFormulaPy	636
B.3.1	Spreadsheet Mathematical Functions	636
B.3.2	Spreadsheet Engineering Functions	639
B.3.3	Spreadsheet Statistical Functions	641
B.3.4	Spreadsheet Revised Statistical Functions	645
B.3.5	Spreadsheet Database Functions	648
B.3.6	Spreadsheet Financial Functions	649
B.3.7	Spreadsheet Date and Time Functions	652
C	Languages with CLR Support	653
C.1	Visual Basic .NET	653
C.1.1	Visual Basic 2005	653
C.1.2	Visual Basic 2008	653
C.1.3	Visual Basic 2010	654
C.1.4	Visual Basic 2012	654
C.1.5	Relation to older versions of Visual Basic	654
C.2	C# 4.0	663
C.3	JScript 10.0	666
C.4	C++ 10.0, Visual Studio	668
C.5	F# 3.0	670
C.6	MatLab (.NET interface)	672
D	Building the library and toolbox	674
D.1	Building and installing the numerical library	674
D.1.1	Downloading and installing mpMath	674
D.1.2	Running tests	674
D.2	Building the documentation and standard interfaces	675
D.2.1	Documentation	675
D.2.2	C interface	675
D.2.3	C++ interface	675
D.2.4	COM interface	675
D.2.5	.NET interface	675
D.3	Building the specific interfaces	676
D.3.1	Excel support via Excel-DNA	676
D.3.2	OpenOffice.org/Apache OpenOffice/LibreOffice Calc support	676
D.4	Building the Toolbox GUI	676
D.4.1	NetOffice	676

D.5	Other Software	677
D.5.1	Downloads	677
D.5.2	Other References	677
D.5.3	Previous version	678
D.6	To Do	679
D.6.1	New GUI	679
D.6.2	GUI	679
D.6.3	C++ Library	680
D.6.4	Manual	680
D.6.5	New Programming	680
E	Acknowledgements	681
E.1	Contributors to libraries used in the numerical routines	681
E.1.1	Contributors to mpMath	681
E.1.2	Contributors to gmpy2	681
E.2	Contributors to libraries used in the GUI	681
E.2.1	Contributors to Sharp Develop	681
E.2.2	Contributors to Unmanaged Exports	682
E.2.3	Contributors to NetOffice	682
E.2.4	Contributors to Excel-DNA	682
E.2.5	Contributors to jni4net	682
E.2.6	System.Data.SQLite	682
F	Licenses	683
F.1	GNU Licenses	683
F.1.1	GNU General Public License, Version 2	683
F.1.2	GNU Library General Public License, Version 2	689
F.1.3	GNU Lesser General Public License, Version 3	695
F.1.4	GNU General Public License, Version 3	697
F.1.5	GNU Free Documentation License, Version 1.3	706
F.2	Other Licenses	712
F.2.1	Mozilla Public License, Version 2.0	712
F.2.2	New BSD License (for mpMath)	717
F.2.3	MIT License	718
F.2.4	Excel-DNA License	719
G	Mathematical Notation	720
Appendices		610
VII	Back Matter	721
Bibliography		722

Nomenclature	727
Index	729

List of Tables

B.1	Spreadsheet Mathematical Functions	636
B.2	Spreadsheet Engineering Functions	639
B.3	Spreadsheet Statistical Functions	641
B.4	Spreadsheet Revised Statistical Functions	645
B.5	Spreadsheet Database Functions	648
B.6	Spreadsheet Financial Functions	649
B.7	Spreadsheet Date and Time Functions	652
G.1	Notation related to Sets	720

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	137
5.2	Complex Logarithm	141
5.3	Complex Square	145
5.4	Complex Cube	146
5.5	Complex Square Root	149
5.6	Complex Sine	158
5.7	Complex Cosine	160
5.8	Complex Tangent	162
5.9	Complex Secant	164
5.10	Complex Cosecant	166
5.11	Complex Cotangent	168
5.12	Complex Hyperbolic Sine	172
5.13	Complex Hyperbolic Cosine	174
5.14	Complex Hyperbolic Tangent	176
5.15	Complex Hyperbolic Secant	178
5.16	Complex Hyperbolic Cosecant	180
5.17	Complex Hyperbolic Cotangent	182
5.18	Complex Arcsine	184
5.19	Complex Arccosine	186
5.20	Complex Arctangent	188
5.21	Complex Arccotangent	191
5.22	Complex Inverse Hyperbolic Sine	193
5.23	Complex Inverse Hyperbolic Cosine	194
5.24	Complex Inverse Hyperbolic Tangent	195
5.25	Complex Inverse Hyperbolic Cotangent	196
A.1	MpMath 2Dplot	612
A.2	MpMath cplot	614
A.3	MpMath Surface plot	615

Part I

Getting Started

Chapter 1

Introduction

1.1 Overview: Features and Setup

1.1.1 Features

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

1.1.2 The mpFormulaPy Library

The mpFormulaPy 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 Python) and is provided in the form of source code in Python.

The following numerical types are supported:

- The conventional double (64 bit) precision binary floating point type (double in C).
- The mpf arbitrary precision binary floating point type of the mpmath library.
- The mpi arbitrary precision interval arithmetic binary floating point type of the mpmath library.
- The mpc arbitrary precision complex binary floating point type of the mpmath library.
- The mpci arbitrary precision complex interval arithmetic binary floating point type of the mpmath library.
- The long arbitrary precision integer type of the Python library.
- The Fraction arbitrary precision rational type of the Python library.
- The Decimal arbitrary precision decimal floating point type of the Python library.

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

The mpFormulaPy Library is based on mpmath [Johansson *et al.* \(2013\)](#), and the standard Python Library, including fractions.py and decimal.py.

1.1.3 The mpFormulaPy Toolbox

The mpFormulaPy Toolbox provides a setup with the Ironpython compiler 2.7.4 for the Windows platform with multiple interfaces:

- A .NET Framework 4.0 interface: 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 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 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.

In addition, the mpFormulaPy Toolbox offers the following features:

- A compact IDE for VB.NET and C# (no need to install Visual Studio). The IDE provides a Code Editor, Windows Forms Designer, Debugger, Profiler, Unit Tester and Microsoft Visual Studio compatible project files. Based on a trimmed version of Sharp Develop ([Krüger et al., 2013](#)).
- A Microsoft Excel (versions 2000 to 2013) interface: multiprecision arithmetic functions are provided for use in spreadsheet cells, using Excel's XLL interface. It is also possible to write user-defined functions and procedures in multiprecision arithmetic (provided as add-ins written in VB.NET or C#), which run even if macro security is set to "disable all macros". Based on Excel-DNA ([van Drimmelen, 2013](#)).
- Full access for VB.NET and C# to the object model of MS Excel (including Intellisense support in the Code Editor). All Microsoft Excel versions from 2000 to 2013, both 32 bit and 64 bit (only 2010 and 2013), are supported, and no other software (like PIAs) is needed. Based on NetOffice ([Lange, 2012](#)).
- An OpenOffice Calc (versions 2.1 or later, incl. Apache OpenOffice Calc or LibreOffice Calc) interface: multiprecision arithmetic functions are provided for use in spreadsheet cells, using the OpenOffice Basic interface. It is also possible to write user-defined functions and procedures in multiprecision arithmetic (provided as add-ins written in VB.NET or C#).

1.1.4 System Requirement

This mpFormulaPy Toolbox has the following system requirement:

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

This mpFormulaPy Toolbox can take advantage of the following software:

- Adobe Reader 8 or later, for use with mpFormulaPy’s interactive help-system.
- Microsoft Office 2000 or later.
- OpenOffice (versions 2.1 or later), or Apache OpenOffice or LibreOffice.

1.1.5 Installation

The mpFormulaPy Library and Toolbox can be downloaded from

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

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

1.2 License

The mpFormulaPy Library and Toolbox is free software. It is licensed under the GNU Lesser General Public License (LGPL), Version 3 (see appendix [F.1.3](#)). The manual for the mpFormulaPy 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 Lesser General Public License, Version 3 (see appendix [F.1.3](#)) for details.

1.4 Related Software

The mpFormulaC Library and Toolbox provides fast multiprecision routines written in C, with interfaces to CPython, R, .NET and COM. It can be downloaded from <http://mpFormula.github.io/C/>.

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 2×2 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.

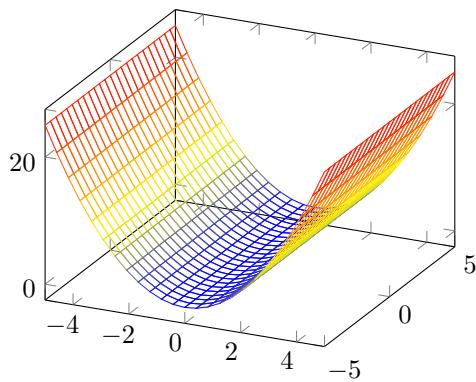


Figure 2.1: A pdf plot

2.3 Graphics using .NET Framework

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

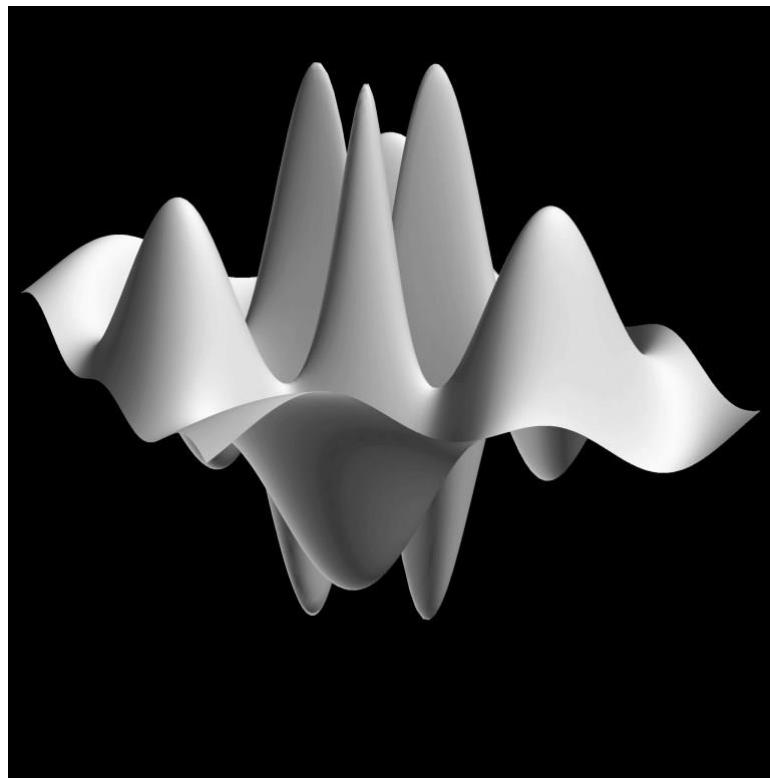


Figure 2.2: plot of a 2-dimensional function

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:

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

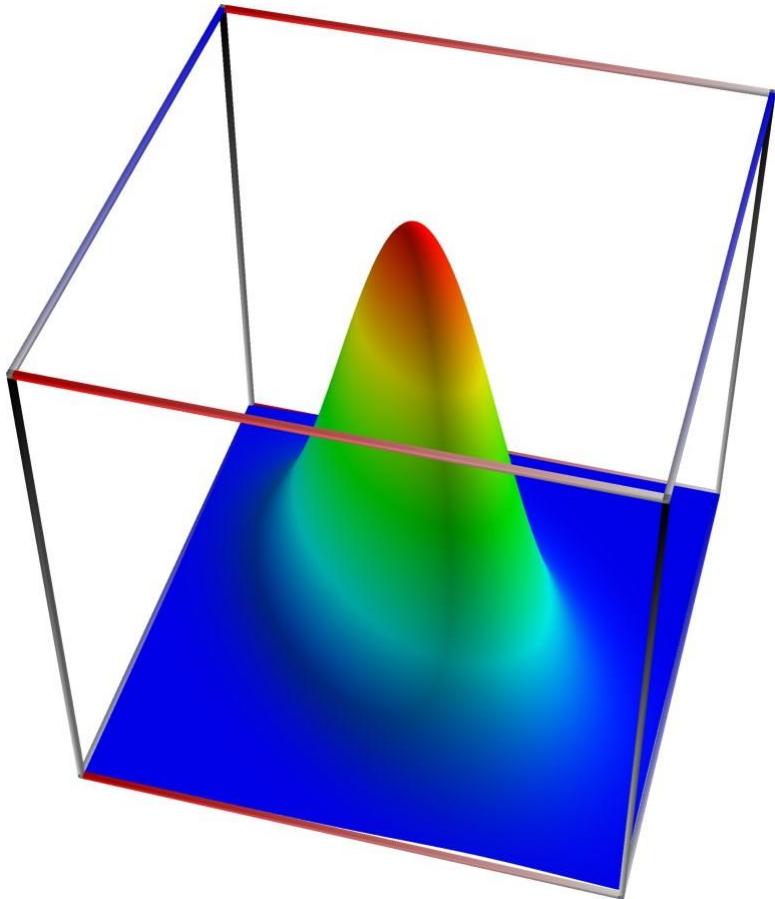


Figure 2.3: Surface plot of the probability density function of the bivariate normal distribution with $\rho = -0.5$

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.

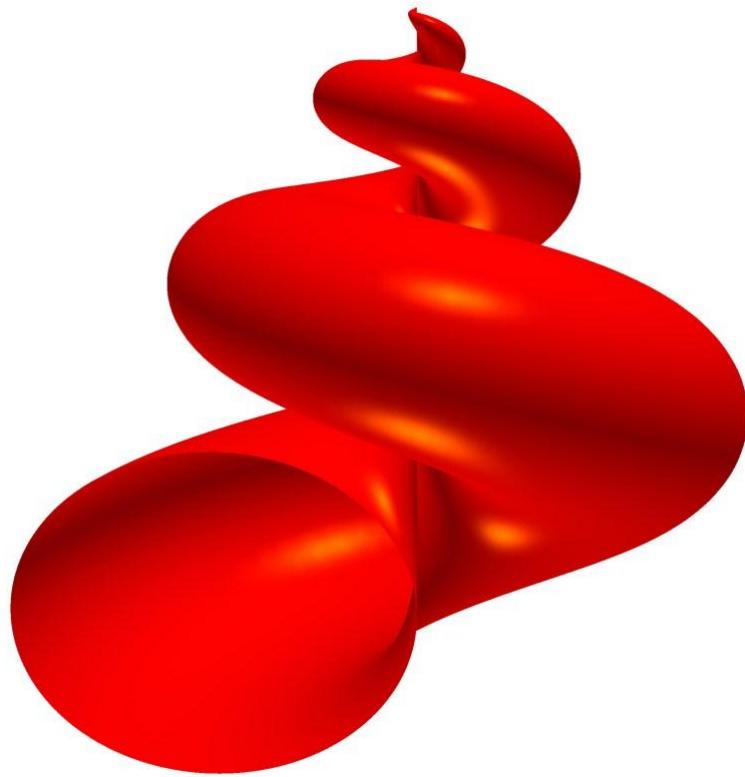


Figure 2.4: 3D plot of a parametric function: Seashell. $u_{\min} = 0$; $u_{\max} = 6 * \text{Math.PI}$; $v_{\min} = 0$; $v_{\max} = 6 * \text{Math.PI}$. Camera angles are $\theta = 135^\circ$ and $\phi = -12^\circ$.

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.

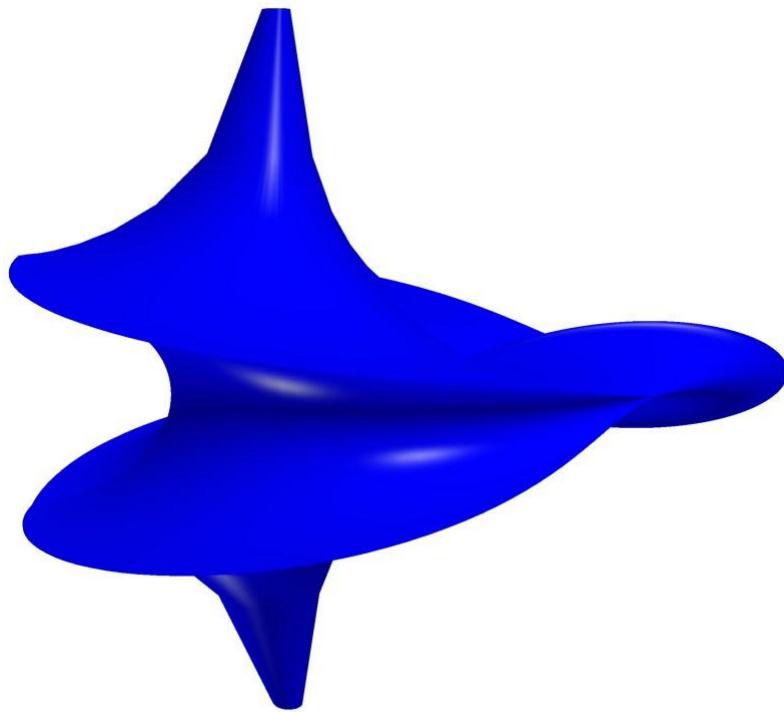


Figure 2.5: 3D plot of a parametric function: Kuen's surface. $u_{\min} = -4.5$; $u_{\max} = 4.5$; $v_{\min} = 0.01$; $v_{\max} = 3.14$. Camera angles are $\theta = 135^\circ$ and $\phi = -12^\circ$.

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.

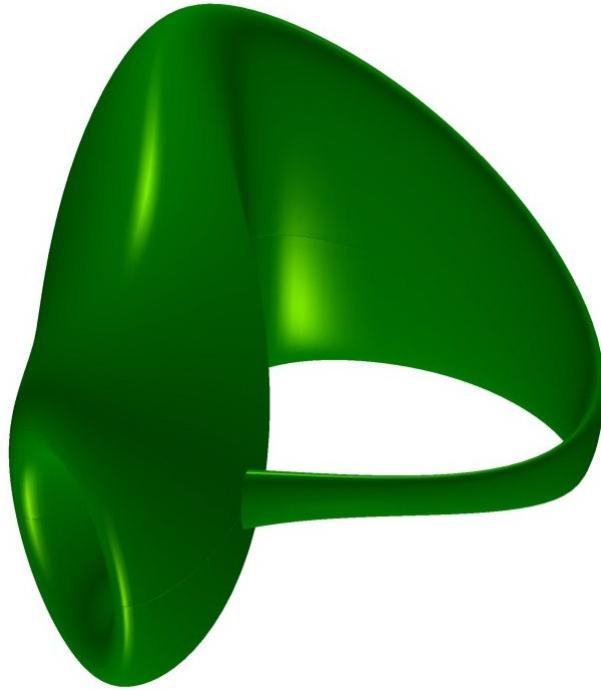


Figure 2.6: 3D plot of a parametric function: Klein's Bottle. $u_{\min} = 0.0$; $u_{\max} = 3.14$; $v_{\min} = 0.0$; $v_{\max} = 6.28$. Camera angles are $\theta = 135^\circ$ and $\phi = -12^\circ$.

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:

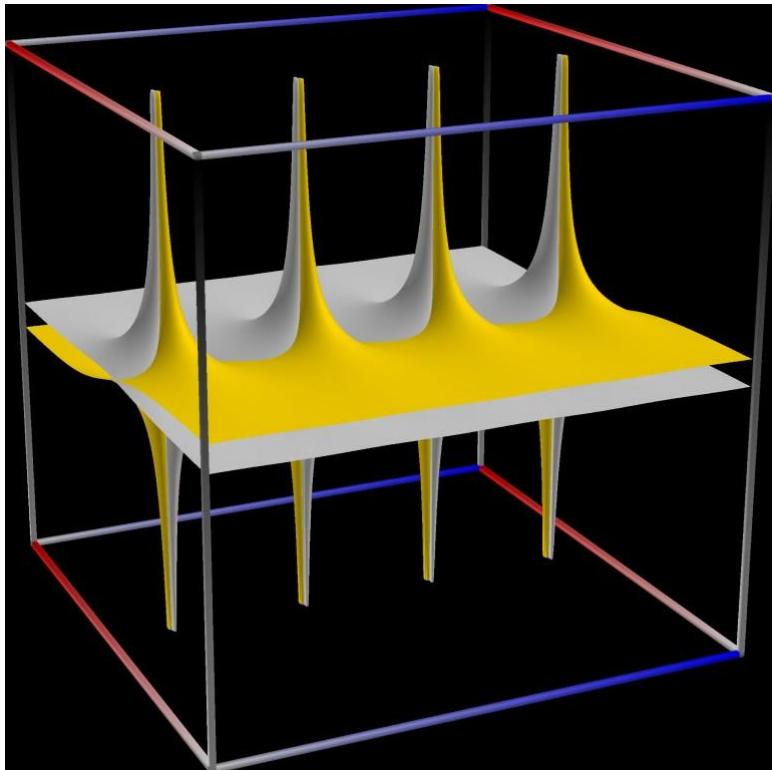


Figure 2.7: Surface plot of the real ("silver") and imaginary ("gold") component of $z = \tan(x + iy)$, $-3 \leq x \leq 3$ (blue axis), $-2\pi \leq y \leq 2\pi$ (red axis), $-10 \leq z \leq 10$ (black axis). z values are truncated at ± 10 . There is a branch cut along the negative real axis. Camera angles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

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

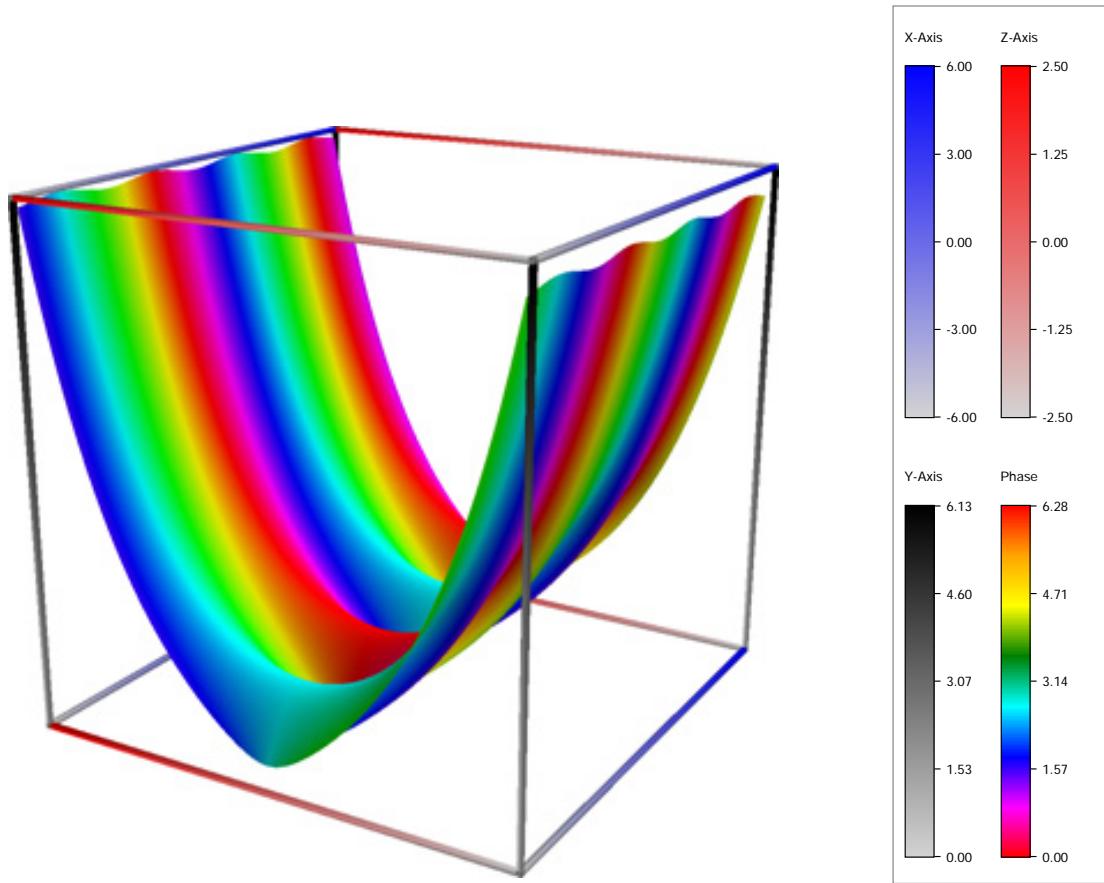


Figure 2.8: Surface plot of the magnitude and phase (color-coded) of $z = \sin(x+iy)$, $-3 \leq x \leq 3$ (blue axis), $-2\pi \leq y \leq 2\pi$ (red axis), $-10 \leq z \leq 10$ (black axis). z values are truncated at ± 10 . There is a branch cut along the negative real axis. Camera angles are $\theta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

2.4 Eval, Options, Tables and Charts

The following functions provide quick access to function evaluations and charts:

Function **Eval**(*Expression* As String) As String

The function **Eval** returns the result of the evaluation of an arithmetic expression, containing number and functions, but no variables.

Parameter:

Expression: an arithmetic expression.

Function **Options**(*BaseOptions* As String) As String

The function **Options** returns an identifier for a set of calculation options.

Parameter:

BaseOptions: an identifier for a set of base calculation options.

Function **Table**(*TableRef* As String) As Range

The function **Table** returns an identifier for a set of calculation options.

Parameter:

TableRef: a reference for a table.

Function **Chart**(*Data* As Range) As String

The function **Chart** returns an identifier for an XML Chart.

Parameter:

Data: a reference for a data table.

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
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	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 `i`, `i=`, `i` and `i=` 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 & y</code>	bitwise and of x and y	
<code>x << n</code>	x shifted left by n bits	(1)(2)
<code>x >> 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 $x.\text{bit_length}()$ 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 $x.\text{bit_length}()$ 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 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 $\lceil \text{self} \rceil$. 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 $\lfloor \text{self} \rfloor$. 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.

3.7 Decimals

3.7.1 Overview

The decimal module provides support for fast correctly-rounded decimal floating point arithmetic. It offers several advantages over the float datatype:

Decimal ”is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school.” – excerpt from the decimal arithmetic specification.

Decimal numbers can be represented exactly. In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point. End users typically would not expect $1.1 + 2.2$ to display as 3.3000000000000003 as it does with binary floating point.

The exactness carries over into arithmetic. In decimal floating point, $0.1 + 0.1 + 0.1 - 0.3$ is exactly equal to zero. In binary floating point, the result is 5.5511151231257827e-017. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.

The decimal module incorporates a notion of significant places so that $1.30 + 1.20$ is 2.50. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the ”schoolbook” approach uses all the figures in the multiplicands. For instance, $1.3 * 1.2$ gives 1.56 while $1.30 * 1.20$ gives 1.5600.

Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

The decimal module was designed to support ”without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic.” – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as Infinity, -Infinity, and NaN. The standard also differentiates -0 from +0.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, ROUND_UP, and ROUND_05UP.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: Clamped, InvalidOperation, DivisionByZero, Inexact, Rounded, Subnormal, Overflow, Underflow and FloatOperation.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See also:

IBM's General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification. IEEE standard 854-1987, Unofficial IEEE 854 Text.

3.7.2 Quick-start Tutorial

The usual start to using decimals is importing the module, viewing the current context with `getcontext()` and, if necessary, setting new values for precision, rounding, or enabled traps:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
InvalidOperation])

>>> getcontext().prec = 7      # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for `Not a number`, positive and negative Infinity, and `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
```

```
>>> Decimal(' -Infinity ')
Decimal(' -Infinity ')
```

If the `FloatOperation` signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

New in version 3.3.

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND\_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

If the internal limits of the C version are exceeded, constructing a decimal raises `InvalidOperationException`:

Changed in version 3.3.

Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
```

```
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[0]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

And some mathematical functions are also available to Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

The `quantize()` method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the `getcontext()` function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the `Context()` constructor. To make an alternate active, use the `setcontext()` function.

In accordance with the standard, the `Decimal` module provides two ready to use standard contexts, `BaseContext` and `ExtendedContext`. The former is especially useful for debugging because many of the traps are enabled:

```
Context(prec=9, rounding=ROUND\_HALF\_EVEN, Emin=-999999, Emax=999999,
capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
File "<pyshell#143>", line 1, in <toplevel>
Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the `clear_flags()` method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND\_HALF\_EVEN, Emin=-999999, Emax=999999,
capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])

```

The flags entry shows that the rational approximation to Pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` field of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
File "<pyshell#112>", line 1, in <toplevel>
Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to `Decimal` with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

3.7.3 Decimal objects

`class decimal.Decimal(value="0", context=None)`

Construct a new `Decimal` object based from `value`.

`value` can be an integer, string, tuple, float, or another `Decimal` object. If no `value` is given,

returns `Decimal('0')`. If `value` is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters are removed:

```

sign      ::= '+' | '-'
digit    ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator ::= 'e' | 'E'
digits   ::= digit [digit]...
decimal-part ::= digits '.' [digits] | ['.'] digits
exponent-part ::= indicator [sign] digits
infinity  ::= 'Infinity' | 'Inf'
nan       ::= 'NaN' [digits] | 'sNaN' [digits]
numeric-value ::= decimal-part [exponent-part] | infinity
numeric-string ::= [sign] numeric-value | [sign] nan

```

Other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanagari digits) along with the fullwidth digits 'uff10' through 'uff19'.

If `value` is a tuple, it should have three components, a sign (0 for positive or 1 for negative), a tuple of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`.

If `value` is a float, the binary floating point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts to `Decimal('1.10000000000000088817841970012523233890533447265625')`.

The context precision does not affect how many digits are stored. That is determined exclusively by the number of digits in `value`. For example, `Decimal('3.00000')` records all five zeros even if the context precision is only three.

The purpose of the `context` argument is determining what to do if `value` is a malformed string. If the context traps `InvalidOperationException`, an exception is raised; otherwise, the constructor returns a new `Decimal` with the value of `NaN`.

Once constructed, `Decimal` objects are immutable.

Changed in version 3.2: The argument to the constructor is now permitted to be a float instance.

Changed in version 3.3: float arguments raise an exception if the `FloatOperation` trap is set. By default the trap is off.

`Decimal` floating point objects share many properties with the other built-in numeric types such as `float` and `int`. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as `float` or `int`).

There are some small differences between arithmetic on `Decimal` objects and arithmetic on integers and floats. When the remainder operator `%` is applied to `Decimal` objects, the sign of the result is the sign of the dividend rather than the sign of the divisor:

```

>>> (-7) % 4
1

```

```
>>> Decimal(-7) % Decimal(4)
Decimal(' -3 ')
```

The integer division operator `//` behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity $x == (x // y) * y + x$

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal(' -1 ')
```

The `%` and `//` operators implement the remainder and divide-integer operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of fractions. Fraction in arithmetic operations: an attempt to add a Decimal to a float, for example, will raise a `TypeError`. However, it is possible to use Python's comparison operators to compare a Decimal instance `x` with another number `y`. This avoids confusing results when doing equality comparisons between numbers of different types.

Changed in version 3.2: Mixed-type comparisons between Decimal instances and other numeric types are now fully supported.

3.7.4 Methods

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

3.7.4.1 `adjusted()`

Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

3.7.4.2 `as_tuple()`

Return a named tuple representation of the number: `DecimalTuple(sign, digits, exponent)`.

3.7.4.3 `canonical()`

Return the canonical encoding of the argument. Currently, the encoding of a Decimal instance is always canonical, so this operation returns its argument unchanged.

3.7.4.4 `compare(other, context=None)`

Compare the values of two Decimal instances. `compare()` returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN ==> Decimal('NaN')
a < b           ==> Decimal(' -1 ')
a == b           ==> Decimal('0 ')
a > b           ==> Decimal('1 ')
```

3.7.4.5 `compare_signal(other, context=None)`

This operation is identical to the `compare()` method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

3.7.4.6 `compare_total(other, context=None)`

Compare two operands using their abstract representation rather than their numerical value. Similar to the `compare()` method, but the result gives a total ordering on Decimal instances. Two Decimal instances with the same numeric value but different representations compare unequal in this ordering:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

3.7.4.7 `compare_total_mag(other, context=None)`

Compare two operands using their abstract representation rather than their value as in `compare_total()`, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

3.7.4.8 `conjugate()`

Just returns `self`, this method is only to comply with the Decimal Specification.

3.7.4.9 `copy_abs()`

Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

3.7.4.10 `copy_negate()`

Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

3.7.4.11 `copy_sign(other, context=None)`

Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise `InvalidOperation` if the second operand cannot be converted exactly.

3.7.4.12 `exp(context=None)`

Return the value of the (natural) exponential function e^{**x} at the given number. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

3.7.4.13 `from_float(f)`

Classmethod that converts a float to a decimal number, exactly.

Note `Decimal.from_float(0.1)` is not the same as `Decimal('0.1')`. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is `0x1.999999999999ap-4`. That equivalent value in decimal is

`0.100000000000000055511151231257827021181583404541015625`.

Note: From Python 3.2 onwards, a `Decimal` instance can also be constructed directly from a float.

```
>>> Decimal.from_float(0.1)
Decimal('0.100000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

3.7.4.14 `fma(other, third, context=None)`

New in version 3.1.

Fused multiply-add. Return `self*other+third` with no rounding of the intermediate product `self*other`.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

3.7.4.15 `is_canonical()`

Return `True` if the argument is canonical and `False` otherwise. Currently, a `Decimal` instance is always canonical, so this operation always returns `True`.

3.7.4.16 is_finite()

Return True if the argument is a finite number, and False if the argument is an infinity or a NaN.

3.7.4.17 is_infinite()

Return True if the argument is either positive or negative infinity and False otherwise.

3.7.4.18 is_nan()

Return True if the argument is a (quiet or signaling) NaN and False otherwise.

3.7.4.19 is_normal(context=None)

Return True if the argument is a normal finite number. Return False if the argument is zero, subnormal, infinite or a NaN.

3.7.4.20 is_qnan()

Return True if the argument is a quiet NaN, and False otherwise.

3.7.4.21 is_signed()

Return True if the argument has a negative sign and False otherwise. Note that zeros and NaNs can both carry signs.

3.7.4.22 is_snan()

Return True if the argument is a signaling NaN and False otherwise.

3.7.4.23 is_subnormal(context=None)

Return True if the argument is subnormal, and False otherwise.

3.7.4.24 is_zero()

Return True if the argument is a (positive or negative) zero and False otherwise.

3.7.4.25 ln(context=None)

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the ROUND_HALF_EVEN rounding mode.

3.7.4.26 log10(context=None)

Return the base ten logarithm of the operand. The result is correctly rounded using the ROUND_HALF_EVEN rounding mode.

3.7.4.27 logb(context=None)

For a nonzero number, return the adjusted exponent of its operand as a Decimal instance. If the operand is a zero then Decimal(''-Infinity') is returned and the DivisionByZero flag is raised. If the operand is an infinity then Decimal('Infinity') is returned.

3.7.4.28 logical_and(other, context=None)

`logical_and()` is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise and of the two operands.

3.7.4.29 logical_invert(context=None)

`logical_invert()` is a logical operation. The result is the digit-wise inversion of the operand.

3.7.4.30 logical_or(other, context=None)

`logical_or()` is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise or of the two operands.

3.7.4.31 logical_xor(other, context=None)

`logical_xor()` is a logical operation which takes two logical operands (see Logical operands). The result is the digit-wise exclusive or of the two operands.

3.7.4.32 max(other, context=None)

Like `max(self, other)` except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

3.7.4.33 max_mag(other, context=None)

Similar to the `max()` method, but the comparison is done using the absolute values of the operands.

3.7.4.34 min(other, context=None)

Like `min(self, other)` except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

3.7.4.35 min_mag(other, context=None)

Similar to the `min()` method, but the comparison is done using the absolute values of the operands.

`next_minus(context=None)` Return the largest number representable in the given context (or in the current threadâŽs context if no context is given) that is smaller than the given operand.

3.7.4.36 next_plus(context=None)

Return the smallest number representable in the given context (or in the current threadâŽs context if no context is given) that is larger than the given operand.

3.7.4.37 next_toward(other, context=None)

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

3.7.4.38 `normalize(context=None)`

Normalize the number by stripping the rightmost trailing zeros and converting any result equal to `Decimal('0')` to `Decimal('0e0')`. Used for producing canonical values for attributes of an equivalence class. For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

3.7.4.39 `number_class(context=None)`

Return a string describing the class of the operand. The returned value is one of the following ten strings.

`"-Infinity"`, indicating that the operand `is` negative infinity.
`"-Normal"`, indicating that the operand `is` a negative normal number.
`"-Subnormal"`, indicating that the operand `is` negative `and` subnormal.
`"-Zero"`, indicating that the operand `is` a negative zero.
`"+Zero"`, indicating that the operand `is` a positive zero.
`"+Subnormal"`, indicating that the operand `is` positive `and` subnormal.
`"+Normal"`, indicating that the operand `is` a positive normal number.
`"+Infinity"`, indicating that the operand `is` positive infinity.
`"NaN"`, indicating that the operand `is` a quiet NaN (Not a Number).
`"sNaN"`, indicating that the operand `is` a signaling NaN.

3.7.4.40 `quantize(exp, rounding=None, context=None, watchexp=True)`

Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an `InvalidOperation` is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand. Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the rounding argument if given, else by the given context argument; if neither argument is given the rounding mode of the current threadâŽs context is used.

If `watchexp` is set (default), then an error is returned whenever the resulting exponent is greater than `Emax` or less than `Etiny`.

Deprecated since version 3.3: `watchexp` is an implementation detail from the pure Python version and is not present in the C version. It will be removed in version 3.4, where it defaults to `True`.

3.7.4.41 `radix()`

Return `Decimal(10)`, the radix (base) in which the `Decimal` class does all its arithmetic. Included for compatibility with the specification.

3.7.4.42 remainder_near(other, context=None)

Return the remainder from dividing self by other. This differs from self % other in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is self - n * other where n is the integer nearest to the exact value of self / other, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of self.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

3.7.4.43 rotate(other, context=None)

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

3.7.4.44 same_quantum(other, context=None)

Test whether self and other have the same exponent or whether both are NaN.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

3.7.4.45 scaleb(other, context=None)

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by $10^{**other}$. The second operand must be an integer.

3.7.4.46 shift(other, context=None)

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

3.7.4.47 sqrt(context=None)

Return the square root of the argument to full precision.

3.7.4.48 `to_eng_string(context=None)`

Convert to an engineering-type string.

Engineering notation has an exponent which is a multiple of 3, so there are up to 3 digits left of the decimal place. For example, converts `Decimal('123E+1')` to `Decimal('1.23E+3')`

3.7.4.49 `to_integral(rounding=None, context=None)`

Identical to the `to_integral_value()` method. The `to_integral` name has been kept for compatibility with older versions.

3.7.4.50 `to_integral_exact(rounding=None, context=None)`

Round to the nearest integer, signaling `Inexact` or `Rounded` as appropriate if rounding occurs. The rounding mode is determined by the rounding parameter if given, else by the given context. If neither parameter is given then the rounding mode of the current context is used.

3.7.4.51 `to_integral_value(rounding=None, context=None)`

Round to the nearest integer without signaling `Inexact` or `Rounded`. If given, applies rounding; otherwise, uses the rounding method in either the supplied context or the current context.

3.7.4.52 Logical operands

The `logical_and()`, `logical_invert()`, `logical_or()`, and `logical_xor()` methods expect their arguments to be logical operands. A logical operand is a `Decimal` instance whose exponent and sign are both zero, and whose digits are all either 0 or 1.

3.7.5 Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the `getcontext()` and `setcontext()` functions:

3.7.5.1 `decimal.getcontext()`

Return the current context for the active thread.

3.7.5.2 `decimal.setcontext(c)`

Set the current context for the active thread to `c`.

You can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

3.7.5.3 `decimal.localcontext(ctx=None)`

Return a context manager that will set the current context for the active thread to a copy of `ctx` on entry to the `with`-statement and restore the previous context when exiting the `with`-statement. If no context is specified, a copy of the current context is used.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42 # Perform a high precision calculation
    s = calculate\ something()
    s = +s # Round the final result back to the default precision
```

New contexts can also be created using the Context constructor described below. In addition, the module provides three pre-made contexts:

3.7.5.4 class decimal.BasicContext

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_UP. All flags are cleared. All traps are enabled (treated as exceptions) except Inexact, Rounded, and Subnormal.

Because many of the traps are enabled, this context is useful for debugging.

3.7.5.5 class decimal.ExtendedContext

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to ROUND_HALF_EVEN. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of NaN or Infinity instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

3.7.5.6 class decimal.DefaultContext

This context is used by the Context constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the Context constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are prec=28, rounding=ROUND_HALF_EVEN, and enabled traps for Overflow, InvalidOperation, and DivisionByZero.

In addition to the three supplied contexts, new contexts can be created with the Context constructor.

3.7.5.7 class decimal.Context(prec=None, rounding=None, Emin=None, Emax=None, capitals=None, clamp=None, flags=None, traps=None)

Creates a new context. If a field is not specified or is None, the default values are copied from the DefaultContext. If the flags field is not specified or is None, all flags are cleared.

prec is an integer in the range [1, MAX_PREC] that sets the precision for arithmetic operations in the context.

The rounding option is one of the constants listed in the section Rounding Modes.

The traps and flags fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The Emin and Emax fields are integers specifying the outer limits allowable for exponents. Emin must be in the range [MIN_EMIN, 0], Emax in the range [0, MAX_EMAX].

The capitals field is either 0 or 1 (the default). If set to 1, exponents are printed with a capital E; otherwise, a lowercase e is used: `Decimal('6.02e+23')`.

The clamp field is either 0 (the default) or 1. If set to 1, the exponent e of a Decimal instance representable in this context is strictly limited to the range $Emin - prec + 1 \leq e \leq Emax - prec + 1$. If clamp is 0 then a weaker condition holds: the adjusted exponent of the Decimal instance is at most Emax. When clamp is 1, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros.

For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A clamp value of 1 allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

3.7.6 Context Methods

The Context class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the Decimal methods described above (with the exception of the `adjusted()` and `as_tuple()` methods) there is a corresponding Context method. For example, for a Context instance C and Decimal instance x, `C.exp(x)` is equivalent to `x.exp(context=C)`. Each Context method accepts a Python integer (an instance of `int`) anywhere that a Decimal instance is accepted.

3.7.6.1 clear_flags()

Resets all of the flags to 0.

3.7.6.2 clear_traps()

Resets all of the traps to 0.

New in version 3.3.

3.7.6.3 copy()

Return a duplicate of the context.

3.7.6.4 copy_decimal(num)

Return a copy of the Decimal instance num.

3.7.6.5 create_decimal(num)

Creates a new Decimal instance from num but using self as context. Unlike the Decimal constructor, the context precision, rounding method, flags, and traps are applied to the conversion. This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace is permitted.

3.7.6.6 create_decimal_from_float(f)

Creates a new Decimal instance from a float f but rounding using self as the context. Unlike the Decimal.from_float() class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```
>>> context = Context(prec=5, rounding=ROUND\_DOWN)
>>> context.create\_decimal\_from\_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create\_decimal\_from\_float(math.pi)
Traceback (most recent call last):
...
decimal.Inexact: None
```

New in version 3.1.

3.7.6.7 Etiny()

Returns a value equal to Emin - prec + 1 which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to Etiny.

3.7.6.8 Etop()

Returns a value equal to Emax - prec + 1.

The usual approach to working with decimals is to create Decimal instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the Decimal class and are only briefly recounted here.

3.7.6.9 abs(x)

Returns the absolute value of x.

3.7.6.10 add(x, y)

Return the sum of x and y.

3.7.6.11 canonical(x)

Returns the same Decimal object x.

3.7.6.12 compare(x, y)

Compares x and y numerically.

3.7.6.13 compare_signal(x, y)

Compares the values of the two operands numerically.

3.7.6.14 compare_total(x, y)

Compares two operands using their abstract representation.

3.7.6.15 compare_total_mag(x, y)

Compares two operands using their abstract representation, ignoring sign.

3.7.6.16 copy_abs(x)

Returns a copy of x with the sign set to 0.

3.7.6.17 copy_negate(x)

Returns a copy of x with the sign inverted.

3.7.6.18 copy_sign(x, y)

Copies the sign from y to x.

3.7.6.19 divide(x, y)

Return x divided by y.

3.7.6.20 divide_int(x, y)

Return x divided by y, truncated to an integer.

3.7.6.21 divmod(x, y)

Divides two numbers and returns the integer part of the result.

3.7.6.22 exp(x)

Returns $e^{**} x$.

3.7.6.23 fma(x, y, z)

Returns x multiplied by y , plus z .

3.7.6.24 is_canonical(x)

Returns True if x is canonical; otherwise returns False.

3.7.6.25 is_finite(x)

Returns True if x is finite; otherwise returns False.

3.7.6.26 is_infinite(x)

Returns True if x is infinite; otherwise returns False.

3.7.6.27 is_nan(x)

Returns True if x is a qNaN or sNaN; otherwise returns False.

3.7.6.28 is_normal(x)

Returns True if x is a normal number; otherwise returns False.

3.7.6.29 is_qnan(x)

Returns True if x is a quiet NaN; otherwise returns False.

3.7.6.30 is_signed(x)

Returns True if x is negative; otherwise returns False.

3.7.6.31 is_snan(x)

Returns True if x is a signaling NaN; otherwise returns False.

3.7.6.32 is_subnormal(x)

Returns True if x is subnormal; otherwise returns False.

3.7.6.33 is_zero(x)

Returns True if x is a zero; otherwise returns False.

3.7.6.34 ln(x)

Returns the natural (base e) logarithm of x .

3.7.6.35 log10(x)

Returns the base 10 logarithm of x.

3.7.6.36 logb(x)

Returns the exponent of the magnitude of the operand's MSD.

3.7.6.37 logical_and(x, y)

Applies the logical operation and between each operand's digits.

3.7.6.38 logical_invert(x)

Invert all the digits in x.

3.7.6.39 logical_or(x, y)

Applies the logical operation or between each operand's digits.

3.7.6.40 logical_xor(x, y)

Applies the logical operation xor between each operand's digits.

3.7.6.41 max(x, y)

Compares two values numerically and returns the maximum.

3.7.6.42 max_mag(x, y)

Compares the values numerically with their sign ignored.

3.7.6.43 min(x, y)

Compares two values numerically and returns the minimum.

3.7.6.44 min_mag(x, y)

Compares the values numerically with their sign ignored.

3.7.6.45 minus(x)

Minus corresponds to the unary prefix minus operator in Python.

3.7.6.46 multiply(x, y)

Return the product of x and y.

3.7.6.47 next_minus(x)

Returns the largest representable number smaller than x.

3.7.6.48 next_plus(x)

Returns the smallest representable number larger than x.

3.7.6.49 next_toward(x, y)

Returns the number closest to x, in direction towards y.

3.7.6.50 normalize(x)

Reduces x to its simplest form.

3.7.6.51 number_class(x)

Returns an indication of the class of x.

3.7.6.52 plus(x)

Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is not an identity operation.

3.7.6.53 power(x, y, modulo=None)

Return x to the power of y, reduced modulo modulo if given.

With two arguments, compute $x^{**}y$. If x is negative then y must be integral. The result will be inexact unless y is integral and the result is finite and can be expressed exactly in precision digits. The rounding mode of the context is used. Results are always correctly-rounded in the Python version.

Changed in version 3.3: The C module computes power() in terms of the correctly-rounded exp() and ln() functions. The result is well-defined but only *almost* always correctly-rounded. With three arguments, compute $(x^{**}y) \% \text{modulo}$. For the three argument form, the following restrictions on the arguments hold:

all three arguments must be integral
y must be nonnegative
at least one of x or y must be nonzero
modulo must be nonzero and have at most precision digits

The value resulting from Context.power(x, y, modulo) is equal to the value that would be obtained by computing $(x^{**}y) \% \text{modulo}$ with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of x, y and modulo. The result is always exact.

3.7.6.54 quantize(x, y)

Returns a value equal to x (rounded), having the exponent of y.

3.7.6.55 radix()

radix() Just returns 10, as this is Decimal.

3.7.6.56 remainder(x, y)

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

3.7.6.57 remainder_near(x, y)

Returns $x - y * n$, where n is the integer nearest the exact value of x / y (if the result is 0 then its sign will be the sign of x).

3.7.6.58 rotate(x, y)

Returns a rotated copy of x , y times.

3.7.6.59 same_quantum(x, y)

Returns True if the two operands have the same exponent.

3.7.6.60 scaleb(x, y)

Returns the first operand after adding the second value its exp.

3.7.6.61 shift(x, y)

Returns a shifted copy of x , y times.

3.7.6.62 sqrt(x)

Square root of a non-negative number to context precision.

3.7.6.63 subtract(x, y)

Return the difference between x and y .

3.7.6.64 to_eng_string(x)

to_eng_string(x) Converts a number to a string, using scientific notation.

3.7.6.65 to_integral_exact(x)

to_integral_exact(x) Rounds to an integer.

3.7.6.66 to_sci_string(x)

Converts a number to a string using scientific notation.

3.7.7 Constants

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

decimal.HAVE_THREADS The default value is True. If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the value is False.

	32-bit	64-bit
decimal.MAX_PREC	425000000	99999999999999999999
decimal.MAX_EMAX	425000000	99999999999999999999
decimal.MIN_EMIN	-425000000	-99999999999999999999
decimal.MIN_ETINY	-849999999	-19999999999999999997

3.7.8 Rounding modes

```

decimal.ROUND\_CEILING
Round towards Infinity.

decimal.ROUND\_DOWN
Round towards zero.

decimal.ROUND\_FLOOR
Round towards -Infinity.

decimal.ROUND\_HALF\_DOWN
Round to nearest with ties going towards zero.

decimal.ROUND\_HALF\_EVEN
Round to nearest with ties going to nearest even integer.

decimal.ROUND\_HALF\_UP
Round to nearest with ties going away from zero.

decimal.ROUND\_UP
Round away from zero.

decimal.ROUND\_05UP
Round away from zero if last digit after rounding towards zero would have been 0 or
5; otherwise round towards zero.

```

3.7.9 Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the DivisionByZero trap is set, then a DivisionByZero exception is raised upon encountering the condition.

3.7.9.1 class decimal.Clamped

Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's Emin and Emax limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

3.7.9.2 class decimal.DecimalException

Base class for other signals and a subclass of ArithmeticError.

3.7.9.3 class decimal.DivisionByZero

Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns Infinity or -Infinity with the sign determined by the inputs to the calculation.

3.7.9.4 class decimal.Inexact

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

3.7.9.5 class decimal.InvalidOperation

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns NaN. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

3.7.9.6 class decimal.Overflow

Numerical overflow.

Indicates the exponent is larger than Emax after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to Infinity. In either case, Inexact and Rounded are also signaled.

3.7.9.7 class decimal.Rounded

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding 5.00 to 5.0). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

3.7.9.8 class decimal.Subnormal

Exponent was lower than Emin prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

3.7.9.9 class decimal.Underflow

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. Inexact and Subnormal are also signaled.

3.7.9.10 class decimal.FloatOperation

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the Decimal constructor, create_decimal() and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting FloatOperation in the context flags. Explicit conversions with from_float() or create_decimal.from_float() do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise FloatOperation.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.Exception)
DecimalException
Clamped
DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
Inexact
Overflow(Inexact, Rounded)
Underflow(Inexact, Rounded, Subnormal)
InvalidOperation
Rounded
Subnormal
FloatOperation(DecimalException, exceptions.TypeError)
```

3.7.10 Floating Point Notes

3.7.10.1 Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent 0.1 exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

Examples from Seminumerical Algorithms, Section 4.2.2.

```
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal('111111113'), Decimal('111111111'), Decimal('7.51111111')
>>> (u + v) + w
```

```

Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')

```

The decimal module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```

>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')

```

3.7.10.2 Special values

The number system for the decimal module provides special values including NaN, sNaN, -Infinity, Infinity, and two zeros, +0 and -0.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the `DivisionByZero` signal is not trapped. Likewise, when the `Overflow` signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return NaN, or if the `InvalidOperation` signal is trapped, raise an exception. For example, `0/0` returns NaN which means "not a number". This variety of NaN is quiet and, once created, will flow through other computations always resulting in another NaN. This behavior can be useful for a series of computations that occasionally have missing inputs – it allows the calculation to proceed while flagging specific results as invalid.

A variant is sNaN which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a NaN is involved. A test for equality where one of the operands is a quiet or signaling NaN always returns `False` (even when doing `Decimal('NaN') == Decimal('NaN')`), while a test for inequality always returns `True`. An attempt to compare two Decimals using any of the `<`, `<=`, `>` or `>=` operators will

raise the `InvalidOperation` signal if either operand is a `NaN`, and return `False` if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the `compare()` and `compare-signal()` methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

3.7.11 Working with threads

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called `DefaultContext`. To control the defaults so that each thread will use the same values throughout the application, directly modify the `DefaultContext` object. This should be done before any threads are started so that there won't be a race condition between threads calling `getcontext()`. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND\_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
...
```

3.7.12 Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class:

```

def moneyfmt(value, places=2, curr='', sep=',', dp='.',
pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

places: required number of places after the decimal point
curr: optional currency symbol before the sign (may be blank)
sep: optional grouping separator (comma, period, space, or blank)
dp: decimal point indicator (comma or period)
only specify as blank when places is zero
pos: optional sign for positive numbers: '+', space or blank
neg: optional sign for negative numbers: '(', space or blank
trailneg:optional trailing minus indicator: '(', ')', space or blank

>>> d = Decimal('-1234567.8901')
>>> moneyfmt(d, curr='$')
'$-1,234,567.89'
>>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
'1.234.568-'
>>> moneyfmt(d, curr='$', neg='(', trailneg=')')
'($1,234,567.89)'
>>> moneyfmt(Decimal(123456789), sep=' ')
'123 456 789.00'
>>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
'<0.02>'

"""
q = Decimal(10) ** -places    # 2 places --> '0.01'
sign, digits, exp = value.quantize(q).as_tuple()
result = []
digits = list(map(str, digits))
build, next = result.append, digits.pop
if sign:
    build(trailneg)
for i in range(places):
    build(next() if digits else '0')
if places:
    build(dp)
if not digits:
    build('0')
i = 0
while digits:
    build(next())
    i += 1
    if i == 3 and digits:
        i = 0
        build(sep)
        build(curr)
        build(neg if sign else pos)
return ''.join(reversed(result))

def pi():

```

```

"""Compute Pi to the current precision.

>>> print(pi())
3.141592653589793238462643383

"""

getcontext().prec += 2 # extra digits for intermediate steps
three = Decimal(3)    # substitute "three=3.0" for regular floats
lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
while s != lasts:
    lasts = s
    n, na = n+na, na+8
    d, da = d+da, da+32
    t = (t * n) / d
    s += t
getcontext().prec -= 2
return +s             # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x. Result type matches input type.

>>> print(exp(Decimal(1)))
2.718281828459045235360287471
>>> print(exp(Decimal(2)))
7.389056098930650227230427461
>>> print(exp(2.0))
7.38905609893
>>> print(exp(2+0j))
(7.38905609893+0j)

"""

getcontext().prec += 2
i, lasts, s, fact, num = 0, 0, 1, 1, 1
while s != lasts:
    lasts = s
    i += 1
    fact *= i
    num *= x
    s += num / fact
getcontext().prec -= 2
return +s

def cos(x):
    """Return the cosine of x as measured in radians.

```

The Taylor series approximation works best for a small value of x.
 For larger values, first compute $x = x \% (2 * \pi)$.

```

>>> print(cos(Decimal('0.5')))
0.8775825618903727161162815826
>>> print(cos(0.5))
0.87758256189

```

```

>>> print(cos(0.5+0j))
(0.87758256189+0j)

"""

getcontext().prec += 2
i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
while s != lasts:
    lasts = s
    i += 2
    fact *= i * (i-1)
    num *= x * x
    sign *= -1
    s += num / fact * sign
getcontext().prec -= 2
return +s

```

```

def sin(x):
    """Return the sine of x as measured in radians.

```

The Taylor series approximation works best for a small value of x.
For larger values, first compute $x = x \% (2 * \pi)$.

```

>>> print(sin(Decimal('0.5')))
0.4794255386042030002732879352
>>> print(sin(0.5))
0.479425538604
>>> print(sin(0.5+0j))
(0.479425538604+0j)

```

```

"""

getcontext().prec += 2
i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
while s != lasts:
    lasts = s
    i += 2
    fact *= i * (i-1)
    num *= x * x
    sign *= -1
    s += num / fact * sign
getcontext().prec -= 2
return +s

```

3.7.13 Decimal FAQ

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```

>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')

```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The quantize() method rounds to a fixed number of decimal places. If the Inexact trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2    # same as Decimal('0.01')

>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')

>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')

>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a quantize() step:

```
>>> a = Decimal('102.72')      # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                      # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                      # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES) # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES) # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the quantize() step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                                # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers 200, 200.000, 2E2, and 02E+4 all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The normalize() method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing 5.0E+3 as 5000 keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```
>>> def remove\_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to\_integral() else d.normalize()

>>> remove\_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. Is there a way to convert a regular float to a Decimal?

A. Yes, any binary floating point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Within a complex calculation, how can I make sure that I have not gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that "what you type is what you get". A disadvantage is that the results can look odd if you forget that the inputs have not been rounded:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
```

```
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')    # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the Context.create_decimal() method:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

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='0.333333333321', imag='0.666666666642')
```

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('10.8999999999999999999999999999997')
>>> mpf(109) / mpf(10) # also good
mpf('10.8999999999999999999999999999997')
>>> 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)
mpi('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:

```
0.000000000000000088817841970012523233890533447265625
```

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 π .

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 of formatted numbers

4.3.1 Conversions between Roman and Arabic Numbers

WorksheetFunction.**ROMAN**(*Number* As *mpNum*, *Form* As *Integer*) As String

NOT YET IMPLEMENTED

The function WorksheetFunction.ROMAN returns Converts an arabic numeral to roman, as text.

Parameters:

Number: The Arabic numeral you want converted.

Form: A number from 0 to 4 specifying the type of roman numeral you want. The roman numeral style ranges from Classic to Simplified, becoming more concise as the value of form increases..

WorksheetFunction.**ARABIC**(*Number* As *mpNum*) As String

NOT YET IMPLEMENTED

The function WorksheetFunction.ARABIC returns Converts an roman numeral to arabic, as text.

Parameter:

Number: The roman numeral you want converted.

4.3.2 Conversions from Binary

WorksheetFunction.**BIN2DEC**(*Number* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.BIN2DEC returns Converts a binary number to decimal.

Parameter:

Number: The binary number you want to convert. Number cannot contain more than 10 characters (10 bits). The most significant bit of number is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

WorksheetFunction.**BIN2HEX**(*Number* As *mpNum*, *Places* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.BIN2HEX returns Converts a binary number to decimal.

Parameters:

Number: The binary number you want to convert. Number cannot contain more than 10 characters (10 bits). The most significant bit of number is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, BIN2HEX uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

WorksheetFunction.**BIN2OCT**(**Number** As mpNum, **Places** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.BIN2OCT returns Converts a binary number to octal.

Parameters:

Number: The binary number you want to convert. Number cannot contain more than 10 characters (10 bits). The most significant bit of number is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, BIN2OCT uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

4.3.3 Conversions from Decimal

WorksheetFunction.**DEC2BIN**(**Number** As mpNum, **Places** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.DEC2BIN returns Converts a decimal number to binary.

Parameters:

Number: The decimal integer you want to convert. If number is negative, valid place values are ignored and DEC2BIN returns a 10-character (10-bit) binary number in which the most significant bit is the sign bit. The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, DEC2BIN uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

WorksheetFunction.**DEC2HEX**(**Number** As mpNum, **Places** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.DEC2HEX returns Converts a decimal number to hexadecimal.

Parameters:

Number: The decimal integer you want to convert. If number is negative, places is ignored and DEC2HEX returns a 10-character (40-bit) hexadecimal number in which the most significant bit is the sign bit. The remaining 39 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, DEC2HEX uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

WorksheetFunction.**DEC2OCT**(**Number** As mpNum, **Places** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.DEC2OCT returns Converts a decimal number to octal.

Parameters:

Number: The decimal integer you want to convert. If number is negative, places is ignored and DEC2OCT returns a 10-character (30-bit) octal number in which the most significant bit is the sign bit. The remaining 29 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, DEC2OCT uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

4.3.4 Conversions from Hexadecimal

WorksheetFunction.**HEX2BIN**(**Number** As mpNum, **Places** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.HEX2BIN returns Converts a hexadecimal number to binary.

Parameters:

Number: The hexadecimal number you want to convert. Number cannot contain more than 10 characters. The most significant bit of number is the sign bit (40th bit from the right). The remaining 9 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, HEX2BIN uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

WorksheetFunction.**HEX2DEC**(**Number** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.HEX2DEC returns Converts a hexadecimal number to decimal.

Parameter:

Number: The hexadecimal number you want to convert. Number cannot contain more than 10 characters (40 bits). The most significant bit of number is the sign bit. The remaining 39 bits are magnitude bits. Negative numbers are represented using two's-complement notation

WorksheetFunction.**HEX2OCT**(**Number** As mpNum, **Places** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.HEX2OCT returns Converts a hexadecimal number to octal.

Parameters:

Number: The hexadecimal number you want to convert. Number cannot contain more than 10 characters. The most significant bit of number is the sign bit. The remaining 39 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, HEX2OCT uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

4.3.5 Conversions from Octal

WorksheetFunction.**OCT2BIN**(*Number* As mpNum, *Places* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.OCT2BIN returns Converts a octal number to binary.

Parameters:

Number: The octal number you want to convert. Number may not contain more than 10 characters. The most significant bit of number is the sign bit. The remaining 29 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, OCT2BIN uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

WorksheetFunction.**OCT2DEC**(*Number* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.OCT2DEC returns Converts an octal number to decimal.

Parameter:

Number: The octal number you want to convert. Number may not contain more than 10 octal characters (30 bits). The most significant bit of number is the sign bit. The remaining 29 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

WorksheetFunction.**OCT2HEX**(*Number* As mpNum, *Places* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.OCT2HEX returns Converts an octal number to hexadecimal.

Parameters:

Number: The octal number you want to convert. Number may not contain more than 10 octal characters (30 bits). The most significant bit of number is the sign bit. The remaining 29 bits are magnitude bits. Negative numbers are represented using two's-complement notation.

Places: The number of characters to use. If places is omitted, OCT2HEX uses the minimum number of characters necessary. Places is useful for padding the return value with leading 0s (zeros).

4.3.6 Conversion to and from a Given Base

WorksheetFunction.**BASE**(*Number* As mpNum, *Radix* As mpNum, *MinLength* As mpNum) As mpNum

The function WorksheetFunction.BASE returns converts a number into a text representation with the given radix (base).

Parameters:

Number: The number that you want to convert. Must be an integer greater than or equal to 0 and less than 2^{53} .

Radix: The base radix that you want to convert the number into. Must be an integer greater than or equal to 2 and less than or equal to 36.

MinLength: The minimum length of the returned string. Must be an integer greater than or equal to 0.

WorksheetFunction.**DECIMAL**(*Text* As String, *Radix* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.DECIMAL returns Converts a text representation of a number in a given base into a decimal number.

Parameters:

Text: The string length of Text must be less than or equal to 255 characters.

Radix: Radix must be an integer greater than or equal to 2 (binary, or base 2) and less than or equal to 36 (base 36).

4.4 Conversion and printing

4.4.1 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')
```

4.4.2 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]
```

4.5 Rounding

4.5.1 Nearest integer: Round(x)

WorksheetFunction.ROUND(**Number** As mpNum, **Digits** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ROUND returns a number rounded to a specified number of digits

Parameters:

Number: A real number you want to round.

Digits: The number of digits to which you want to round. Negative rounds to the left of the decimal point; zero to the nearest integer.

ROUND rounds to the nearest representable integer, rounding halfway cases away from zero (as in the roundTiesToAway mode of IEEE 754-2008).

The returned indicator value is zero when the result is exact, positive when it is greater than the original value of op, and negative when it is smaller. More precisely, the returned value is 0 when op is an integer representable in rop, 1 or -1 when op is an integer that is not representable in rop, 2 or -2 when op is not an integer.

Note that mpfr_round is different from mpfr_rint called with the rounding to nearest mode (where halfway cases are rounded to an even integer or significand). Note also that no double rounding is performed; for instance, 10.5 (1010.1 in binary) is rounded by mpfr_rint with rounding to nearest to 12 (1100 in binary) in 2-bit precision, because the two enclosing numbers representable on two bits are 8 and 12, and the closest is 12. (If one first rounded to an integer, one would round 10.5 to 10 with even rounding, and then 10 would be rounded to 8 again with even rounding.)

4.5.2 Next higher or equal integer: Ceil(x)

WorksheetFunction.CEILING(**Number** As mpNum, **Significance** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.CEILING returns a number rounded up to the nearest multiple of significance

Parameters:

Number: A real number you want to round.

Significance: The multiple to which you want to round.

WorksheetFunction.CEILING.PRECISE(**Number** As mpNum, **Significance** As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.CEILING.PRECISE returns a number rounded up to the nearest multiple of significance

Parameters:

Number: A real number you want to round.

Significance: The multiple to which you want to round.

WorksheetFunction.**CEILING.MATH**(*Number* As *mpNum*, *Significance* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.CEILING.MATH returns a number rounded up to the nearest multiple of significance

Parameters:

Number: A real number you want to round.

Significance: The multiple to which you want to round.

CEILING rounds to the next higher or equal representable integer.

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().

4.5.3 Next lower or equal integer: Floor(*x*)

WorksheetFunction.**FLOOR**(*Number* As *mpNum*, *Significance* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.FLOOR returns a number rounded down to the nearest multiple of significance

Parameters:

Number: A real number you want to round.

Significance: The multiple to which you want to round. Number and Significance must either both be positive or both negative

WorksheetFunction.**FLOOR.PRECISE**(*Number* As *mpNum*, *Significance* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function `WorksheetFunction.FLOOR.PRECISE` returns a number rounded down to the nearest integer or to the nearest multiple of significance

Parameters:

Number: A real number you want to round.

Significance: The multiple to which you want to round.

`WorksheetFunction.FLOOR.MATH(Number As mpNum, Significance As mpNum) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.FLOOR.MATH` returns a number rounded down to the nearest multiple of significance

Parameters:

Number: A real number you want to round.

Significance: The multiple to which you want to round.

`FLOOR` rounds to the next lower or equal representable integer.

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 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)))
1000000000000000000019884624838656
>>> print(int(floor(10**30+1, prec=0)))
100000000000000000000000000000000000000000000000000000000000000001
```

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')
```

4.5.4 Next integer, rounded toward zero: Trunc(x)

WorksheetFunction.**TRUNC**(*Number* As mpNum, *Digits* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.TRUNC returns a number truncated to an integer by removing the decimal, or fractional, part of a number

Parameters:

Number: A real number you want to round.

Digits: A number specifying the precision of the truncation, 0 if omitted.

TRUNC rounds to the next representable integer toward zero.

4.5.5 EVEN(x)

WorksheetFunction.**EVEN**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.EVEN returns the rounded value of x . Rounds a positive number up and a negative number down to the nearest even integer.

Parameter:

x : A real number.

Returns number rounded up to the nearest even integer. You can use this function for processing items that come in twos. For example, a packing crate accepts rows of one or two items. The crate is full when the number of items, rounded up to the nearest two, matches the crate's capacity.

4.5.6 ODD(x)

WorksheetFunction.**ODD**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ODD returns the rounded value of x . Rounds a positive number up and a negative number down to the nearest odd integer.

Parameter:

x : A real number.

Returns number rounded to the nearest odd integer.

4.5.7 Nearest integer

WorksheetFunction.**INT**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.INT returns a number down to the nearest integer.

Parameter:

x: A real number.

rounds a number down to the nearest integer.

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 `floor()`.

4.5.8 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 `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.

4.5.9 ROUNDDOWN(*Number*, *Digits*)

WorksheetFunction.**ROUNDDOWN**(*Number* As mpNum, *Digits* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ROUNDDOWN returns a number rounded down, toward zero.

Parameters:

Number: A real number you want to round.

Digits: A number specifying the precision of the truncation, 0 if omitted.

Rounds a number down, toward zero.

4.5.10 ROUNDUP(*Number*, *Digits*)

WorksheetFunction.**ROUNDUP**(*Number* As mpNum, *Digits* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ROUNDUP returns a number rounded down, away from zero.

Parameters:

Number: A real number you want to round.

Digits: A number specifying the precision of the truncation, 0 if omitted.

Rounds a number up, away from 0 (zero).

4.5.11 MROUND(*Number*, *Multiple*)

WorksheetFunction.**MROUND**(*Number* As mpNum, *Multiple* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.MROUND returns a number rounded to the desired multiple.

Parameters:

Number: A real number you want to round.

Multiple: The multiple to which you want to round.

MROUND rounds up, away from zero, if the remainder of dividing number by multiple is greater than or equal to half the value of multiple.

4.5.12 QUOTIENT(*x, y*)

WorksheetFunction.QUOTIENT(*x As mpNum, y As mpNum*) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.QUOTIENT returns the integer portion of a division.

Parameters:

x: A real number

y: A real number

Returns the integer portion of a division. Use this function when you want to discard the remainder of a division.

4.6 Components of Real and Complex Numbers

4.6.1 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')
```

4.6.2 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 exp (formally, the value pointed to by `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 `exp` is set to 0. If x is NaN or an infinity, then y is set to the same value and `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)
```

4.6.3 Building a Complex Number from Real Components

WorksheetFunction.**COMPLEX**(*x* As *mpReal*, *y* As *mpReal*) As String

NOT YET IMPLEMENTED

The function WorksheetFunction.COMPLEX returns a complex number z build from the real components x and y , as string.

Parameters:

x: A real number.

y: A real number.

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.

4.6.4 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, ϕ) 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, ϕ) .

Parameters:

x: A real number.

y: A real number.

Returns the complex number represented by polar coordinates (r, ϕ) :

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> chop(rect(2, pi))
```

```
-2.0
>>> rect(sqrt(2), -pi/4)
(1.0 - 1.0j)
```

4.6.5 Real Component

WorksheetFunction.**IMREAL**(*z* As String) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.IMREAL returns the real component x of $z = x + iy$.

Parameter:

z: A String representing a complex number.

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.real*, **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')
```

4.6.6 Imaginary Component

WorksheetFunction.**IMAGINARY**(*z* As String) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.IMAGINARY returns the imaginary component y of $z = x + iy$.

Parameter:

z: A String representing a complex number.

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

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

Parameter:

z: A complex number.

Unlike *x.imag*, **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')
```

4.6.7 Absolute Value

WorksheetFunction.**ABS**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.ABS returns the absolute value of x , $|x| = \sqrt{x^2}$.

Parameter:

x: A real number.

WorksheetFunction.**IMABS**(*z* As *String*) As *mpReal*

NOT YET IMPLEMENTED

The function WorksheetFunction.IMABS returns the absolute value of $z = x + iy$

Parameter:

z: A String representing a complex number.

The absolute value of $z = x + iy$ is calculated as

$$|z| = \sqrt{x^2 + y^2}. \quad (4.6.1)$$

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

The function **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 **fabs** returns the absolute value of $z = x + iy$

Parameter:

z: A real or complex number.

Returns the absolute value of x , $|x|$. Unlike **abs()**, **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')
```

4.6.8 Argument

WorksheetFunction.**IMARGUMENT**(*z* As String) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.IMARGUMENT returns the argument of $z = x + iy$

Parameter:

z: A String representing a complex number.

The argument θ of $z = x + iy$, is defined such that

$$z = x + iy = |x + iy|e^\theta = |x + iy|(\cos(\theta) + i \sin(\theta)). \quad (4.6.2)$$

cplxArg(*z*) is calculated as

$$\text{cplxArg}(z) = \arctan\left(\frac{y}{x}\right) = \theta, \text{ where } \theta \in (-\pi; \pi]. \quad (4.6.3)$$

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.

4.6.9 Sign

WorksheetFunction.**SIGN**(*x* As `mpNum`) As `mpNum`

NOT YET IMPLEMENTED

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

Parameter:

x: A real number.

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)
```

4.6.10 Conjugate

WorksheetFunction.**IMCONJUGATE**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

The function `WorksheetFunction.IMCONJUGATE` returns the conjugate of *z*, $\bar{z} = x - iy$

Parameter:

z: A String representing a complex number.

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')
```

4.7 Arithmetic operations

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

4.7.1 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:

4.7.2 Sums and Series

WorksheetFunction.IMSUM(*z* As String $[\!]$) As String

NOT YET IMPLEMENTED

The function `WorksheetFunction.IMSUM` returns the sum of up to 255 complex numbers.

Parameter:

`z`: An array of Strings representing an array of complex numbers.

The function **IMSUM**(z_1, z_2) returns the sum of z_1 and z_2 :

$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2). \quad (4.7.1)$$

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 $\text{len}(\text{terms}) \geq 2$, 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.

WorksheetFunction.**SUMX2MY2**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMX2MY2 returns the sum of the difference of squares of corresponding values in two arrays.

Parameters:

X: A matrix of real numbers.

Y: A matrix of real numbers.

Returns the sum of the difference of squares of corresponding values in two arrays, which need have the same number of rows R and same number of columns C . The equation for the sum of the difference of squares is:

$$\text{SUMX2MY2} = \sum_{i=1}^R \sum_{j=1}^C (X_{i,j}^2 - Y_{i,j}^2) \quad (4.7.2)$$

WorksheetFunction.**SUMX2PY2**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMX2PY2 returns the sum of the sum of squares of corresponding values in two arrays.

Parameters:

X: A matrix of real numbers.

Y: A matrix of real numbers.

Returns the sum of the sum of squares of corresponding values in two arrays, which need have the same number of rows R and same number of columns C . The equation for the sum of the sum of squares is:

$$\text{SUMX2PY2} = \sum_{i=1}^R \sum_{j=1}^C (X_{i,j}^2 + Y_{i,j}^2) \quad (4.7.3)$$

WorksheetFunction.**SUMXMY2**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMXMY2 returns the sum of squares of differences of corresponding values in two arrays.

Parameters:

X: A matrix of real numbers.

Y : A matrix of real numbers.

Returns the sum of squares of differences of corresponding values in two arrays, which need have the same number of rows R and same number of columns C . The equation for the sum of squared differences is:

$$\text{SUMXMY2} = \sum_{i=1}^R \sum_{j=1}^C (X_{i,j} - Y_{i,j})^2 \quad (4.7.4)$$

WorksheetFunction.**SUMSQ**(X As mpNumList) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMSQ returns the sum of the sum of the squares of up to 255 given arrays.

Parameter:

X : A list of up to 255 given arrays.

Returns the sum of the sum of the squares of up to 255 given arrays. The arrays A, B, \dots do not need have the same number of rows or same number of columns. The equation for SUMSQ is:

$$\text{SUMSQ} = \sum A_{i,j}^2 + \sum B_{i,j}^2 + \dots \quad (4.7.5)$$

where the summation is over all entries of A, B, \dots

WorksheetFunction.**SUMPRODUCT**(X As mpNumList) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMPRODUCT returns the product of corresponding components in up to 255 given arrays.

Parameter:

X : A list of up to 255 given arrays.

Multiplies corresponding components in up to 255 given arrays, and returns the sum of those products. The arrays A, B, \dots all need have the same number of rows R and same number of columns C . The equation for the sum of the products is:

$$\text{SUMPRODUCT} = \sum_{i=1}^R \sum_{j=1}^C (A_{i,j} \times B_{i,j} \times \dots) \quad (4.7.6)$$

WorksheetFunction.**SERIESSUM**(x As mpNum, n As Integer, m As Integer, a As mpNum $[]$) As mpNum

The function WorksheetFunction.SERIESSUM returns the sum of a (finite) power series.

Parameters:

x : The input value to the power series.

n : The initial power to which you want to raise x .

m : The step by which to increase n for each term in the series..

a : A set of j coefficients by which each successive power of x is multiplied.

The sum of a power series is calculated based on this formula:

$$\text{SERIESSUM} = a_1 x^n + a_2 x^{n+m} + a_3 x^{n+2m} + \dots + a_j x^{n+(j-1)m}. \quad (4.7.7)$$

The number of values in coefficients determines the number of terms in the power series. For example, if there are three values in coefficients, then there will be three terms in the power series.

4.7.3 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: $c = a.Minus(b)$

WorksheetFunction.**IMSUB(z1 As String, z2 As String) As String**

NOT YET IMPLEMENTED

The function WorksheetFunction.IMSUB returns the difference of $z1$ and $z2$

Parameters:

$z1$: A Strings representing a complex number.
 $z2$: A Strings representing a complex number.

The function IMSUB($z1, z2$) returns the difference of $z1$ and $z2$:

$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2). \quad (4.7.8)$$

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)
```

```
1.9999999999999999999999999  
>>> nprint(fsub(2, 1e-20, dps=15), 25)  
2.0  
>>> nprint(fsub(2, 1e-20, dps=25), 25)  
1.9999999999999999999999999  
>>> nprint(fsub(2, 1e-20, exact=True), 25)  
1.9999999999999999999999999
```

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:

4.7.4 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:

4.7.5 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)
10000000001000010000000001
>>> 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)))
10000000001000010000000001
>>> print(int(fmul(x, y, exact=True)))
10000000001000010000000001
```

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-10000000000000000000000000')
>>> fmul(x, y)
mpc(real='2.0', imag='4.0')
>>> fmul(x, y, rounding='u')
mpc(real='2.0', imag='4.000000000000009')
>>> fmul(x, y, exact=True)
Traceback (most recent call last):
...
OverflowError: the exact result does not fit in memory
```

4.7.6 Products

WorksheetFunction.**PRODUCT**(*X* As *mpNumList*) As *mpNum*

NOT YET IMPLEMENTED

The function `WorksheetFunction.PRODUCT` returns the product of all the numbers of up to 255 given arrays.

Parameter:

`X`: A list of up to 255 given arrays.

The `PRODUCT` function multiplies all the numbers of up to 255 given arrays and returns the product. The arrays A, B, \dots do not need have the same number of rows or same number of columns. The equation for `PRODUCT` is:

$$\text{PRODUCT} = \prod A_{i,j} \times \prod B_{i,j} \times \dots \quad (4.7.9)$$

where the multiplication is over all entries of A, B, \dots

`WorksheetFunction.IMPRODUCT(z As String[])` As String

NOT YET IMPLEMENTED

The function `WorksheetFunction.IMPRODUCT` returns the product of up to 255 complex numbers.

Parameter:

`z`: An array of Strings representing an array of complex numbers.

The function `cplxMul(z1, z2)` returns the product of $z1$ and $z2$:

$$z_1 \cdot z_2 = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1). \quad (4.7.10)$$

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 ,

$$\sum_{k=0} A_k B_k \quad (4.7.11)$$

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:

$$\sum_{k=0} A_k \overline{B_k} \quad (4.7.12)$$

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')
```

4.7.7 Multiplication by multiples of 2 (LSH)

WorksheetFunction.**BITLSHIFT**(*n* As Integer, *k* As Integer) As Integer

NOT YET IMPLEMENTED

The function `WorksheetFunction.BITLSHIFT` returns the product of n and 2^k .

Parameters:

- n*: An Integer.
- k*: An Integer.

The function `BITLSHIFT(n, k)` returns the product of n and 2^k :

$$\text{intLSH}(n, k) = n \times 2^k. \quad (4.7.13)$$

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

4.7.8 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: $c = a.\text{Div}(b)$

The function **.DivInt** can be used if the second operand is an integer: $c = a.\text{DivInt}(b)$

WorksheetFunction.**IMDIV**(*z1* As String, *z2* As String) As String

NOT YET IMPLEMENTED

The function WorksheetFunction.**IMDIV** returns the quotient of *z1* and *z2*

Parameters:

z1: A Strings representing a complex number.

z2: A Strings representing a complex number.

The function **cplxDiv**(*z1*, *z2*) returns the quotient of *z1* and *z2*:

$$\frac{z_1}{z_2} = \frac{x_1x_2 + y_1y_2 + i(x_2y_1 - x_1y_2)}{x_2^2 + y_2^2} \quad (4.7.14)$$

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
```

4.7.9 Division by multiples of 2 (RSH)

WorksheetFunction.BITRSHIFT(*n* As Integer, *k* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.BITRSHIFT returns the quotient of *n* and 2^k .

Parameters:

n: An Integer.

k: An Integer.

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

$$\text{intRSH}(n, k) = n \div 2^k. \quad (4.7.15)$$

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

4.7.10 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: $c = a.\text{Mod}(b)$

WorksheetFunction.**MOD**(*x* As *mpReal*, *y* As *mpReal*) As *mpReal*

NOT YET IMPLEMENTED

The function WorksheetFunction.MOD returns the remainder of x/y

Parameters:

x: A real number.

y: A real number.

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
```

4.7.11 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: $c = a.\text{Pow}(b)$

4.8 Logical Operators

4.8.1 Bitwise AND

WorksheetFunction.**BITAND**(*n1* As Integer, *n2* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.BITAND returns n_1 bitwise-and n_2 .

Parameters:

n1: An Integer.

n2: An Integer.

4.8.2 Bitwise Inclusive OR

WorksheetFunction.**BITOR**(*n1* As Integer, *n2* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.BITOR returns n_1 bitwise-inclusive-or n_2 .

Parameters:

n1: An Integer.

n2: An Integer.

4.8.3 Bitwise Exclusive OR

WorksheetFunction.**BITXOR**(*n1* As Integer, *n2* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.BITXOR returns n_1 bitwise-exclusive-or n_2 .

Parameters:

n1: An Integer.

n2: An Integer.

4.9 Comparison Operators and Sorting

4.9.1 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`

4.9.2 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`

4.9.3 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`

4.9.4 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`

4.9.5 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

4.9.6 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

4.9.7 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 `True` if 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
```

`WorksheetFunction.GESTEP(Number As mpNum, Step As mpNum) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.GESTEP` returns 1 if $number \geq step$; returns 0 (zero) otherwise. Use this function to filter a set of values. For example, by summing several `GESTEP` functions you calculate the count of values that exceed a threshold.

Parameters:

Number: The value to test against step.

Step: The threshold value. If you omit a value for step, `GESTEP` uses zero.

`WorksheetFunction.DELTA(Number1 As mpNum, Number2 As mpNum) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.DELTA` returns 1 if `Number1` = `Number2`; returns 0 otherwise.

Parameters:

`Number1`: The first number to compare.

`Number2`: The second number to compare.

Use this function to filter a set of values. For example, by summing several `DELTA` functions you calculate the count of equal pairs. This function is also known as the Kronecker Delta function.

4.10 Properties of numbers

4.10.1 Testing for special values

WorksheetFunction.ISNUMBER(*x* As *mpNum*) As Boolean

NOT YET IMPLEMENTED

The function WorksheetFunction.ISNUMBER returns TRUE if *x* is an ordinary number (i.e. neither NaN nor an infinity), and FALSE otherwise.

Parameter:

x: A real number.

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

```

4.10.2 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

```

WorksheetFunction.**ISEVEN**(*x* As *mpNum*) As Boolean

NOT YET IMPLEMENTED

The function WorksheetFunction.ISEVEN returns TRUE if *n* is an even integer, and FALSE otherwise.

Parameter:

x: A real number.

WorksheetFunction.**ISODD**(*x* As *mpNum*) As Boolean

NOT YET IMPLEMENTED

The function `WorksheetFunction.ISODD` returns TRUE if n is an odd integer, and FALSE otherwise.

Parameter:

x : A real number.

4.10.3 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 Return (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
```

4.11 Number generation

4.11.1 Random numbers

WorksheetFunction.RAND() As mpNum

The function WorksheetFunction.RAND returns an evenly distributed random real number greater than or equal to 0 and less than 1.

Returns an evenly distributed random real number greater than or equal to 0 and less than 1. To generate a random real number between a and b, use: RAND()*(b-a)+a.

WorksheetFunction.RANDBETWEEN(*Bottom* As mpNum, *Top* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.RANDBETWEEN returns a random integer number between the numbers you specify.

Parameters:

Bottom: The smallest integer RANDBETWEEN will return.

Top: The largest integer RANDBETWEEN will return.

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.

4.11.2 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
```

4.11.3 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')]
```

4.12 Matrices

4.12.1 Basic methods

WorksheetFunction.**MUNIT**(*n* As Integer) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.MUNIT returns the unity matrix of dimension *n*.

Parameter:

n: An integer specifying the dimension of the matrix.

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']])

```

4.12.2 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')

```

4.12.3 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')]]
```

4.12.4 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]])
```

4.12.5 Matrix Properties

Add a note on matrix properties:

Rows, Cols, T etc.

4.12.6 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']])

```

4.12.7 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: $c = a.Times(b)$

WorksheetFunction.MMULT(X As mpNum[], Y As mpNum[]) As mpNum[]

NOT YET IMPLEMENTED

The function WorksheetFunction.MMULT returns the matrix product of two arrays X and Y . The result is an array with the same number of rows as X and the same number of columns as Y .

Parameters:

X : A matrix of real numbers.

Y : A matrix of real numbers.

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

The function MatrixMul 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. 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]])
```

4.12.8 Python-specific convenience functions

4.12.8.1 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 π 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
```

4.12.8.2 `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.

4.12.8.3 `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.

4.12.8.4 `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.

4.12.8.5 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.

4.12.8.6 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
```

4.12.8.7 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
```

4.12.8.8 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('0.99999999906867742538452148'),) {}
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('3.1415076583334066'),), {})
8.49952562843408e-5
>>> print(input[4]); print(output[4])
((mpf('3.1415928201669122'),), {})
-1.66577118985331e-7
```

4.12.8.9 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')
```

WorksheetFunction.PI() As mpNum

The function WorksheetFunction.PI returns the value of $\pi = 3.1415926535897932\dots$

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...

Function `e()` As mpNum

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

Function `phi()` As mpNum

The function `phi` returns Golden ratio phi: 1.61803...

Function `euler()` As mpNum

The function `euler` returns Euler's constant: 0.577216...

Function `catalan()` As mpNum

The function `catalan` returns Catalan's constant: 0.915966...

Function `apery()` As mpNum

The function `apery` returns Apery's constant: 1.20206...

Function `khinchin()` As mpNum

The function `khinchin` returns Khinchin's constant: 2.68545...

Function `glaisher()` As mpNum

The function `glaisher` returns Glaisher's constant: 1.28243...

Function `mertens()` As mpNum

The function `mertens` returns Mertens' constant: 0.261497...

Function `twinprime()` As mpNum

The function `twinprime` returns Twin prime constant: 0.660162...

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 $e^z = \exp(z)$

WorksheetFunction.**EXP**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.EXP returns the value of the exponential function, $\exp(x) = e^x = \exp(x)$.

Parameter:

x: A real number.

WorksheetFunction.**IMEXP**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

The function WorksheetFunction.IMEXP returns the complex exponential of *z*, as a String representing a complex number.

Parameter:

z: A String representing a complex number.

The function IMEXP(*z*) returns the complex exponential function of *z*:

$$\exp(z) = e^x \cos(y) + i e^x \sin(y). \quad (5.2.1)$$

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,

$$\exp(x) = e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}. \quad (5.2.2)$$

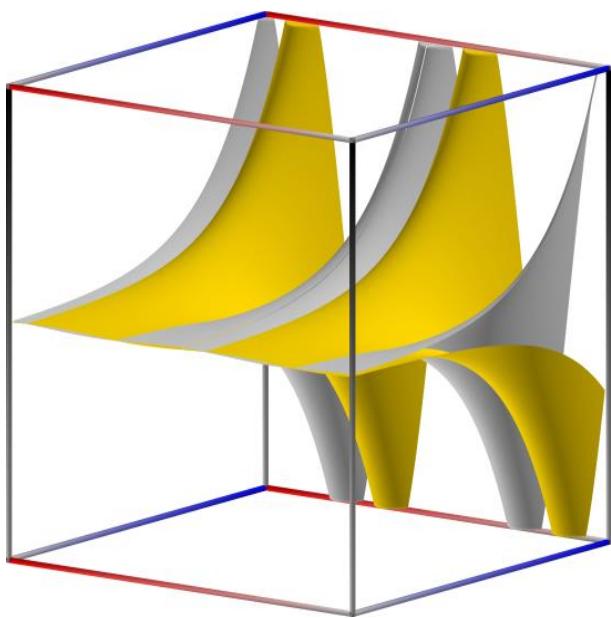
For complex numbers, the exponential function also satisfies

$$\exp(x + yi) = e^x(\cos(y) + i \sin(y)). \quad (5.2.3)$$

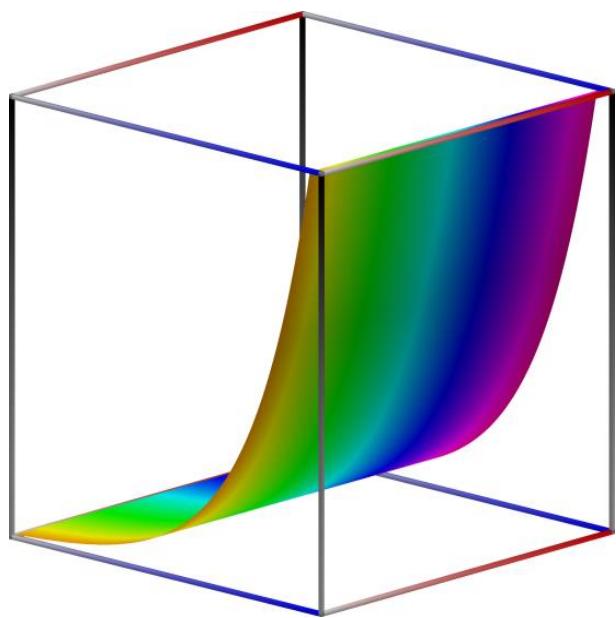
Basic examples

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
```



(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 ± 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.

```
>>> exp(inf)
+inf
>>> exp(-inf)
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 $\exp(jx)$

Function **expj(z As mpNum)** As mpNum

The function \expj returns 10^z

Parameter:

z : A complex number.

Convenience function for computing e^{ix} :

```
>>> 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(z)`

Function `expjpi(z As mpNum) As mpNum`

The function `expjpi` returns 10^z

Parameter:

`z`: A complex number.

Convenience function for computing $e^{i\pi z}$. 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 $\ln(x) = \log_e(x)$

WorksheetFunction.**LN**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.LN returns the value of the natural logarithm $\ln(x) = \log_e(x)$.

Parameter:

x: A real number.

WorksheetFunction.**IMLN**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

The function WorksheetFunction.ILog returns the complex natural logarithm of *z*, as a String representing a complex number.

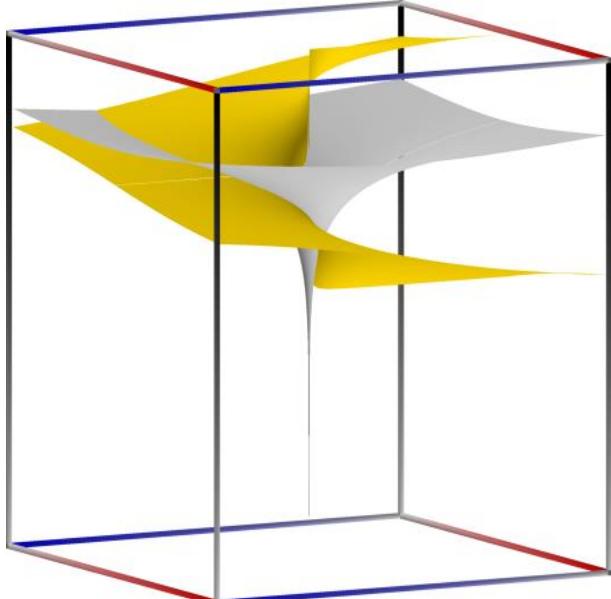
Parameter:

z: A String representing a complex number.

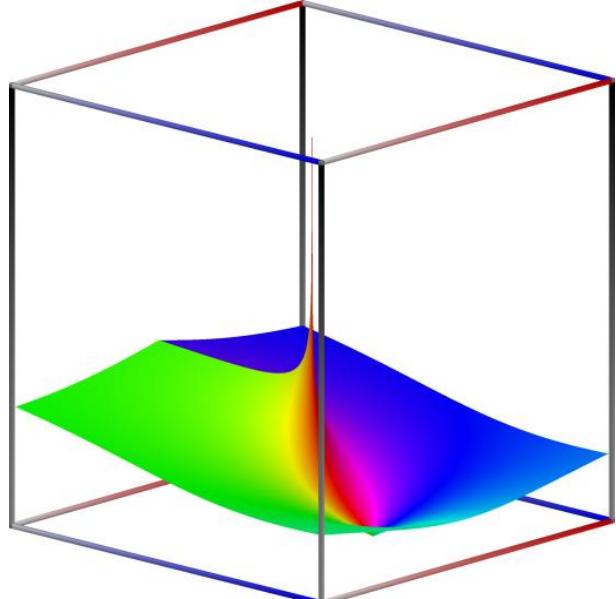
The function **cplxLn**(*z*) returns the complex natural logarithm of *z*:

$$\ln(z) = \log_e(z) = \ln(r) + i\theta, \quad (5.2.4)$$

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 ± 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)`

WorksheetFunction.**LOG**(*x* As *mpNum*, *b* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.LOG 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 **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)
0.3333333333333333333333333333333333333333333333333333333333
>>> mp.dps = 25
>>> log(3+4j)
(1.609437912434100374600759 + 0.9272952180016122324285125j)
```

5.2.5 Common (decadic) logarithm $\log_{10}(z)$

The function `log10(z)` returns the complex natural logarithm of z :

$$\log_{10}(z) = \ln(z)/\ln(10). \quad (5.2.5)$$

`log10(x)` is equivalent to `log(x, 10)`.

`WorksheetFunction.LOG10(x As mpNum) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.LOG10` returns the value of the decadic logarithm $\log_{10}(x) = \log_{10}(x)$.

Parameter:

`x`: A real number.

`WorksheetFunction.IMLOG10(z As String) As String`

NOT YET IMPLEMENTED

The function `WorksheetFunction.IMLOG10` returns $\log_{10}(z)$, as a String representing a complex number.

Parameter:

`z`: A String representing a complex number.

Function **log10(z As mpNum)** As mpNum

The function `log10` returns $\log_{10}(z)$

Parameter:

`z`: A complex number.

5.2.6 Binary logarithm $\log_2(z)$

`WorksheetFunction.IMLOG2(z As String)` As String

NOT YET IMPLEMENTED

The function `WorksheetFunction.IMLOG2` returns $\log_2(z)$, as a String representing a complex number.

Parameter:

`z`: A String representing a complex number.

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 **lnp1(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*

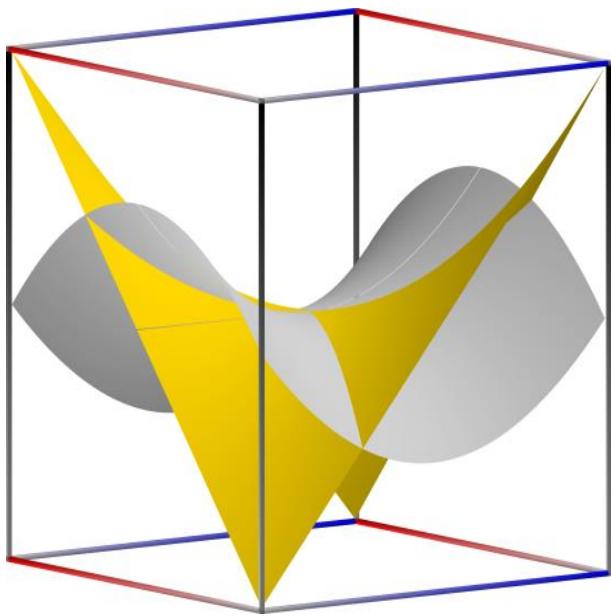
The function **square** returns the square of *z*.

Parameter:

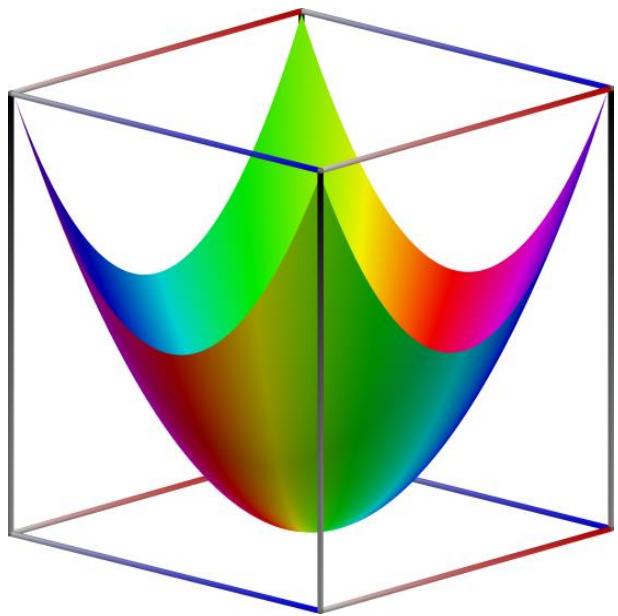
z: A complex number.

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 ± 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

WorksheetFunction.**POWER**(*x* As mpNum, *y* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.POWER returns the value of x^y , $y \in \mathbb{R}$.

Parameters:

x: A real number.

y: A real number.

WorksheetFunction.**IMPOWER**(*z* As String, *k* As Integer) As String

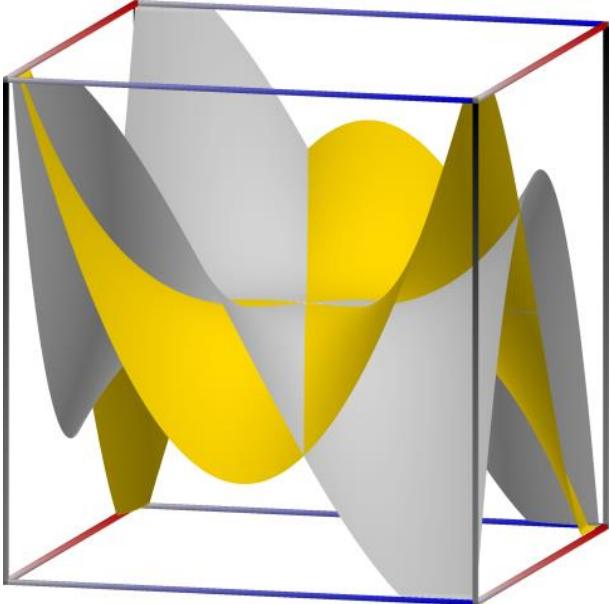
NOT YET IMPLEMENTED

The function WorksheetFunction.IMPOWER returns an integer power of *z*, as a String representing a complex number.

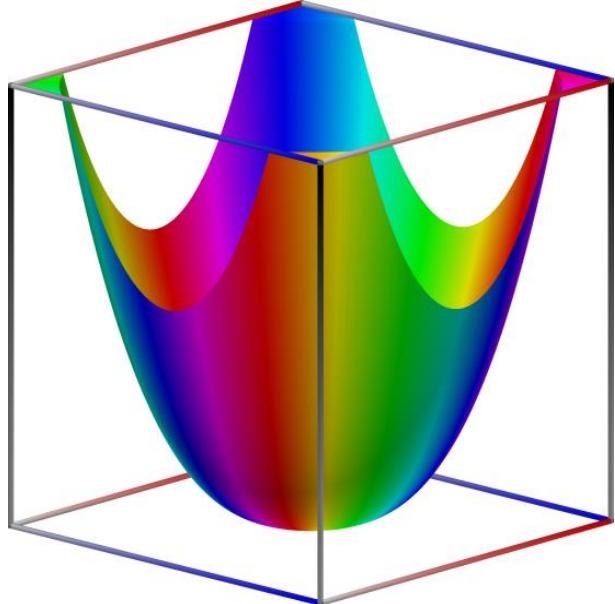
Parameters:

z: A String representing a complex number.

k: An integer.



(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 ± 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*:

$$z_1^{z_2} = \exp(\ln(z_1)z_2), \quad z_1, z_2 \in \mathbb{C}. \quad (5.3.4)$$

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}

WorksheetFunction.**SQRT**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.SQRT returns the absolute value of the square root of x , \sqrt{x} .

Parameter:

x: A real number.

WorksheetFunction.**IMSQRT**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

The function WorksheetFunction.IMSQRT returns the square root of z , as a String representing a complex number.

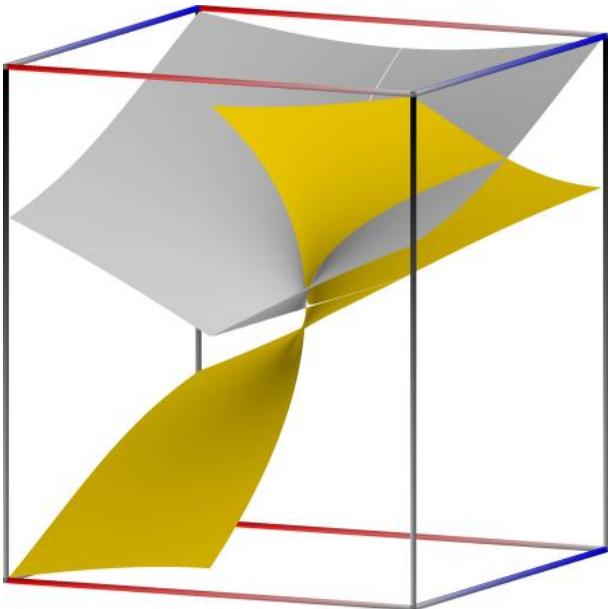
Parameter:

z: A String representing a complex number.

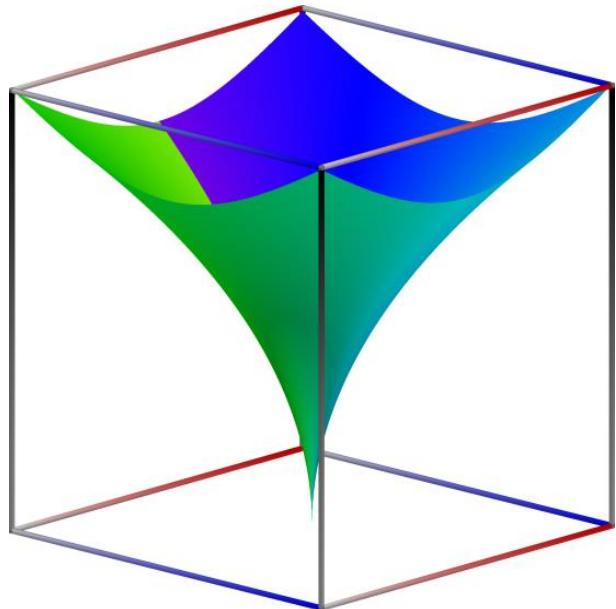
The function *cplxSqrt*(*z*) returns the square root of z :

$$\sqrt{z} = \sqrt{r} \cos\left(\frac{1}{2}\theta\right) + i\sqrt{r} \sin\left(\frac{1}{2}\theta\right), \quad (5.3.5)$$

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 ± 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
[1.999999999999995559, 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: $\sqrt[3]{x}, n = 2, 3, \dots$ **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 (ζ_k and ζ_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

WorksheetFunction.**DEGREES**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DEGREES returns the value of *x* converted to degrees, with the input *x* in radians.

Parameter:

x: A real number. The following formula is used:

$$\text{Degrees}(x) = x \cdot \frac{180}{\pi}, \quad (5.4.2)$$

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
```

WorksheetFunction.**RADIANS**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.RADIANS returns the value of *x* converted to radians, with the input *x* in degrees.

Parameter:

x: A real number.

The following formula is used:

$$\text{Radians}(x) = x \cdot \frac{\pi}{180}. \quad (5.4.3)$$

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 SQRTPI

WorksheetFunction.SQRTPI(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.SQRTPI` returns the value of $\sqrt{n \cdot \pi}$.

Parameter:

x: A real number.

5.4.4 Sine: $\sin(z)$

WorksheetFunction.**SIN**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.SIN returns the value of the sine of x , with x in radians.

Parameter:

x: A real number.

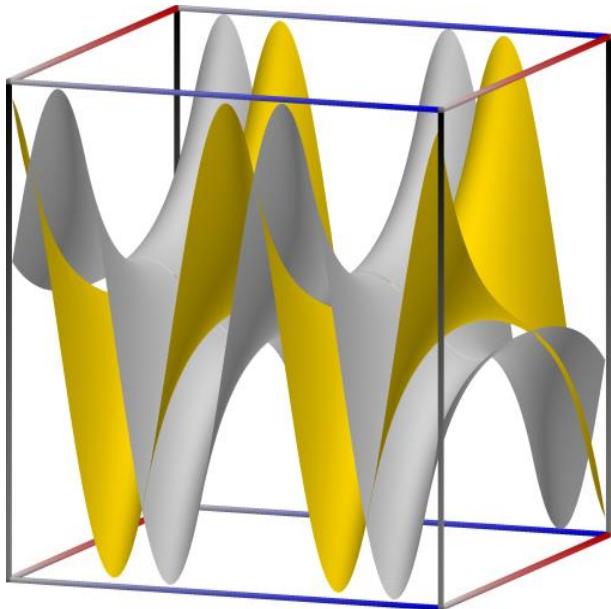
WorksheetFunction.**IMSIN**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

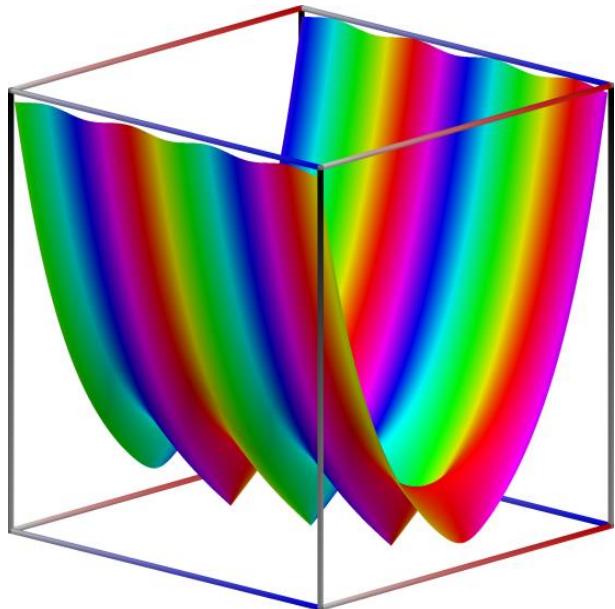
The function WorksheetFunction.IMSIN returns complex sine of z , as a String representing a complex number.

Parameter:

z: A String representing 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 5.6: 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 ± 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 :

$$\sin(z) = \sin(x) \cosh(y) + i \cos(x) \sinh(y). \quad (5.4.4)$$

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 π , 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')
>>>
```

5.4.5 Cosine: $\cos(z)$

WorksheetFunction.**COS**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COS returns the value of the cosine of *x*, with *x* in radians.

Parameter:

x: A real number.

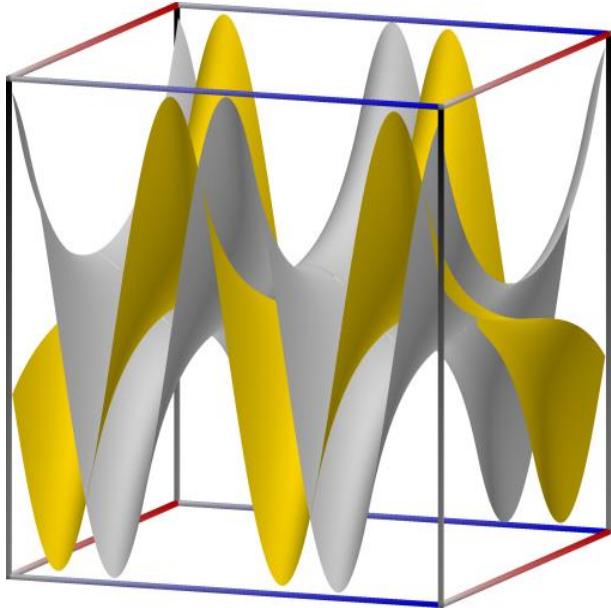
WorksheetFunction.**IMCOS**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

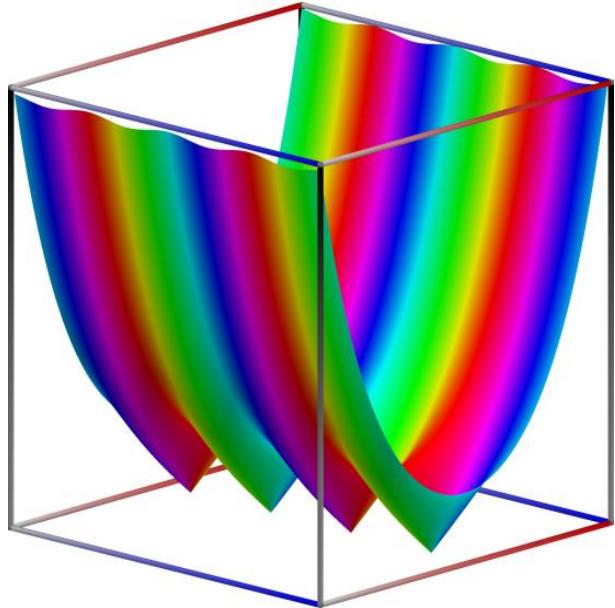
The function WorksheetFunction.IMCOS returns complex cosine of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.7: 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 ± 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 :

$$\cos(z) = \cos(x) \cosh(y) - i \sin(x) \sinh(y). \quad (5.4.5)$$

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]
```

5.4.6 Tangent: $\tan(z)$

WorksheetFunction.TAN(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.TAN returns the value of the tangent of *x*, with *x* in radians.

Parameter:

x: A real number.

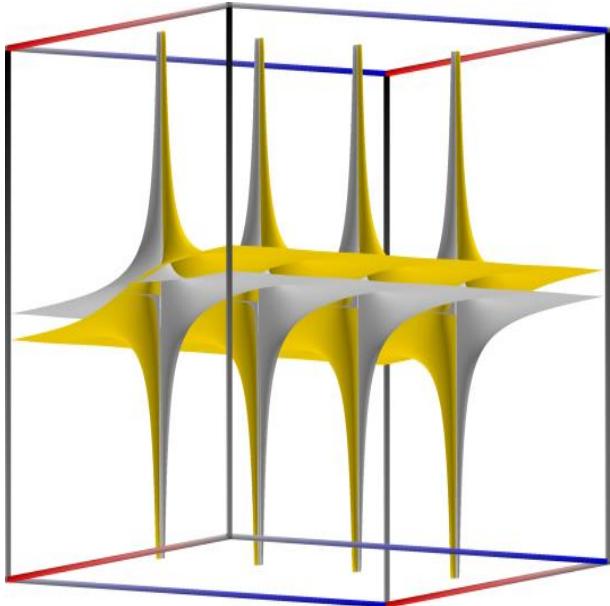
WorksheetFunction.IMTAN(*z* As String) As String

NOT YET IMPLEMENTED

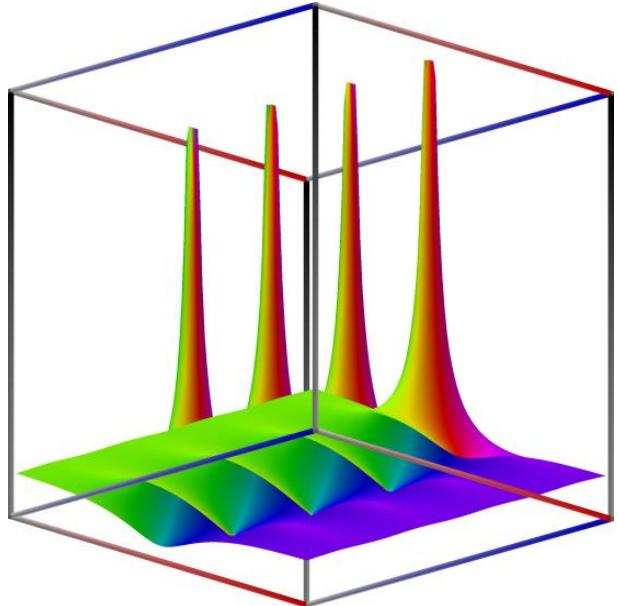
The function WorksheetFunction.IMTAN returns complex tangent of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.8: 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 ± 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 :

$$\tan(z) = \frac{\sin(z)}{\cos(z)} = \frac{\sin(2x) + i \sinh(2y)}{\cos(2x) + i \cosh(2y)} \quad (5.4.6)$$

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]
```

5.4.7 Secant: $\sec(z) = 1/\cos(z)$

WorksheetFunction.**SEC**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SEC returns the value of the secant of *x*, with *x* in radians.

Parameter:

x: A real number.

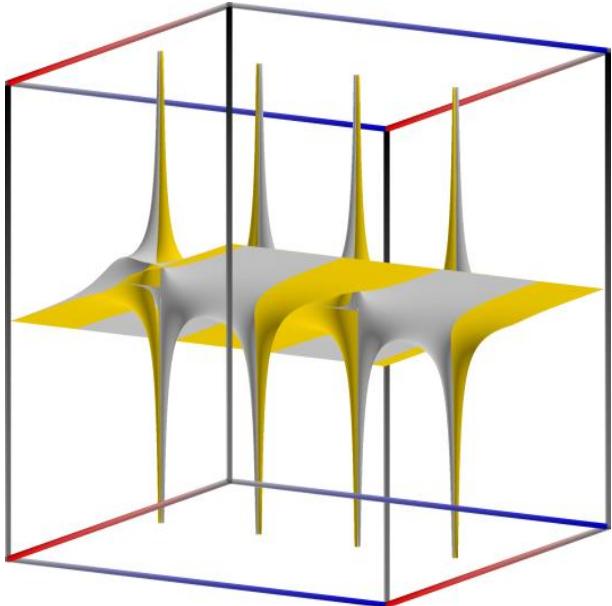
WorksheetFunction.**IMSEC**(*z* As String) As String

NOT YET IMPLEMENTED

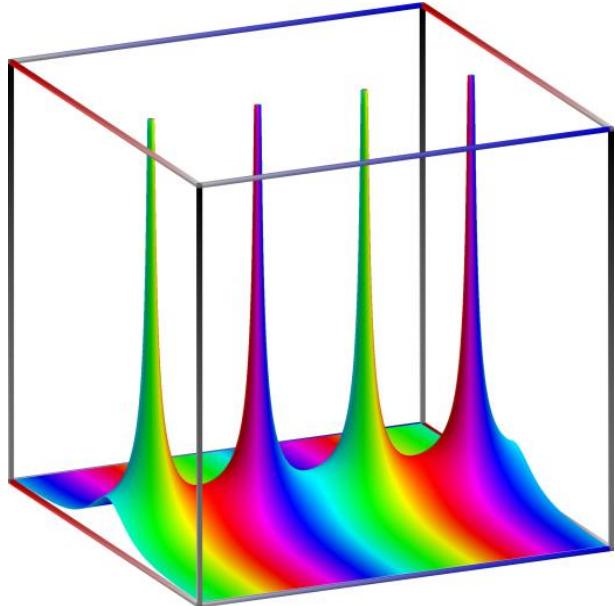
The function WorksheetFunction.IMSEC returns the complex secant of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.9: 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 ± 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 :

$$\sec(z) = 1/\cos(z). \quad (5.4.7)$$

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
>>> nprint(chop(taylor(sec, 0, 6)))
[1.0, 0.0, 0.5, 0.0, 0.208333, 0.0, 0.0847222]
```

5.4.8 Cosecant: $\csc(z) = 1/\sin(z)$

WorksheetFunction.**CSC**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.CSC returns the value of the cosecant of *x*, with *x* in radians.

Parameter:

x: A real number.

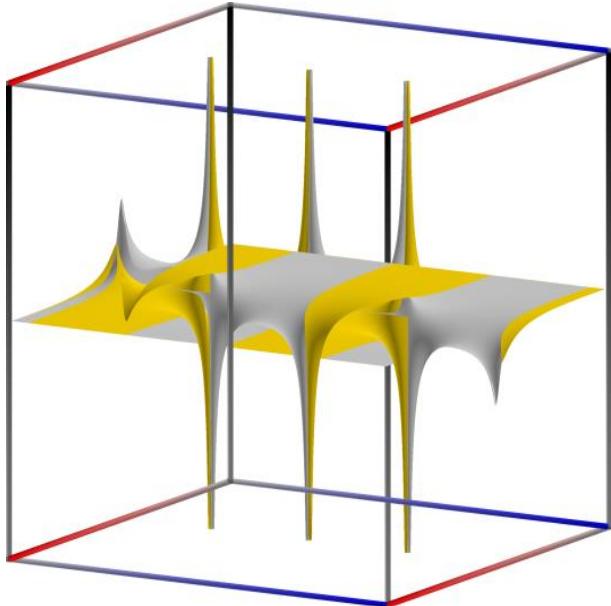
WorksheetFunction.**IMCSC**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

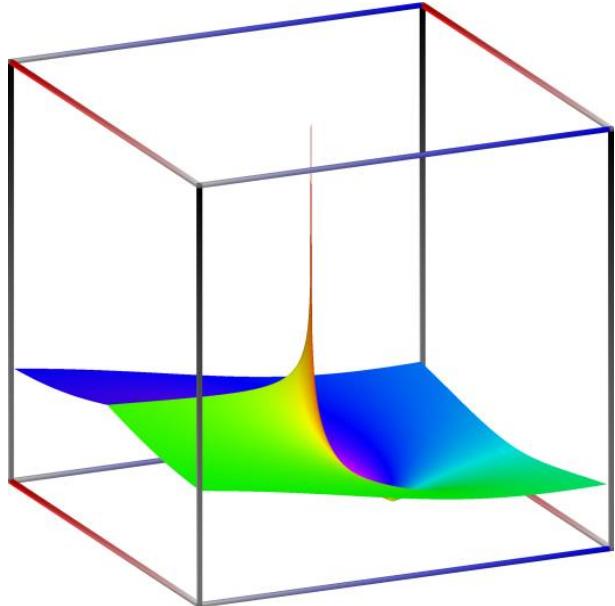
The function WorksheetFunction.IMCSC returns the complex cosecant of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.10: 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 ± 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 :

$$\sec(z) = 1/\sin(z). \quad (5.4.8)$$

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
```

5.4.9 Cotangent: $\cot(z) = 1/\tan(z)$

WorksheetFunction.**COT**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COT returns the value of the cotangent of *x*, with *x* in radians.

Parameter:

x: A real number.

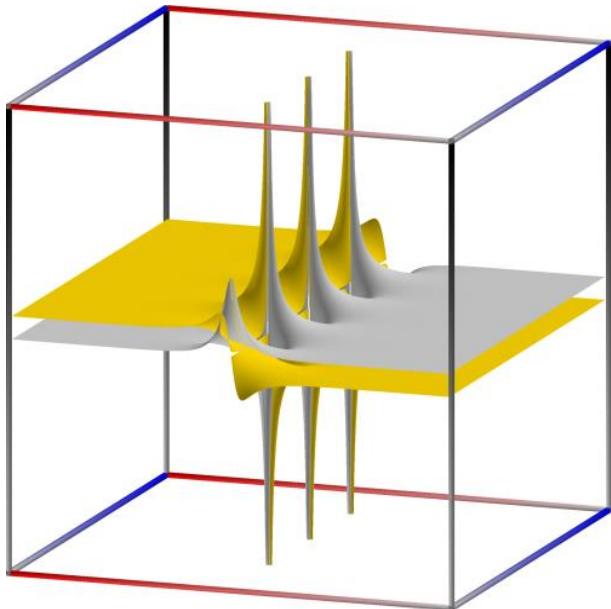
WorksheetFunction.**IMCOT**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

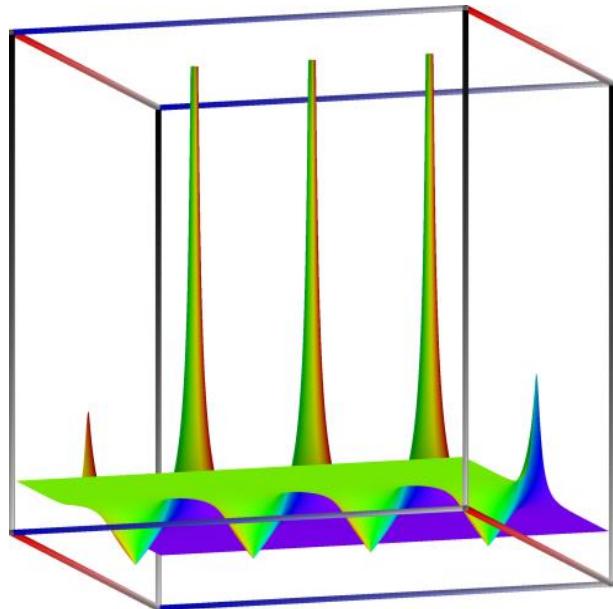
The function WorksheetFunction.IMCOT returns the complex cotangent of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.11: 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 ± 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 :

$$\cot(z) = \frac{\cos(z)}{\sin(z)} = \frac{\sin(2x) - i \sinh(2y)}{\cosh(2y) - i \cos(2x)} \quad (5.4.9)$$

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
```

5.4.10 Sinc function

5.4.10.1 `sinc(x)`

`sinc(x)` computes the unnormalized sinc function, defined as

$$\text{sinc}(x) = \begin{cases} \sin(x)/x & \text{for } x \neq 0 \\ 1 & \text{for } x = 0. \end{cases} \quad (5.4.10)$$

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
```

5.4.10.2 `sincpi(x)`

`sincpi(x)` computes the normalized sinc function, defined as

$$\text{sinc}_\pi(x) = \begin{cases} \sin(\pi x)/(\pi x) & \text{for } x \neq 0 \\ 1 & \text{for } x = 0. \end{cases} \quad (5.4.11)$$

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
```

5.4.11 Trigonometric functions with modified argument

5.4.11.1 `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)
```

5.4.11.2 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: $\sinh(z)$

WorksheetFunction.**SINH**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.SINH returns the value of the hyperbolic sine of x , with x in radians.

Parameter:

x: A real number.

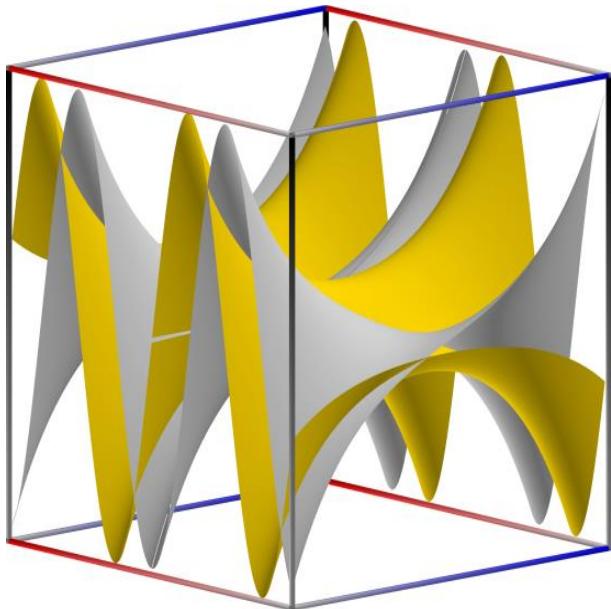
WorksheetFunction.**IMSINH**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

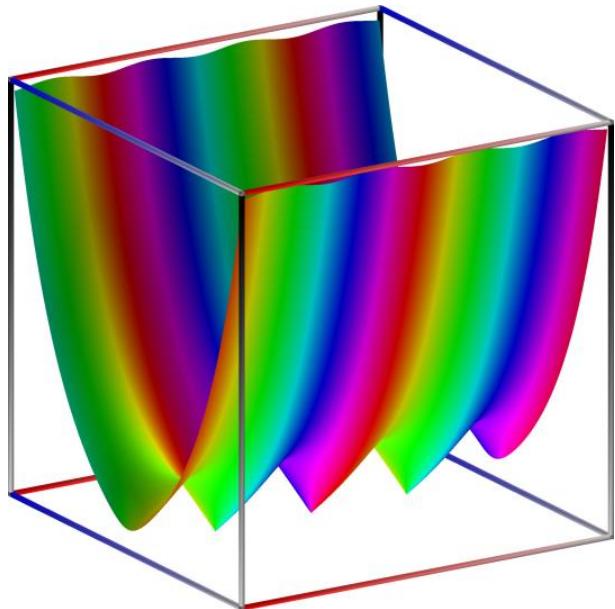
The function WorksheetFunction.IMSINH returns the complex hyperbolic sine of z , as a String representing a complex number.

Parameter:

z: A String representing 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 5.12: 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 ± 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: $\cosh(z)$

WorksheetFunction.**COSH**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COSH returns the value of the hyperbolic cosine of x , with x in radians.

Parameter:

x: A real number.

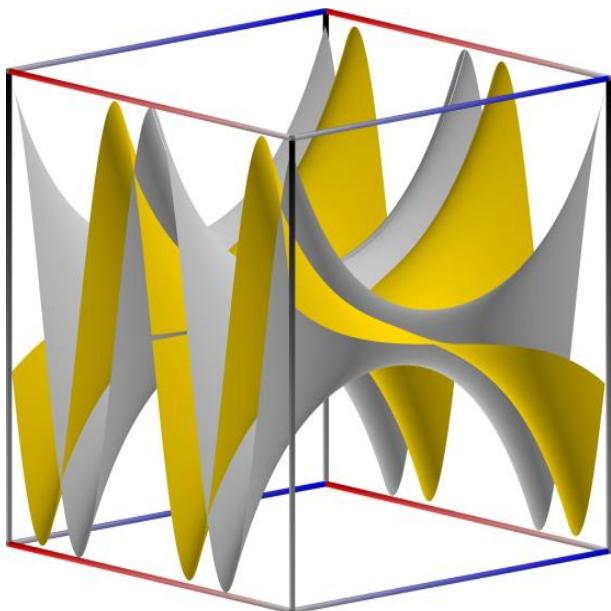
WorksheetFunction.**IMCOSH**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

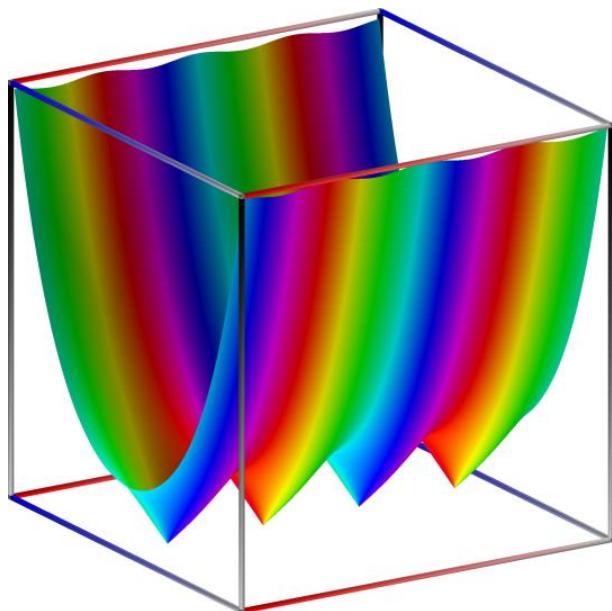
The function WorksheetFunction.IMCOSH returns the complex hyperbolic cosine of z , as a String representing a complex number.

Parameter:

z: A String representing 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 5.13: 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 ± 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: $\tanh(z)$

WorksheetFunction.TANH(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.TANH returns the value of the hyperbolic cosine of *x*, with *x* in radians.

Parameter:

x: A real number.

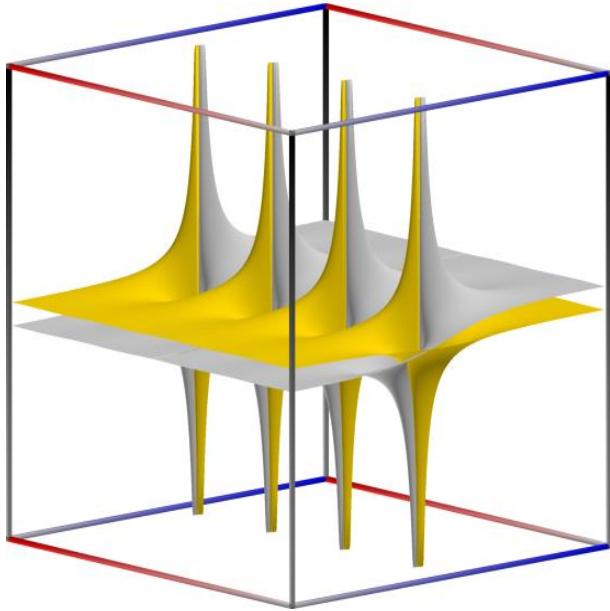
WorksheetFunction.IMTANH(*z* As String) As String

NOT YET IMPLEMENTED

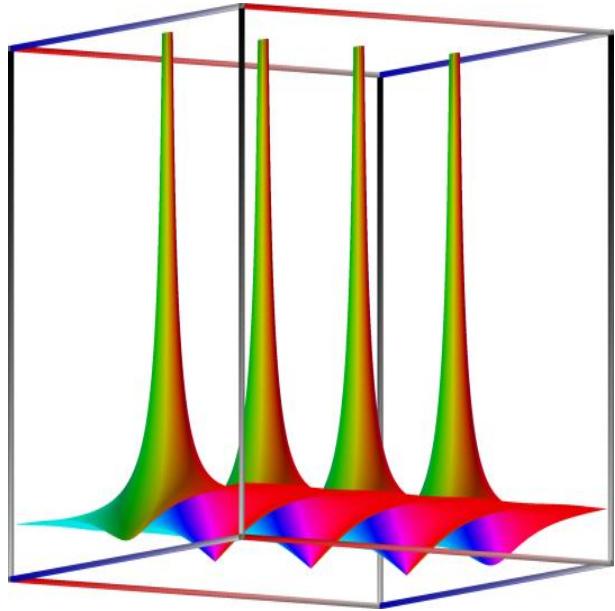
The function WorksheetFunction.IMTANH returns the complex hyperbolic tangent of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.14: 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 ± 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: $\text{sech}(x) = 1/\cosh(z)$

WorksheetFunction.**SECH**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SECH returns the value of the hyperbolic cosecant of *x*, with *x* in radians.

Parameter:

x: A real number.

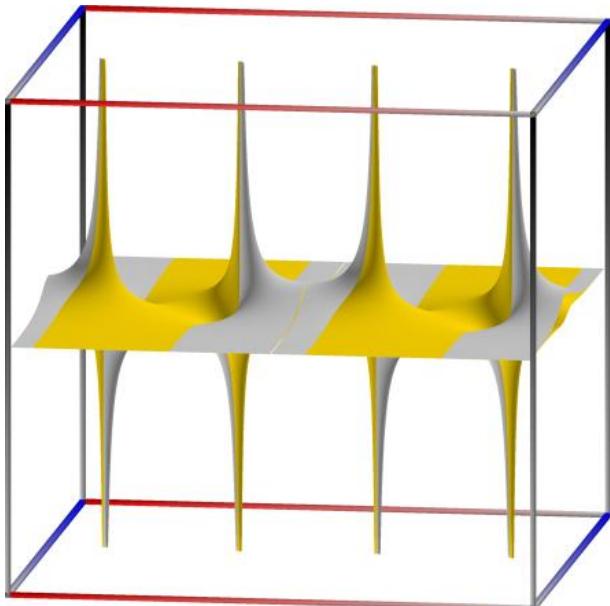
WorksheetFunction.**IMSECH**(*z* As String) As String

NOT YET IMPLEMENTED

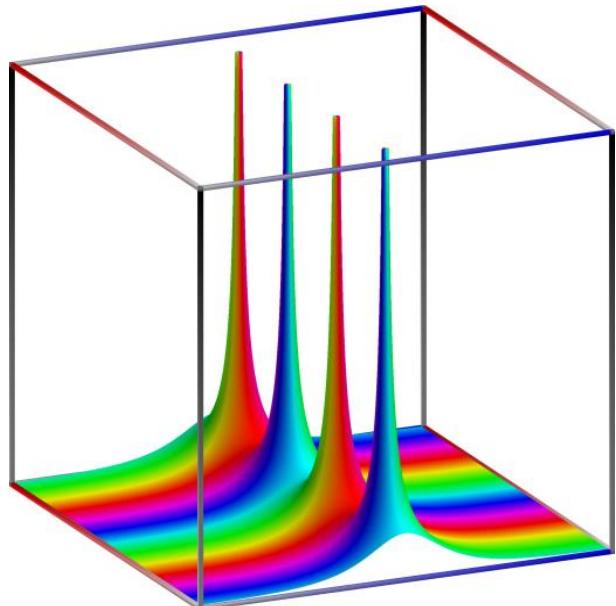
The function WorksheetFunction.IMSECH returns the complex hyperbolic secant of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.15: 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 ± 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 :

$$\operatorname{sech}(z) = 1/\cosh(z). \quad (5.5.4)$$

5.5.5 Hyperbolic Cosecant: $\text{csch}(x) = 1/\sinh(z)$

WorksheetFunction.**CSCH**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.CSCH returns the value of the hyperbolic cosecant of *x*, with *x* in radians.

Parameter:

x: A real number.

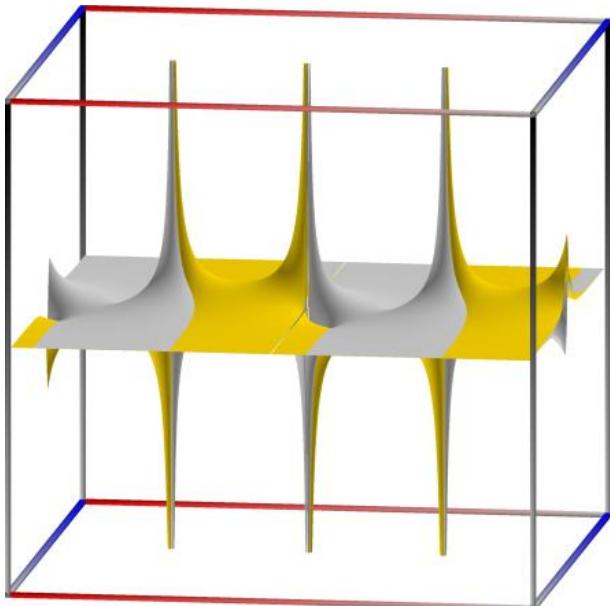
WorksheetFunction.**IMCSCH**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

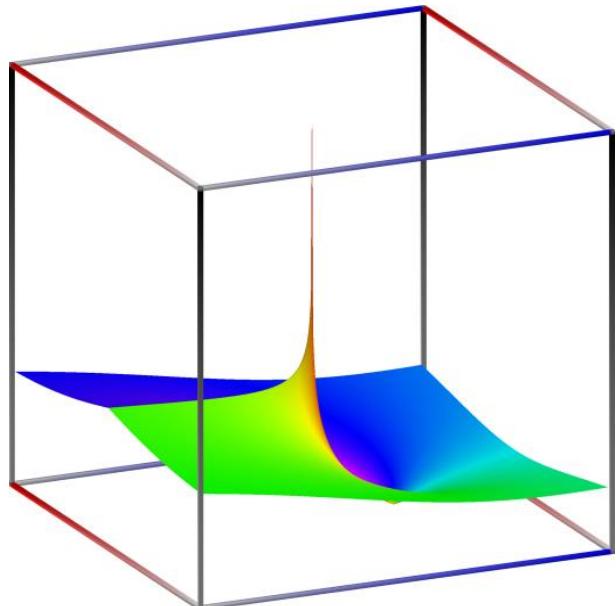
The function WorksheetFunction.IMCSCH returns the complex hyperbolic cosecant of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 $\phi = -112^\circ$. 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 ± 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 :

$$\operatorname{csch}(z) = 1 / \sinh(z). \quad (5.5.5)$$

5.5.6 Hyperbolic Cotangent: $\coth(x) = 1/\tanh(z)$

WorksheetFunction.**COTH**(*x* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COTH returns the value of the hyperbolic cotangent of *x*, with *x* in radians.

Parameter:

x: A real number.

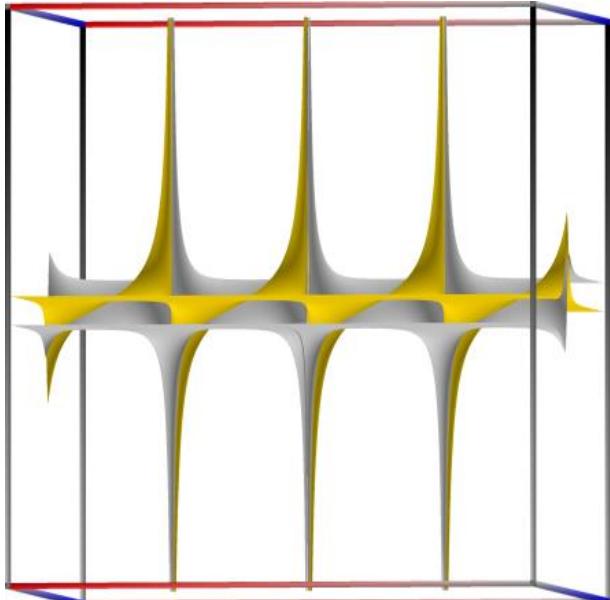
WorksheetFunction.**IMCOTH**(*z* As *String*) As *String*

NOT YET IMPLEMENTED

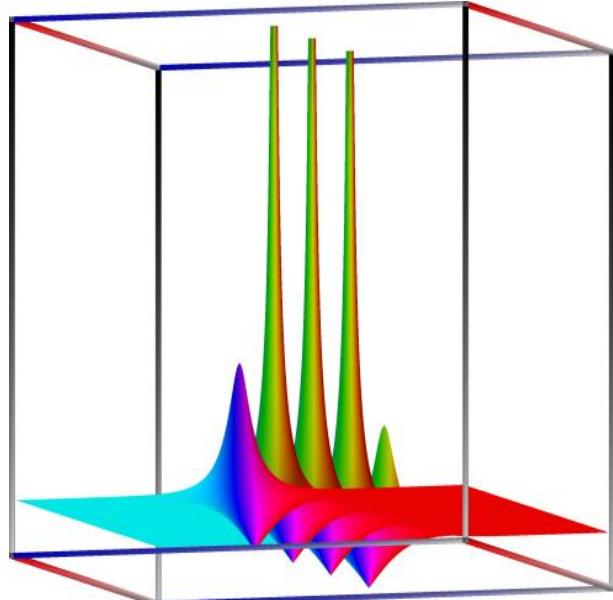
The function WorksheetFunction.IMCOTH returns the complex hyperbolic cotangent of *z*, as a String representing a complex number.

Parameter:

z: A String representing 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 5.17: 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 ± 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: $\text{asin}(z)$

WorksheetFunction.**ASIN**(x As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ASIN returns the value of the arc-sine of x in radians.

Parameter:

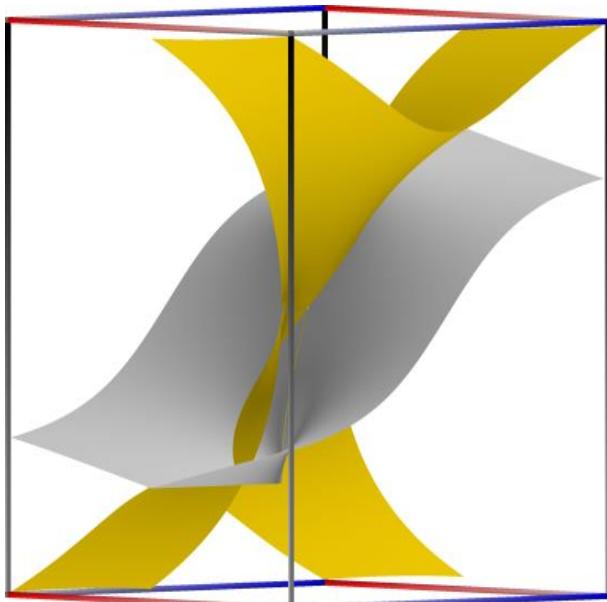
x : A real number.

Function **asin**(z As mpNum) As mpNum

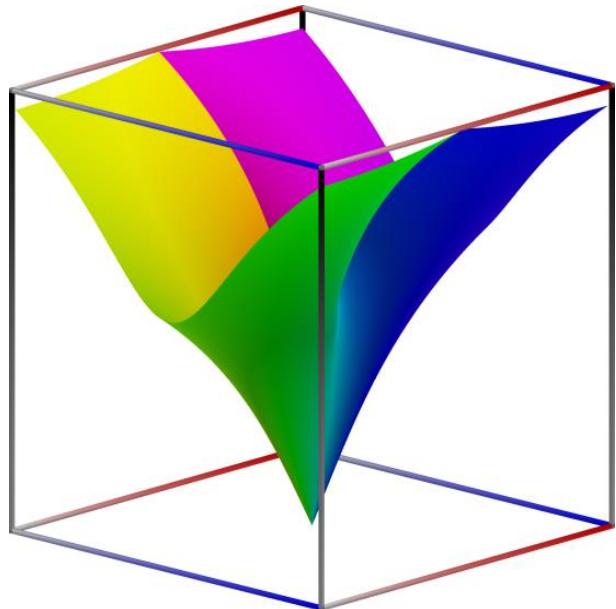
The function asin returns the inverse complex sine of z

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 ± 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 Arcosine: $\text{acos}(z)$

WorksheetFunction.ACOS(x As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ACOS returns the value of the arc-cosine of x in radians.

Parameter:

x : A real number.

Function **acos(z As mpNum)** As mpNum

The function **acos** returns the inverse complex cosine of z

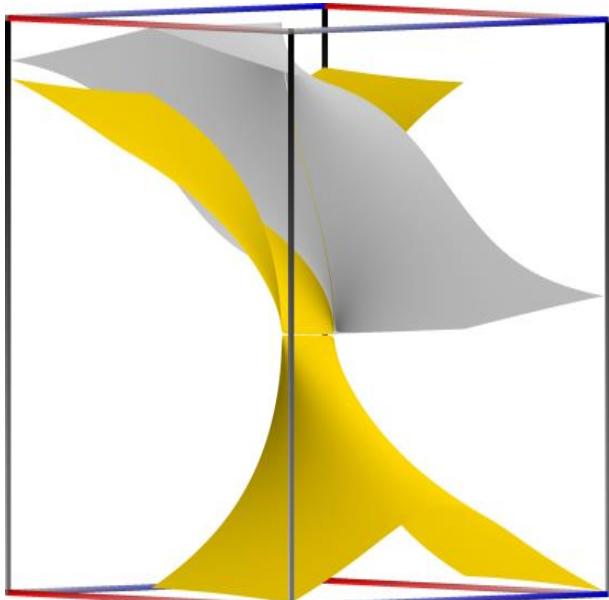
Parameter:

z : A complex number.

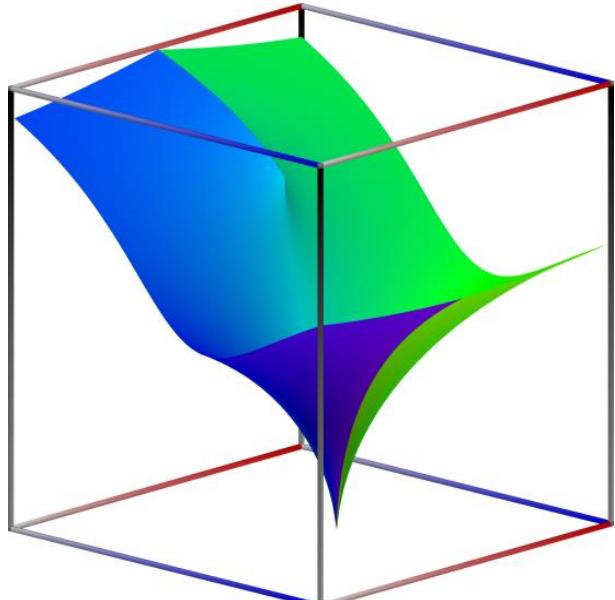
The function **cplxACos(z)** returns the inverse complex cosine of $z = x + iy$:

$$\arccos(z) = \arccos(\beta) - i \ln \left(\alpha + \sqrt{\alpha^2 - 1} \right), \quad \text{where} \quad (5.6.5)$$

α and β 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 ± 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: $\text{atan}(z)$

WorksheetFunction.ATAN(x As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ATAN returns the value of the arc-tangent of x in radians.

Parameter:

x : A real number.

Function atan(z As mpNum) As mpNum

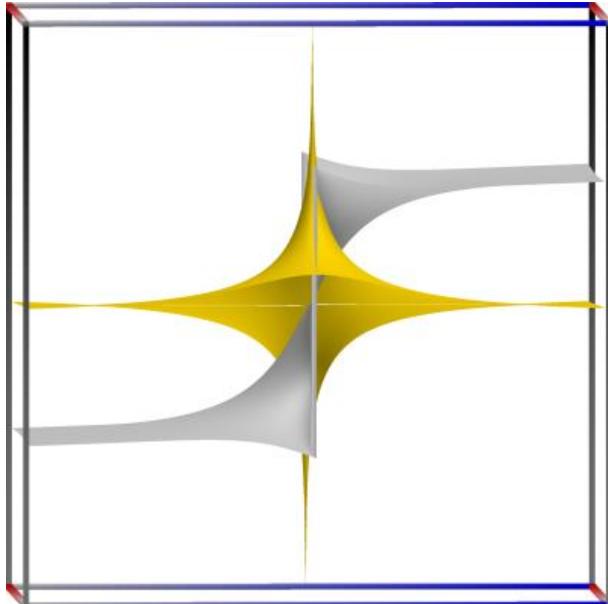
The function atan returns the inverse complex tangent of z

Parameter:

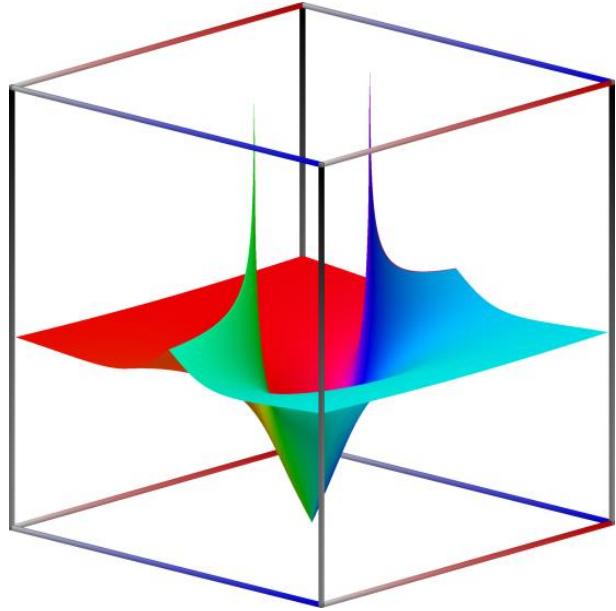
z : A complex number.

The function 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 ± 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()).

$\text{atan}()$ is defined so as to be a proper inverse function of $\tan(\theta)$ for $-\pi/2 < \theta < \pi/2$. We have $\tan(\text{atan}^{-1}(x) = x)$ for all x , but $\text{atan}^{-1}(\tan(x)) = x$ 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) (1.0000000000000000000000000001 + 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)`

WorksheetFunction.**ATAN2**(*x* As *mpNum*, *y* As *mpNum*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.ATAN2 returns the value of the arc-tangent of *x* in radians.

Parameters:

x: A real number.

y: A real number.

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: $\text{acot}(z)$

WorksheetFunction.ACOT(x As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ACOT returns the value of the arc-cotangent of x in radians.

Parameter:

x : A real number.

Function $\text{acot}(z$ As mpNum) As mpNum

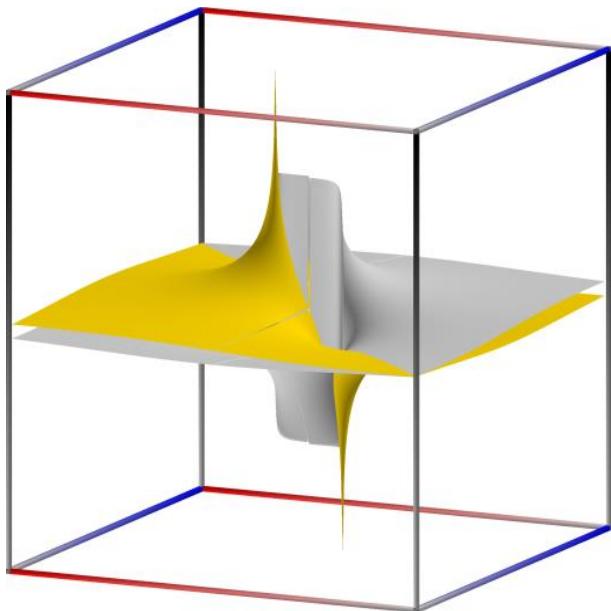
The function acot returns the inverse complex cotangent of z

Parameter:

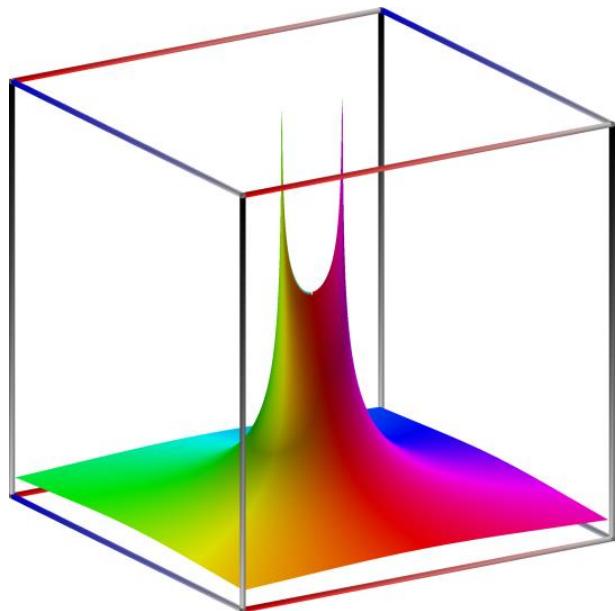
z : A complex number.

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 ± 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: $\text{asinh}(z)$

WorksheetFunction.**ASINH**(*x As mpNum*) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ASINH returns the value of the hyperbolic arc-sine of *x* in radians.

Parameter:

x: A real number.

Function **asinh**(*z As mpNum*) As mpNum

The function **asinh** returns the inverse complex hyperbolic sine of *z*

Parameter:

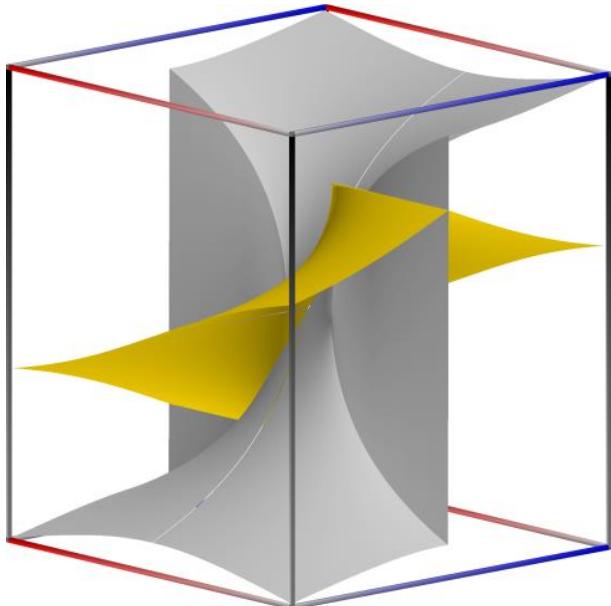
z: A complex number.

The function **cplxASinh**(*z*) returns the inverse complex hyperbolic sine of *z*:

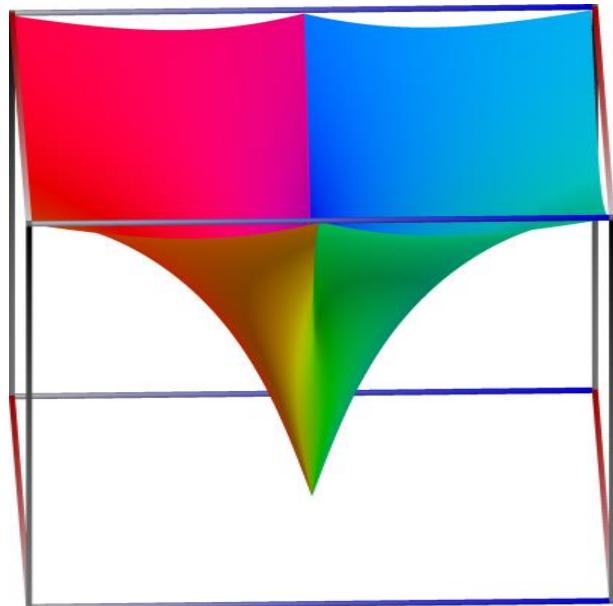
$$\text{arcsinh}(z) = -i \arcsin(iz), \quad (5.7.1)$$

where $\arcsin(z)$ 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 $z = \text{asinh}(x + iy)$, $-3 \leq x \leq 3$ (blue axis), $-2\pi \leq y \leq 2\pi$ (red axis), $z_{\min} \leq z \leq 10$ (black axis). *z* values are truncated at ± 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: $\text{acosh}(z)$

WorksheetFunction.ACOSH(x As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ACOSH returns the value of the hyperbolic arc-cosine of x in radians.

Parameter:

x : A real number.

Function **acosh(z As mpNum)** As mpNum

The function **acosh** returns the inverse complex hyperbolic cosine of z

Parameter:

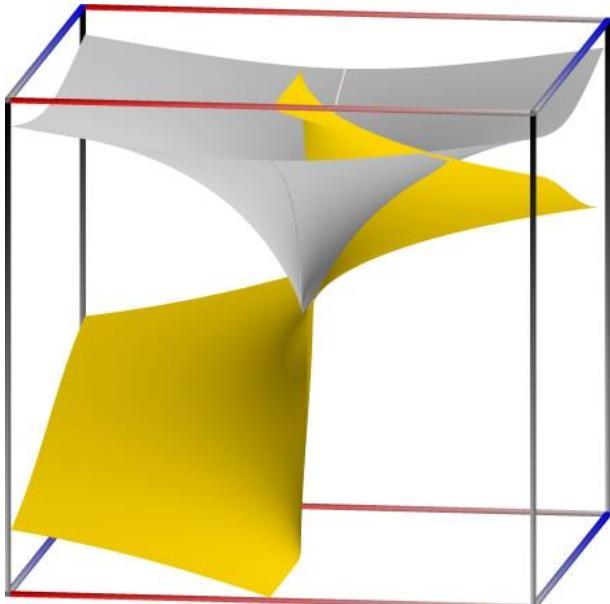
z : A complex number.

The function **cplxACosh(z)** returns the inverse complex hyperbolic cosine of z :

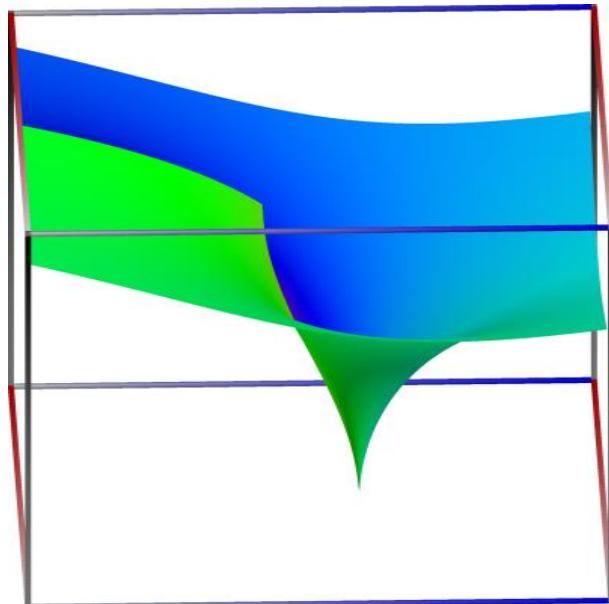
$$\text{arccosh}(z) = \pm i \arccos(z), \quad (5.7.2)$$

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 ± 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: $\operatorname{atanh}(z)$

WorksheetFunction.ATANH(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ATANH returns the value of the hyperbolic arc-tangent of *x* in radians.

Parameter:

x: A real number.

Function **atanh**(*z* As mpNum) As mpNum

The function **atanh** returns the inverse complex hyperbolic tangent of *z*

Parameter:

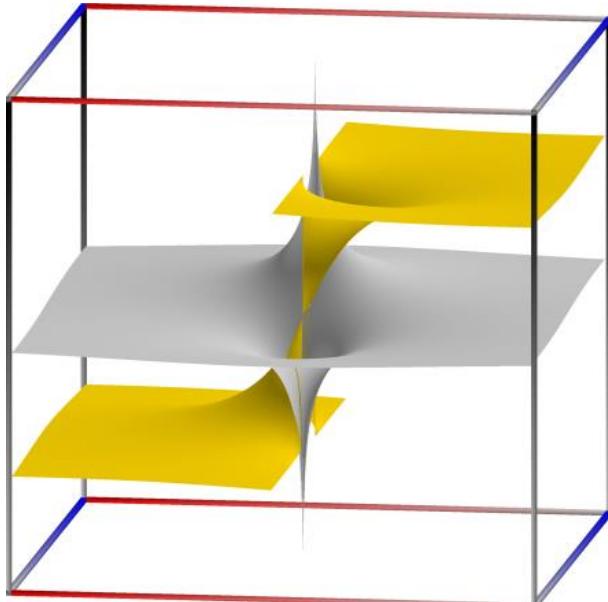
z: A complex number.

The function **cplxATanh**(*z*) returns the inverse complex hyperbolic tangent of *z*:

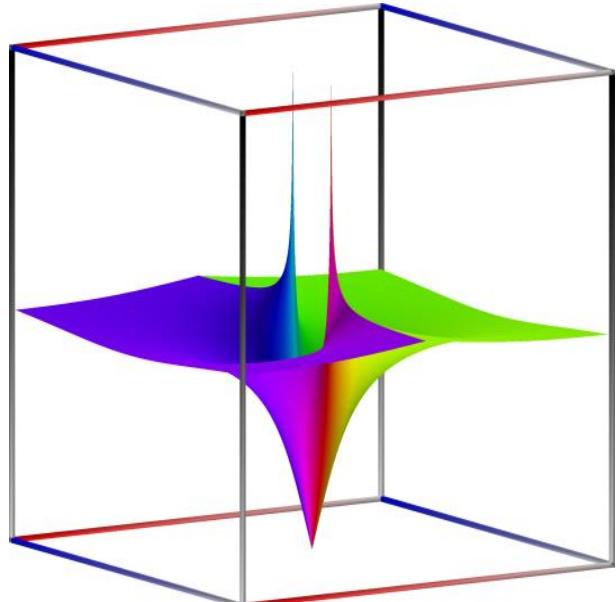
$$\operatorname{arctanh}(z) = -i \operatorname{arctan}(z), \quad (5.7.3)$$

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 ± 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: $\text{acoth}(z)$

WorksheetFunction.ACOTH(x As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ACOTH returns the value of the hyperbolic arc-cotangent of x in radians.

Parameter:

x : A real number.

Function **acoth**(z As mpNum) As mpNum

The function acoth returns the inverse complex hyperbolic cotangent of z

Parameter:

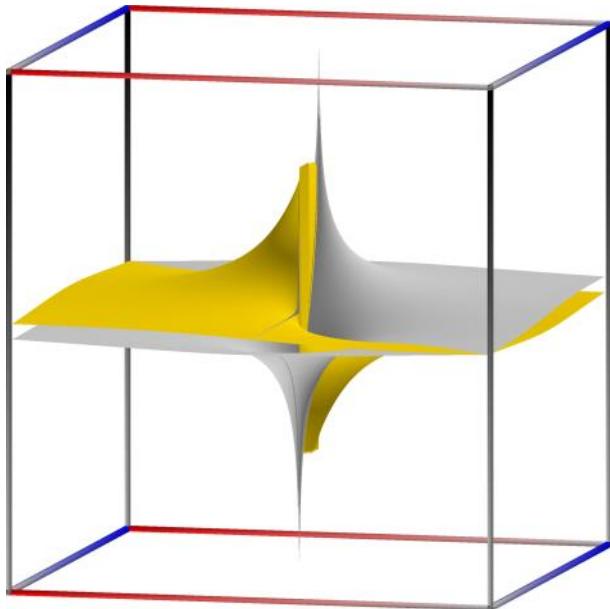
z : A complex number.

The function $\text{cplxACoth}(z)$ returns the inverse complex hyperbolic cotangent of z :

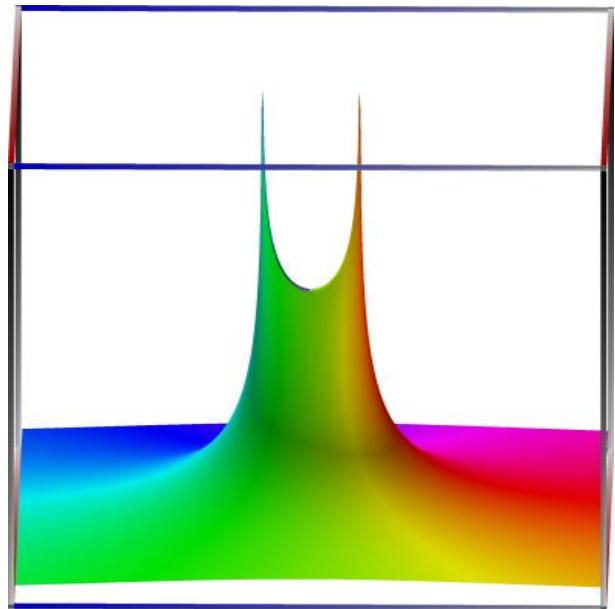
$$\text{arctanh}(z) = i \arctan(iz), \quad (5.7.4)$$

where $\arctan(z)$ 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 ± 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 asech(x)

Computes the inverse hyperbolic secant of x , $\text{sech}^{-1}(x) = \cosh^{-1}(1/x)$

5.7.6 acsch(x)

Computes the inverse hyperbolic cosecant of x , $\text{csch}^{-1}(x) = \sinh^{-1}(1/x)$

5.8 Elementary Functions of Mathematical Physics

5.8.1 Bessel Function $J_\nu(x)$

WorksheetFunction.**BESSELJ**(*x* As mpNum, ν As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.BESSELJ returns $J_\nu(z)$, the Bessel function of the first kind of real order ν .

Parameters:

x: A real number.

ν : A real number.

$J_\nu(z)$, the Bessel function of the first kind of order ν , is defined as

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{k=0}^{\infty} (-1)^k \frac{(x^2/4)^k}{k!\Gamma(\nu+k+1)} \quad (5.8.1)$$

5.8.2 Bessel Function $Y_\nu(x)$

WorksheetFunction.**BESSELY**(*x* As mpNum, ν As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.BESSELY returns $Y_\nu(z)$, the Bessel function of the second kind of order ν .

Parameters:

x: A real number.

ν : A real number.

$Y_\nu(z)$, the Bessel function of the second kind of order ν , is defined as

$$Y_\nu(x) = \frac{J_\nu(x) \cos(\nu\pi) - J_{-\nu}(x)}{\sin(\nu\pi)} \quad (5.8.2)$$

5.8.3 Bessel Function $I_\nu(x)$

WorksheetFunction.**BESSELI**(*x* As mpNum, ν As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.BESSELI returns $J_\nu(z)$, the Bessel function of the first kind of real order ν .

Parameters:

x: A real number.

ν : A real number.

This function returns the modified Bessel function $I_\nu(z)$ of the first kind of order ν , defined as

$$I_\nu(z) = \frac{z}{2} \sum_{j=0}^{\infty} \frac{(z^2/4)^j}{j!\Gamma(\nu+j+1)} \quad (5.8.3)$$

5.8.4 Bessel Function $K_\nu(x)$

WorksheetFunction.**BESSELK**(*x* As mpNum, *ν* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.BESSELK returns $K_\nu(x)$, the modified Bessel function of the second kind of order ν .

Parameters:

x: A real number.

ν: A real number.

This function returns $K_\nu(z)$, the modified Bessel function of the second kind of order ν , defined as

$$K_\nu(x) = \frac{\pi}{2} \frac{I_{-\nu}(x)) - I_\nu(x)}{\sin(\nu\pi)} \quad (5.8.4)$$

WorksheetFunction.**ERF**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ERF returns the value of the error function.

Parameter:

x: A real number.

WorksheetFunction.**ERF.PRECISE**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ERF.PRECISE returns the value of the error function.

Parameter:

x: A real number.

The error function is defined by

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt, \quad (5.8.5)$$

5.8.5 Complementary Error Function

WorksheetFunction.**ERFC**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ERFC returns the value of the complementary error function.

Parameter:

x: A real number.

WorksheetFunction.**ERFC.PRECISE**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ERFC.PRECISE returns the value of the complementary error function.

Parameter:

x: A real number.

The complementary error function is defined by

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt, \quad (5.8.6)$$

5.8.6 Gamma function $\Gamma(x)$

WorksheetFunction.**GAMMA**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.GAMMA 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

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt \quad (x > 0), \quad (5.8.7)$$

and by analytic continuation if $x < 0$, using the reflection formula

$$\Gamma(x)\Gamma(1-x) = \pi/\sin(\pi x). \quad (5.8.8)$$

WorksheetFunction.**GAMMALN**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.GAMMALN returns the logarithm of the gamma function.

Parameter:

x: A real number.

WorksheetFunction.**GAMMALN.PRECISE**(*x* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.GAMMALN.PRECISE returns the logarithm of the gamma function.

Parameter:

x: A real number.

5.8.7 Beta Function B(a,b)

Function **Beta**(*a* As mpNum, *b* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function Beta returns the Beta function.

Parameters:

a: A real number.

b: A real number.

This function computes $B(a, b)$ for $a, b \neq 0, -1, -2, \dots$

5.9 Factorial and Related Functions

5.9.1 Factorial

WorksheetFunction.**FACT**(*n* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.FACT returns $n!$, the factorial of n

Parameter:

n: An Integer.

5.9.2 Double Factorial

WorksheetFunction.**FACTDOUBLE**(*n* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunctionFACTDOUBLE returns $n!!$, the double factorial of n

Parameter:

n: An Integer.

5.9.3 Binomial Coefficient, Combinations

WorksheetFunction.**COMBIN**(*n* As Integer, *k* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.COMBIN 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

$$\binom{-n}{k} = (-1)^k \binom{n+k-1}{k}. \quad (5.9.1)$$

WorksheetFunction.**COMBINA**(*n* As Integer, *k* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.COMBINA returns the binomial coefficient

Parameters:

n: An Integer.

k: An Integer.

5.9.4 Multinomial

WorksheetFunction.**MULTINOMIAL**(*a[]* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.MULTINOMIAL returns the multinomial

Parameter:

a[]: An array of integers.

Returns the multinomial, defined as the ratio of the factorial of a sum of values to the product of factorials:

$$\text{MULTINOMIAL}(a_1, a_2, \dots, a_n) = \frac{(a_1 + a_2 + \dots + a_n)!}{a_1! a_2! \dots a_n!} \quad (5.9.2)$$

5.9.5 Permutations

WorksheetFunction.**PERMUT**(*n* As Integer, *k* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.PERMUT returns the number of permutations for a given number *k* of objects that can be selected from *n* objects.

Parameters:

n: An Integer.

k: An Integer.

WorksheetFunction.**PERMUTATIONA**(*n* As Integer, *k* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.PERMUTATIONA returns the number of permutations for a given number *k* of objects that can be selected from *n* objects.

Parameters:

n: An Integer.

k: An Integer.

Returns the number of permutations for a given number *k* of objects that can be selected from *n* objects. A permutation is any set or subset of objects or events where internal order is significant.

$$\text{PERMUT}(n, k) = \frac{n!}{(n - k)!} \quad (5.9.3)$$

5.9.6 Greatest Common Divisor (GCD)

WorksheetFunction.**GCD**(*n1* As Integer, *n2* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.GCD 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 $\text{intGcd}(0, 0) = 0$.

5.9.7 Least Common Multiple (LCM)

WorksheetFunction.**LCM**(*n1* As Integer, *n2* As Integer) As Integer

NOT YET IMPLEMENTED

The function WorksheetFunction.LCM 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.

Chapter 6

Linear Algebra

6.1 Multiple Linear Regression

6.1.1 Determinant

WorksheetFunction.MDETERM(**X** As mpNum $[][]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.MDETERM returns the matrix determinant of a numeric array X with an equal number of rows and columns.

Parameter:

X : A matrix of real numbers.

6.1.2 Inverse

WorksheetFunction.MINVERSE(**X** As mpNum $[][]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.MINVERSE returns the inverse matrix for the matrix stored in the numeric array X with an equal number of rows and columns.

Parameter:

X : A matrix of real numbers.

Negative powers will calculate the inverse:

```
>>> A**-1
matrix(
[[ '-2.0', '1.0'],
 ['1.5', '-0.5']])
>>> nprint(A * A**-1, 3)
[ 1.0 1.08e-19]
[-2.17e-19 1.0]
```

6.1.3 LinEst

WorksheetFunction.**LINEST**(*Y* As *mpNum[]*, *X* As *mpNum[]*, *Const* As Boolean, *Stats* As Boolean) As *mpNumList*

NOT YET IMPLEMENTED

The function WorksheetFunction.LINEST returns information obtained by performing multiple liner regression.

Parameters:

Y: An array of real numbers.

X: An array of real numbers.

Const: A logical value.

Stats: A logical value.

The LINEST function calculates the statistics for a line by using the "least squares" method to calculate a straight line that best fits your data, and then returns an array that describes the line.

You can also combine LINEST with other functions to calculate the statistics for other types of models that are linear in the unknown parameters, including polynomial, logarithmic, exponential, and power series. Because this function returns an array of values, it must be entered as an array formula. Instructions follow the examples in this article.

This function can also be used to perform multiple linear regression.

This function needs a detailed explanation.

6.1.4 TREND

WorksheetFunction.**TREND**(*Y* As *mpNum[]*, *X* As *mpNum[]*, *NewX* As *mpNum[]*, *Const* As Boolean) As *mpNumList*

NOT YET IMPLEMENTED

The function WorksheetFunction.TREND returns values along a linear trend.

Parameters:

Y: An array of real numbers.

X: An array of real numbers.

NewX: An array of real numbers.

Const: A logical value.

Returns values along a linear trend. Fits a straight line (using the method of least squares) to the arrays *Y* and *X*. Returns the y-values along that line for the array of *NewX* that you specify. For information about how Microsoft Excel fits a line to data, see LINEST. You can use TREND for polynomial curve fitting by regressing against the same variable raised to different powers. For example, suppose column A contains y-values and column B contains x-values. You can enter x^2 in column C, x^3 in column D, and so on, and then regress columns B through D against column A.

Formulas that return arrays must be entered as array formulas.

6.2 Exponential Growth Curves

6.2.1 LogEst

WorksheetFunction.**LOGEST**(*Y* As *mpNum[]*, *X* As *mpNum[]*, *Const* As Boolean, *Stats* As Boolean) As *mpNumList*

NOT YET IMPLEMENTED

The function WorksheetFunction.LOGEST returns an exponential curve that fits your data and returns an array of values that describes the curve.

Parameters:

Y: An array of real numbers.

X: An array of real numbers.

Const: A logical value.

Stats: A logical value.

In regression analysis, calculates an exponential curve that fits your data and returns an array of values that describes the curve. Because this function returns an array of values, it must be entered as an array formula.

This function does not perform a non-linear estimation.

6.2.2 Growth

WorksheetFunction.**GROWTH**(*Y* As *mpNum[]*, *X* As *mpNum[]*, *NewX* As *mpNum[]*, *Const* As Boolean) As *mpNumList*

NOT YET IMPLEMENTED

The function WorksheetFunction.GROWTH returns predicted exponential growth by using existing data.

Parameters:

Y: An array of real numbers.

X: An array of real numbers.

NewX: An array of real numbers.

Const: A logical value.

GROWTH returns the *y*-values for a series of new *x*-values that you specify by using existing *x*-values and *y*-values. You can also use the GROWTH worksheet function to fit an exponential curve to existing *x*-values and *y*-values.

6.3 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

$$\left(\sum_k |x_k|^p \right)^{1/p}, \quad (6.3.1)$$

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= ∞ 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 ' ∞ ' 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*= ∞ 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*= ∞ are supported:

p=1 gives the 1-norm (maximal column sum)

`p=inf` gives the ∞ -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

$$\frac{1}{\sqrt{\text{rank}(A)}} \|A\|_F \leq \|A\|_2 \leq \|A\|_F. \quad (6.3.2)$$

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')
```

6.4 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
```

6.5 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.

6.6 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]
```

Chapter 7

Distribution Functions

7.1 Introduction to Distribution Functions

The following references have been used in this chapter:

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 \(2008\)](#)

7.1.1 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.

7.1.2 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 $P(1) = p(1)$, $Q(1) = 1 - p(1)$.

7.1.3 Commonly Used Function Types

7.1.3.1 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, ν , the degrees of freedom. Therefore, the function has the form

`TDist(x As nmNum; ν 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
```

7.1.3.2 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, ν , the degrees of freedom. Therefore, the function has the form

`TDistInv(Prob As mpNum; ν 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`

7.1.3.3 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, ν , the degrees of freedom. Therefore, the function has the form

`TDistInfo(ν 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
```

7.1.3.4 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 ν , 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
```

7.1.3.5 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 δ , and the other distribution parameter is ν , 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 δ 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`

7.1.3.6 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, ν , the degrees of freedom. Therefore, the function has the form

`TDistRan(Size As Integer; ν 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`

7.2 Beta-Distribution

7.2.1 Definition

If X_1 and X_2 are independent random variables following χ^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 \(1979\)](#)

7.2.2 Density and CDF

Function **BetaDist**(*x* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function **BetaDist** 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 [7.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [7.2.2.1](#) and [7.2.2.2](#).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**BETADIST**(*x* As mpReal, *a* As mpNum, *b* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.BETADIST** returns the CDF and of the central Beta-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

a: A real number greater 0, representing the numerator degrees of freedom

b: A real number greater 0, representing the denominator degrees of freedom

WorksheetFunction.**BETA.DIST**(*x* As mpReal, *a* As mpNum, *b* As mpNum, *Cumulative* As Boolean) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.BETA.DIST** returns the CDF and of the central Beta-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

a: A real number greater 0, representing the numerator degrees of freedom

b: A real number greater 0, representing the denominator degrees of freedom

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

7.2.2.1 Density

The pdf of a variable following a central Beta-distribution with a and b degrees of freedom is given by

$$f_{\text{Beta}}(a, b, x) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1} \quad (7.2.1)$$

where $B(a, b)$ denotes the beta function (see section 5.8.7).

7.2.2.2 CDF: General formulas

The cdf of a variable following a central Beta-distribution with a and b degrees of freedom is given by

$$\Pr [X \leq x] = F_{\text{Beta}}(a, b, x) = \int_0^x f_{\text{Beta}}(a, b, t) dt \quad (7.2.2)$$

7.2.2.3 Exact cdf as continued fraction

The following representation as continued fraction is used (Peizer 1968, .1428 and 1452):

$$I(a, b, x) = \binom{n}{a} p^{b-1} q^a \frac{1}{(1 + u_1/(v_1 + u_2/(v_2 + u_3/(v_3 + \dots)))), \quad \text{where}} \quad (7.2.3)$$

$$p = (1-x), \quad q = x, \quad n = a+b-1, \quad u_1 = \frac{-(b-1)q}{p}, \quad u_{2j} = \frac{j(n+j)q}{p},$$

$$u_{2j+1} = \frac{-(a+j)(b-j-1)q}{p}, \quad v_j = a+j, \quad j = 1, 2, \dots$$

7.2.3 Quantiles

Function **BetaDistInv**(*Prob* As mpNum, *m* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function **BetaDistInv** 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 7.1.3.2 for the options for *Prob* and *Output*).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**BETAINV**(*Prob* As mpReal, *a* As mpNum, *b* As mpNum) As mpReal

The function **WorksheetFunction.BETAINV** returns the two-tailed inverse of the central Beta-distribution

Parameters:

Prob: 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

WorksheetFunction.BETA.INV(*Prob* As mpReal, *a* As mpNum, *b* As mpNum) As mpReal

The function WorksheetFunction.BETA.INV returns the left-tailed inverse of the central Beta-distribution

Parameters:

Prob: 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

7.2.4 Properties

Function BetaDistInfo(*a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function BetaDistInfo 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.2.4.1 Moments: algorithms and formulas

The raw moments are given by:

$$E^h(W) = \frac{\Gamma(a+h)\Gamma(a+b)}{\Gamma(a)\Gamma(a+b+h)} \quad (7.2.4)$$

The raw moments of the power of a beta variable are given by:

$$E^h(W^s) = \frac{\Gamma(a+hs)\Gamma(a+b)}{\Gamma(a)\Gamma(a+b+hs)} \quad (7.2.5)$$

7.2.4.2 Recurrences

$$I(a, b; x) = 1 - I(b, a; 1 - x) \quad (7.2.6)$$

$$I(a, b; x) = \binom{n}{a} x^a (1 - x)^{b-1} + I(a + 1, b - 1; x) \quad (7.2.7)$$

$$I(a, b; x) = \binom{n}{a} x^a (1 - x)^b + I(a + 1, b; x) \quad (7.2.8)$$

$$I(a, b + 1; x) = \binom{n}{a} x^a (1 - x)^b + I(a, b; x) \quad (7.2.9)$$

$$I(a, b; x) = \binom{n}{a+b} x^a (1 - x)^b \frac{a}{a+b-x} + I(a + 1, b + 1; x) \quad (7.2.10)$$

$$I(a, b; x) = F\left(2a, 2b, \frac{nx}{m - mx}\right) \quad (7.2.11)$$

7.2.5 Random Numbers

Function **BetaDistRandom**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

7.2.5.1 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 ξ 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](#)).

7.3 Binomial Distribution

These functions return PMF and CDF of the (discrete) binomial distribution with number of trials $n \geq 0$ and success probability $0 \leq p \leq 1$.

7.3.1 Density and CDF

Function **BinomialDist**(*x* As *mpNum*, *n* As *mpNum*, *p* As *mpNum*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **BinomialDist** returns pdf, CDF and related information for the 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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.3.1.1 and 7.3.1.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**BINOMDIST**(*x* As *mpNum*, *n* As *mpNum*, *p* As *mpNum*, **Cumulative** As *Boolean*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.BINOMDIST** returns pdf, CDF, and related information of the 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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.**BINOM.DIST**(*x* As *mpNum*, *n* As *mpNum*, *p* As *mpNum*, **Cumulative** As *Boolean*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.BINOM.DIST** returns the CDF and pdf of the 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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.**BINOM.DIST.RANGE**(*n* As mpNum, *p* As mpNum, *x1* As mpNum, *x2* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.BINOM.DIST.RANGE returns the probability that the number of successful trials will fall between x1 and x2

Parameters:

n: The number of independent trials.

p: The probability of success on each trial

x1: The number x1 of successes in trials.

x2: The number x2 of successes in trials.

7.3.1.1 Density

$$f_{\text{Bin}}(n, k; p) = \binom{n}{k} p^k (1-p)^{n-k} = f_{\text{Beta}}(k+1, n-k+1, p) / (n+1) \quad (7.3.1)$$

7.3.1.2 CDF

$$F_{\text{Bin}}(n, k; p) = I_{1-p}(n-k, k+1) = \text{ibeta}(n-k, k+1, 1-p) \quad (7.3.2)$$

7.3.2 Quantiles

Function **BinomialDistInv**(*Prob* As mpNum, *n* As mpNum, *p* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function BinomialDistInv 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 7.1.3.2 for the options for *Prob* and *Output*.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**CRITBINOM**(*n* As mpNum, *p* As mpNum, *Alpha* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function `WorksheetFunction.CRITBINOM` returns the smallest value for which the cumulative binomial distribution is greater than or equal to a criterion value.

Parameters:

n: The number of Bernoulli trials.

p: The probability of a success on each trial.

Alpha: The criterion value.

`WorksheetFunction.BINOM.INV(n As mpNum, p As mpNum, Alpha As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.BINOM.INV` returns the smallest value for which the cumulative binomial distribution is greater than or equal to a criterion value.

Parameters:

n: The number of Bernoulli trials.

p: The probability of a success on each trial.

Alpha: The criterion value.

7.3.3 Properties

Function `BinomialDistInfo(n As mpNum, p As mpNum, Output As String) As mpNumList`

NOT YET IMPLEMENTED

The function `BinomialDistInfo` 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.3.3.1 Moments: algorithms and formulas

$$\mu'_r = \sum_{i=0}^r \binom{n}{i} \left(\sum_{j=0}^i \binom{i}{j} (-1)^j (i-j)^r \right) \quad (7.3.3)$$

$$\mu_1 = np \quad (7.3.4)$$

$$\mu_2 = np(1-p) = npq \quad (7.3.5)$$

$$\mu_3 = npq(q-p) \quad (7.3.6)$$

$$\mu_4 = 3(npq)^3 + npq(1-6pq) \quad (7.3.7)$$

7.3.4 Random Numbers

Function **BinomialDistRandom**(*Size* As *mpNum*, *n* As *mpNum*, *p* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

7.3.4.1 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 ξ a uniform random number between zero and one. In these cases

$$\begin{aligned} p = 1 \Rightarrow x &= 1 - \xi^{1/q} \\ q = 1 \Rightarrow x &= \xi^{1/p} \end{aligned}$$

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).

7.4 Chi-Square Distribution

7.4.1 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 χ^2 -distribution with n degrees of freedom.

7.4.2 Density and CDF

Function **CDist**(*x* As *mpNum*, *n* As *mpNum*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **CDist** returns pdf, CDF and related information for the central χ^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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.4.2.1 and 7.4.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**CHIDIST**(*x* As *mpReal*, *deg_freedom* As *mpReal*, *Tails* As *Integer*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.CHIDIST** returns the CDF and of the central χ^2 -distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

deg_freedom: An integer greater 0, indicating the degrees of freedom

Tails: Specifies the number of distribution tails to return. If tails = 1, TDIST returns the one-tailed distribution. If tails = 2, TDIST returns the two-tailed distribution.

WorksheetFunction.**CHISQDIST**(*x* As *mpReal*, *deg_freedom* As *mpReal*, *Tails* As *Integer*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.CHISQDIST** returns the CDF and of the central χ^2 -distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

deg_freedom: An integer greater 0, indicating the degrees of freedom

Tails: Specifies the number of distribution tails to return. If tails = 1, TDIST returns the one-tailed distribution. If tails = 2, TDIST returns the two-tailed distribution.

WorksheetFunction.**CHISQ.DIST**(*x* As mpReal, *deg_freedom* As mpReal, *Cumulative* As Boolean) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.CHISQ.DIST returns the CDF and of the central χ^2 -distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

deg_freedom: An integer greater 0, indicating the degrees of freedom

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.**CHISQ.DIST.RT**(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.CHISQ.DIST.RT returns the complement of the CDF and of the central χ^2 -distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

WorksheetFunction.**CHISQ.DIST.2T**(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.CHISQ.DIST.2T returns the two-sided CDF of the central χ^2 -distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

7.4.2.1 Density

The density of a central chi-square variable with *n* degrees of freedom is given by

$$f_{\chi^2}(n, x) = \frac{1}{2^{n/2}\Gamma(n/2)}x^{(n-2)/2}e^{-x/2}. \quad (7.4.1)$$

7.4.2.2 CDF: General formulas

The cdf of a central chi-square variable with *n* degrees of freedom is given by

$$\Pr [\chi^2 \leq x] = F_{\chi^2}(n, x) = \int_0^x f_{\chi^2}(n, t)dt \quad (7.4.2)$$

7.4.2.3 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

$$1 - F_{\chi^2}(n, x) = f_{\chi^2}(n, x) \frac{1}{(1 + u_1/(v_1 + u_2/(v_2 + u_3/(v_3 + \dots))))} \quad (7.4.3)$$

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

$$F_{\chi^2}(n, x) = f_{\chi^2}(n, x) \frac{m}{b} \frac{1}{(1 + u_1/(v_1 + u_2/(v_2 + u_3/(v_3 + \dots))))} \quad (7.4.4)$$

where $M = \frac{1}{2}x$, $b = \frac{1}{2}n$, $u_1 = -M$, $u_{2j} = jM$, $u_{2j+1} = -(b + j)M$, $v_j = b + j$, $j = 1, 2, \dots$

7.4.2.4 CDF (central case): Finite sum

The cdf can be expressed as a finite sum if n is an integer:

$$F_{\chi^2}(n, x) = 1 + 2\Phi(-\sqrt{x}) + 2\phi(\sqrt{x}) \sum_{r=1}^{(n-1)/2} \frac{\sqrt{x}^{2r-1}}{1 \cdot 3 \cdot 5 \dots (2r-1)}, \quad \text{for } n \text{ odd,} \quad (7.4.5)$$

$$F_{\chi^2}(n, x) = e^{-x/2} \left(1 + \sum_{r=1}^{(n-2)/2} \frac{x^r}{2 \cdot 4 \cdot 6 \dots (2r)} \right), \quad \text{for } n \text{ even,} \quad (7.4.6)$$

where $\phi(\cdot)$ denotes the pdf of the normal distribution (see section 7.11.2.1) and $\Phi(\cdot)$ denotes the cdf of the normal distribution (see section 7.11.2.2).

7.4.3 Quantiles

Function **CDistInv**(*Prob* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function **CDistInv** returns quantiles and related information for the the central χ^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 7.1.3.2 for the options for *Prob* and *Output*).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**CHIINV**(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.CHIINV** returns the two-tailed inverse of the central χ^2 -distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

WorksheetFunction.**CHISQ.INV**(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function `WorksheetFunction.CHISQ.INV` returns the left-tailed inverse of the central χ^2 -distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

`WorksheetFunction.CHISQ.INV.RT(x As mpReal, deg_freedom As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.CHISQ.INV.RT` returns the two-tailed inverse of the central χ^2 -distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

`WorksheetFunction.CHISQINV(x As mpReal, deg_freedom As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.CHISQINV` returns the two-tailed inverse of the central χ^2 -distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

7.4.3.1 Quantiles (central case): algorithms and formulas

Let z_α and $\chi_{n,\alpha}^2$ be the α -quantiles of the standard normal distribution and central chi-square distribution with n the degrees of freedom. For $n = 1$ and $n = 2$, the following closed form expressions can be used:

$$\chi_{1,\alpha}^2 = z_\alpha^2, \quad \chi_{2,\alpha}^2 = 2 \log(1 - \alpha) \quad (7.4.7)$$

At the extreme left tail of the distribution, for small x , the CDF of a χ^2 variable with n degrees of freedom can be approximated by the density of a χ^2 variable with $n + 2$ degrees of freedom:

$$F_{\chi^2}(n, x) \approx 2f_{\chi^2}(n + 2, x).$$

The density of a χ^2 variable with $n + 2$ degrees of freedom can be inverted in closed form using the Lambert W function, which leads to the following approximation:

$$\chi_{n,\alpha}^2 \approx f_{\chi^2}^{-1}(n + 2, \alpha) = -2W(t)/a, \quad \text{where} \quad (7.4.8)$$

$$a = \frac{1}{(n + 2)/2 - 1}, \quad k = \ln(\Gamma((n + 2)/2)), \quad d = a - \ln(1 - \alpha) + k, \quad t = -ae^{p+d}$$

This approximation is used for $|t| < 0.1$, and the Lambert W function is approximated as

$$W(x) \approx x - x^2 + \frac{3}{2}x^3 - \frac{8}{3}x^4 - \frac{125}{24}x^5.$$

Otherwise, the quantile is approximated by inverting a formula proposed by [Canal \(2005\)](#):

$$\chi_{n,\alpha}^2 \approx n \left(\frac{1}{2} + \frac{t}{2} - \frac{3}{2t} \right)^6, \quad \text{where} \quad (7.4.9)$$

$$t = \left(-5 + 2L + 2\sqrt{13 - 5L + L^2} \right)^{1/3}, \quad L = 6 \left(m + s \left(az_\alpha^2 + z_\alpha - a \right) \right)$$

$$m = \frac{5}{6} - \frac{1}{9n} - \frac{7}{648n^2} - \frac{25}{2187n^3}, \quad s^2 = \frac{1}{18n} + \frac{1}{162n^2} - \frac{37}{11664n^3}, \quad a = \frac{1}{162\sqrt{2n^3}}.$$

These approximations are then used as a starting point for a Newton iteration.

7.4.4 Properties

Function **CDistInfo**(*n* As *mpNum*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **CDistInfo** returns moments and related information for the central χ^2 -distribution

Parameters:

n: A real number greater 0, representing the degrees of freedom

Output: A string describing the output choices

See section 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.4.4.

7.4.4.1 Median (central case)

The median is given approximately by

$$k - \frac{2}{3} + \frac{4}{27k} - \frac{8}{729k^2}. \quad (7.4.10)$$

7.4.4.2 Moments and Cumulants (central case)

The cumulants are given by

$$\kappa_{r+1} = 2^r r! n \quad (7.4.11)$$

The k^{th} null-moment of the r^{th} root of a chi-square variable is given by:

$$E(X^{k/r}) = \frac{2^{k/r} \Gamma(n/2) + k/r}{\Gamma(n/2)} \quad (7.4.12)$$

where $\Gamma(\cdot)$ is the Gamma function (see section 5.8.6.)

The first 4 cumulants of cube root central χ^2 , $\chi^{2/3}$, are given by (Aty, 1954)

$$\kappa_1^*(n, 0) = 1 - \frac{2}{9n} + \frac{80}{3^7 n^3} + \frac{176}{3^9 n^4} + o(n^{-4}) \quad (7.4.13)$$

$$\kappa_2^*(n, 0) = \frac{2}{9n} - \frac{104}{3^7 n^3} - \frac{160}{3^8 n^4} + o(n^{-4}) \quad (7.4.14)$$

$$\kappa_3^*(n, 0) = -\frac{32}{3^6 n^3} - \frac{256}{3^8 n^4} + o(n^{-4}) \quad (7.4.15)$$

$$\kappa_4^*(n, 0) = -\frac{16}{3^6 n^3} - \frac{256}{3^8 n^4} + o(n^{-4}) \quad (7.4.16)$$

7.4.4.3 Recurrence Relations (central case)

The following recurrence relations hold for the pdf and CDF:

$$f_{\chi^2}(n+2, x) = \frac{x}{n} f_{\chi^2}(n, x) \quad (7.4.17)$$

$$F_{\chi^2}(n, x) - F_{\chi^2}(n+2, x) = 2f_{\chi^2}(n+2, x) \quad (7.4.18)$$

7.4.5 Random Numbers

Function **CDistRan**(*Size* As *mpNum*, *n* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **CDistRan** returns random numbers following a central χ^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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 7.4.5.

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.

7.4.6 Wishart Matrix

See [Gleser \(1976\)](#)

7.5 Exponential Distribution

These functions return PDF, CDF, and ICDF of the exponential distribution with location a , rate $\alpha > 0$, and the support interval $(a, +\infty)$:

7.5.1 Density and CDF

Function **ExponentialDist**(*x* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **ExponentialDist** 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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.2.2.1 and 7.2.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**EXPONDIST**(*x* As *mpNum*, *lambda* As *mpNum*, *Cumulative* As *Boolean*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.EXPONDIST** returns pdf, CDF, and related information of the central Binomial-distribution

Parameters:

x: The value of the distribution.

lambda: The parameter of the distribution.

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.**EXPON.DIST**(*x* As *mpNum*, *lambda* As *mpNum*, *Cumulative* As *Boolean*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.EXPON.DIST** returns the CDF and pdf of the central Binomial-distribution

Parameters:

x: The value of the distribution.

lambda: The parameter of the distribution.

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

7.5.1.1 Density

$$f(x) = \alpha \exp(-\alpha(x - a)) \quad (7.5.1)$$

7.5.1.2 CDF

$$F(x) = 1 - \exp(-\alpha(x - a)) = \text{expm1}(-\alpha(x - a)) \quad (7.5.2)$$

7.5.2 Quantiles

Function **ExponentialDistInv**(*Prob* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **ExponentialDistInv** 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 7.1.3.2 for the options for *Prob* and *Output*).

$$F^{-1}(y) = a - \text{ln1p}(-y)/\alpha \quad (7.5.3)$$

7.5.3 Properties

Function **ExponentialDistInfo**(*lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **ExponentialDistInfo** 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.5.3.1 Moments and cumulants

The mean or expected value of an exponentially distributed random variable *X* with rate parameter λ is given by

$$E[X] = \frac{1}{\lambda} \quad (7.5.4)$$

The variance of *X* is given by

$$E[X^2] = \frac{1}{\lambda^2} \quad (7.5.5)$$

so the standard deviation is equal to the mean.

The moments of X , for $n = 1, 2, \dots$, are given by

$$E[X^n] = \frac{n!}{\lambda^n} \quad (7.5.6)$$

7.5.4 Random Numbers

Function **ExponentialDistRandom**(*Size* As *mpNum*, *lambda* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **ExponentialDistRandom** 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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 7.6.5.

7.5.4.1 Random Numbers: algorithms and formulas

Random numbers can be generated using the inversion formula.

7.6 Fisher's F-Distribution

7.6.1 Definition

If X_1 and X_2 are independent random variables following χ^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.

7.6.2 Density and CDF

Function **FDist**(*x* As mpNum, *m* As mpNum, *n* As mpNum, **Output** As String) As mpNumList

NOT YET IMPLEMENTED

The function FDist 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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.6.2.1 and 7.6.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**FDIST**(*x* As mpReal, *m* As mpNum, *n* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.FDIST returns the CDF and of the central *F*-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

m: A real number greater 0, representing the numerator degrees of freedom

n: A real number greater 0, representing the denominator degrees of freedom

WorksheetFunction.**F.DIST**(*x* As mpReal, *m* As mpNum, *n* As mpNum, **Cumulative** As Boolean) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.F.DIST returns the CDF and of the central *F*-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

m: A real number greater 0, representing the numerator degrees of freedom

n: A real number greater 0, representing the denominator degrees of freedom

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, F.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.F.DIST.RT(*x* As mpReal, *m* As mpNum, *n* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.F.DIST.RT returns the complement of the CDF and of 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

7.6.2.1 Density

The density of a variable following a central F-distribution with *m* and *n* degrees of freedom is given by

$$f_F(m, n, x) = \frac{m^{m/2} n^{n/2}}{B(m/2, n/2)} x^{(m-2)/2} (n + mx)^{-(m+n)/2} \quad (7.6.1)$$

7.6.2.2 CDF: General formulas

The cdf of a variable following a central F-distribution with *m* and *n* degrees of freedom is given by

$$\Pr[X \leq x] = F_F(m, n, x) = \int_0^x f(m, n, t) dt \quad (7.6.2)$$

7.6.2.3 CDF (central): finite series

The cdf can be expressed as a finite sum if *m* is an integer, and *n* is a positive real number:

$$1 - F_F(m, n, x) = a_m + b_m(c_1 + c_3 + \dots + c_{m-2}), \quad \text{for } m \text{ odd,} \quad (7.6.3)$$

$$\text{where } a_m = 2T(n, -z_m); b_m = 2t(n, z_m) \cdot z_m; z_m = \sqrt{mx}; \quad (7.6.4)$$

$$1 - F_F(m, n, x) = d_m(c_0 + c_2 + \dots + c_{m-2}), \quad \text{for } m \text{ even,} \quad (7.6.5)$$

$$\text{where } d_m = (1 - u_m)^{n/2} \quad (7.6.6)$$

$$\text{and } u_m = mx/(mx + n), \quad c_0 = c_1 = 1, \quad c_k = c_{k-2}u_m \cdot (n + k - 2)/k \quad (7.6.7)$$

7.6.3 Quantiles

Function FDistInv(*Prob* As mpNum, *m* As mpNum, *n* As mpNum) As mpNumList

NOT YET IMPLEMENTED

The function FDistInv 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 7.1.3.2 for the options for *Prob* and *Output*).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.FINV(*x* As mpReal, *m* As mpNum, *n* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.FINV returns the two-tailed inverse of the central *t*-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

WorksheetFunction.F.INV(*x* As mpReal, *m* As mpNum, *n* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.F.INV returns the left-tailed inverse of the central *t*-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

WorksheetFunction.F.INV.RT(*x* As mpReal, *m* As mpNum, *n* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.F.INV.RT returns the right-tailed inverse of the central *t*-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

7.6.3.1 Quantiles (central case): algorithms and formulas

Let *p* be a right tail probability, z_α the α -quantile of the standard normal distribution, and *n* the degrees of freedom. Depending on *n* and *p*, one proceeds as follows:

$$F_{1,n,\alpha} = t_{n,\alpha}^2 \tag{7.6.8}$$

$$F_{2,n,\alpha} = \frac{n(1-x)}{2x}, \quad \text{where } x = \frac{2 \log(1-\alpha)}{n} \tag{7.6.9}$$

7.6.3.2 Cornish-Fisher expansion

Otherwise, the Cornish-Fisher expansion of $\frac{1}{2} \log(F_{m,n})$ is used, as given by [Sahai & Thompson \(1974\)](#), including cumulants through order eight.

$$F_{m,n,\alpha} \approx e^{2w}, \quad \text{where} \tag{7.6.10}$$

$$s = \frac{1}{m} + \frac{1}{n}, \quad d = \frac{1}{m} - \frac{1}{n}, \quad r = \sqrt{s/2}, \quad \text{and} \quad (7.6.11)$$

$$\begin{aligned} w = & zr - \frac{d(z^2 + 2)}{6} + \frac{rs(z^3 + 3z)}{24} + \frac{rd^2(z^3 + 11z)}{72s} - \frac{ds(z^4 + 9z^2 + 8)}{120} + \frac{d^3(3z^4 + 7z^2 - 16)}{3240s} \\ & + \frac{rs^2(z^5 + 20z^3 + 15z)}{1920} + \frac{rd^2(z^5 + 44z^3 + 183z)}{2880} + \frac{d^4(9z^5 + 284z^3 - 1513z)}{155520s^2} \\ & + \frac{ds^2(4z^6 - 25z^4 - 177z^2 + 192)}{20160} + \frac{d^3(4z^6 + 101z^4 + 177z^2 - 480)}{90720} \\ & + \frac{d^5(12z^6 + 513z^4 + 841z^2 - 2560)}{1632960s^2} - \frac{rs^3(z^7 + 7z^5 + 7z^3 + 105z)}{21504} \\ & + \frac{rd^2s(801z^7 + 10511z^5 + 30151z^3 + 62241z)}{4838400} - \frac{rd^4(477z^7 + 4507z^5 - 82933z^3 - 264363z)}{43545600s} \\ & + \frac{rd^6(3753z^7 + 55383z^5 - 368897z^3 - 1213927z)}{1175731200s^3} \end{aligned}$$

7.6.3.3 Box-Davis Expansion

From [Box \(1949\)](#) and [Davis \(1971\)](#) we derive the following approximation: let u_α be the α percentage point of a chi-square-distribution with m degrees of freedom

$$F_{m,n,\alpha} \approx \frac{n}{m} \frac{1 - e^{-X}}{e^{-X}}, \quad \text{where} \quad (7.6.12)$$

$$\mu = n + \frac{1}{2}m - 1, \quad m_1 = m, \quad m_k = m_{k-1}(m + 2k - 2), \quad P_1 = u/m, \quad P_k = P_{k-1} + u^k/m_k$$

$$P_{22} = \frac{-8u^4(m+3)}{m_2m_4} + \frac{8u^3}{m_2m_3} + \frac{6u^2}{mm_2} + \frac{2u}{m^2}$$

$$P_{42} = \frac{-16u^6(m+5)}{m_2m_6} - \frac{4u^5(m-4)}{m_2m_5} + \frac{2u^4(3m+14)}{m_2m_4} + \frac{2u^3(3m+10)}{m_2m_3} + \frac{6u^2}{mm_2} + \frac{2u}{m^2}$$

$$\begin{aligned} P_{222} = & \frac{32u^6(7m^2 + 62m + 120)}{m_2^2m_6} - \frac{32u^5(2m^2 + 37m + 96)}{m_2^2m_5} - \frac{8u^4(23m^2 + 124m + 132)}{m_2^2m_4} \\ & - \frac{8u^3(m-10)}{m_1m_2m_3} + \frac{28u^2}{m^2m_2} + \frac{4u}{m^3} \end{aligned}$$

$$\omega_2 = \frac{m(m^2 - 4)}{48\mu^2}, \quad \omega_4 = \frac{m(3m^4 - 40m^2 + 112)}{1920\mu^4}, \quad \omega_6 = \frac{m(3m^6 - 84m^3 + 784m^2 - 1984)}{16128\mu^6},$$

$$s_2 = \omega_2 P_2, \quad s_4 = \omega_4 P_4 + \frac{1}{2}\omega_2^2 P_{22}, \quad s_6 = \omega_6 P_6 + \omega_4 \omega_2 P_{42} + \omega_2^3 P_{222}$$

$$X = (u + 2(s_2 + s_4 + s_6))/\mu$$

7.6.3.4 Confidence Interval

See [Guenther \(1977\)](#)

7.6.4 Properties

Function **FDistInfo**(*m* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function FDistInfo 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.6.4.

7.6.4.1 Recurrence relations (central case)

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. Then the following recurrence relations hold (Chattamvelli & Jones, 1995)

$$n [G_{m,n+2}(y) - G_{m-2,n+2}(y)] = -2g_{m,n}(y) \quad (7.6.13)$$

$$m(1+y)g_{m+2,n}(y) + y(m+n)g_{m,n}(y). \quad (7.6.14)$$

$$n(1+y)g_{m,n+2}(y) = (m+n)g_{m,n}(y). \quad (7.6.15)$$

$$mg_{m+2,n-2}(y) = (n-2)yg_{m,n}(y). \quad (7.6.16)$$

From equations 7.6.13 to 7.6.16 we obtain

$$\begin{aligned} [(m+2)(1+y)]G_{m+4,n}(y) &= [(m+2)(1+y) + y(m+n)]G_{m+2,n}(y) \\ &\quad - y(m+n)G_{m,n}(y) \end{aligned} \quad (7.6.17)$$

$$n(1+y)[G_{m,n+2}(y) - G_{m+2,n+2}(y)] = (m+n)[G_{m,n}(y) - G_{m+2,n}(y)] \quad (7.6.18)$$

$$(m+2)[G_{m+2,n}(y) - G_{m+4,n-2}(y)] = (n-2)[G_{m,n}(y) - G_{m+2,n}(y)] \quad (7.6.19)$$

7.6.4.2 Relations to other distributions (central case)

$$F_F(m, n; x) = 1 - F_F\left(n, m; \frac{1}{x}\right) \quad (7.6.20)$$

$$F_F(m, n; x) = F_B\left(n/2, m/2; \frac{mx}{mx+n}\right) \quad (7.6.21)$$

where $F_B(\cdot)$ denotes the cdf of the central Beta-distribution (see section 7.2.2.2).

7.6.5 Random Numbers

Function **FDistRan**(*Size* As mpNum, *m* As mpNum, *n* As mpNum, *Generator* As String, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function FDistRan 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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 7.6.5.

7.6.5.1 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 ...).

7.7 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.

7.7.1 Density and CDF

Function **GammaDist**(*x* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNum-List

NOT YET IMPLEMENTED

The function **GammaDist** 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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.2.2.1 and 7.2.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**GAMMADIST**(*x* As mpReal, *a* As mpNum, *b* As mpNum, *Cumulative* As Boolean) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.GAMMADIST** returns the CDF and of the central Gamma-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

a: A real number greater 0, a parameter to the distribution

b: A real number greater 0, a parameter to the distribution

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, GAMMA.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function.

WorksheetFunction.**GAMMA.DIST**(*x* As mpReal, *a* As mpNum, *b* As mpNum, *Cumulative* As Boolean) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.GAMMA.DIST** returns the CDF and of the central Gamma-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

- a*: A real number greater 0, a parameter to the distribution
b: A real number greater 0, a parameter to the distribution

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, GAMMA.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function.

7.7.1.1 Density

$$f(x; a, b) = \frac{x^{a-1} e^{-x/b}}{\Gamma(a) b^a} \quad (7.7.1)$$

7.7.1.2 CDF: General formulas

$$F(x; a, b) = P(a, x/b) = igammap(a, x/b) \quad (7.7.2)$$

7.7.2 Quantiles

Function **GammaDistInv**(*Prob* As mpNum, *m* As mpNum, *n* As mpNum) As mpNumList

NOT YET IMPLEMENTED

The function GammaDistInv 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 7.1.3.2 for the options for *Prob* and *Output*).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**GAMMAINV**(*Prob* As mpReal, *a* As mpNum, *b* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.GAMMAINV returns the two-tailed inverse of the central Gamma-distribution

Parameters:

Prob: 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

WorksheetFunction.**GAMMA.INV**(*Prob* As mpReal, *a* As mpNum, *b* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.GAMMA.INV returns the left-tailed inverse of the central Gamma-distribution

Parameters:

Prob: 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

$$F^{-1}(y) = b \cdot igammaInv(a, y) \quad (7.7.3)$$

7.7.3 Properties

Function **GammaDistInfo**(*a* As *mpNum*, *b* As *mpNum*) As *mpNumList*

NOT YET IMPLEMENTED

The function **GammaDistInfo** 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.7.3.1 Moments

The algebraic moments are given by (Wolfram)

$$\mu'_r = \frac{b^r \Gamma(a + r)}{\Gamma(a)} \quad (7.7.4)$$

7.7.4 Random Numbers

Function **GammaDistRandom**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Generator** As String, **Output** As String) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, **Generator** and **Output**. Algorithms and formulas are given below.

7.7.4.1 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 ξ_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.

7.7.4.2 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 ξ 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 $r = r/a$.

7.8 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.* \(1978\)](#)

7.8.1 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.

7.8.2 Density and CDF

Function **HypergeometricDist**(*x As mpNum, n As mpNum, M As mpNum, N As mpNum, Output As String*) As mpNumList

NOT YET IMPLEMENTED

The function **HypergeometricDist** 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 [7.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [7.2.2.1](#) and [7.2.2.2](#).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**HYPGEOMDIST**(*x As mpNum, n As mpNum, M As mpNum, N As mpNum, Cumulative As Boolean*) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.HYPGEOMDIST** returns pdf, CDF, and related information of 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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.HYPGEOM.DIST(*x* As mpNum, *n* As mpNum, *M* As mpNum, *N* As mpNum, *Cumulative* As Boolean) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.HYPGEOM.DIST returns the CDF and pdf of 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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

7.8.2.1 Density

$$f(k) = \frac{\binom{n_1}{k} \binom{n_2}{n-k}}{\binom{n_1+n_2}{n}}, \quad (n, n_1, n_2 \geq 0; n \leq n_1 + n_2). \quad (7.8.1)$$

f(k) is computed with the R trick [39], which replaces the binomial coefficients by binomial PMFs with $p = n/(n_1 + n_2)$.

7.8.2.2 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:

$$f(k+1) = \frac{(n_1 - k)(n - k)}{(k+1)(n_2 - n + k + 1)} f(k) \quad (7.8.2)$$

$$f(k-1) = \frac{k(n_2 - n + k)}{(n_1 - k + 1)(n - k + 1)} f(k) \quad (7.8.3)$$

7.8.3 Quantiles

Function HypergeometricDistInv(*Prob* As mpNum, *n* As mpNum, *M* As mpNum, *N* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

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 7.1.3.2 for the options for *Prob* and *Output*).

7.8.4 Properties

Function **HypergeometricDistInfo**(*n* As mpNum, *M* As mpNum, *N* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.8.4.1 Moments

$$\mu_1 = nP \quad (7.8.4)$$

$$\mu_2 = nPQ \frac{N-n}{N-1} \quad (7.8.5)$$

$$\mu_3 = nPQ(Q-P) \frac{(N-n)(N-2n)}{(N-1)(N-2)} \quad (7.8.6)$$

$$\kappa_4 = \frac{6nP^2Q^2(N-n)}{N-1} \frac{n(N-n)(5N-6)-N(N-1)}{(N-2)(N-3)} \quad (7.8.7)$$

7.8.5 Random Numbers

Function **HypergeometricDistRandom**(*Size* As mpNum, *n* As mpNum, *M* As mpNum, *N* As mpNum, *Generator* As String, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

7.9 Lognormal Distribution

7.9.1 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 μ and σ are, respectively, the mean and standard deviation of the variable's natural logarithm (by definition, the variable's logarithm is normally distributed), which means

$$X = e^{\mu + \sigma Z} \quad (7.9.1)$$

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, μ and σ 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

$$\mu = \ln \left(\frac{m^2}{\sqrt{v + m^2}} \right), \quad \sigma = \sqrt{\ln \left(1 + \frac{v}{m^2} \right)} \quad (7.9.2)$$

7.9.2 Density and CDF

Function **LogNormalDist**(*x* As *mpNum*, **mean** As *mpNum*, **stdev** As *mpNum*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function LogNormalDist 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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.2.2.1 and 7.2.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**LOGNORMDIST**(*x* As mpReal, **mean** As mpNum, **stdev** As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.LOGNORMDIST returns the CDF and of the Lognormal-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

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

WorksheetFunction.**LOGNORM.DIST**(*x* As mpReal, **mean** As mpNum, **stdev** As mpNum, **Cumulative** As Boolean) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.LOGNORM.DIST returns the CDF and of the Lognormal-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

7.9.2.1 Density

$$f(x) = \frac{1}{bx\sqrt{2\pi}} \exp\left(-\frac{(\ln(x) - a)^2}{2b^2}\right) \quad (7.9.3)$$

7.9.2.2 CDF

$$F(x) = \frac{1}{2} \left(1 + \operatorname{erf}\left(\frac{\ln(x) - a}{b\sqrt{2}}\right) \right) \quad (7.9.4)$$

7.9.3 Quantiles

Function **LognormalDistInv**(*Prob* As mpNum, **mean** As mpNum, **stdev** As mpNum, **Output** As String) As mpNumList

NOT YET IMPLEMENTED

The function LognormalDistInv 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 7.1.3.2 for the options for *Prob* and *Output*).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**LOGINV**(*Prob* As mpReal, *mean* As mpNum, *stdev* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.LOGINV returns the two-tailed inverse of the Lognormal-distribution

Parameters:

Prob: 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

WorksheetFunction.**LOGNORM.INV**(*Prob* As mpReal, *mean* As mpNum, *stdev* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.LOGNORM.INV returns the left-tailed inverse of the Lognormal-distribution

Parameters:

Prob: 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

7.9.3.1 Quantiles: algorithms and formulas

$$F^{-1}(y) = \exp(a + b \cdot \text{normstdinv}(y)) \quad (7.9.5)$$

7.9.4 Properties

Function **LognormalDistInfo**(*mean* As mpNum, *stdev* As mpNum, *Output* As String) As mp-NumList

NOT YET IMPLEMENTED

The function LognormalDistInfo 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.9.4.1 Moments: algorithms and formulas

Algebraic moments of the log-normal distribution are given by

$$\mu'_k = e^{k\mu + k^2\sigma^2/2} \quad (7.9.6)$$

7.9.5 Random Numbers

Function **LognormalRandom**(*Size* As *mpNum*, *mean* As *mpNum*, *stdev* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function LognormalRandom 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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 7.6.5.

7.9.5.1 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 μ and standard deviation σ and construct $r = e^u$.

7.10 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

7.10.1 Density and CDF

Function **NegativeBinomialDist**(*x As mpNum, r As mpNum, p As mpNum, Output As String*)
As mpNumList

NOT YET IMPLEMENTED

The function NegativeBinomialDist 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 [7.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [7.3.1.1](#) and [7.3.1.2](#).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**NEGBINOMDIST**(*x As mpNum, r As mpNum, p As mpNum, Cumulative As Boolean*) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NEGBINOMDIST returns pdf, CDF, and related information of 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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.**NEGBINOM.DIST**(*x As mpNum, r As mpNum, p As mpNum, Cumulative As Boolean*) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NEGBINOM.DIST returns the CDF and pdf of 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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

7.10.1.1 Density

$$f_{\text{NegBin}}(r, k; p) = \frac{\Gamma(k+r)}{k!\Gamma(r)} p^r (1-p)^k = \frac{p}{r+k} f_{\text{Beta}}(r, k+1, p) \quad (7.10.1)$$

7.10.1.2 CDF

$$F_{\text{NegBin}}(r, k; p) = I_{1-p}(r, k+1) = \text{ibeta}(r, k+1, 1-p) \quad (7.10.2)$$

7.10.2 Quantiles

Function **NegativeBinomialDistInv**(*Prob* As mpNum, *r* As mpNum, *p* As mpNum, **Output** As String) As mpNumList

NOT YET IMPLEMENTED

The function NegativeBinomialDistInv 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 7.1.3.2 for the options for *Prob* and *Output*).

7.10.3 Properties

Function **NegativeBinomialDistInfo**(*r* As mpNum, *p* As mpNum, **Output** As String) As mpNumList

NOT YET IMPLEMENTED

The function NegativeBinomialDistInfo 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.10.3.1 Moments: algorithms and formulas

$$\mu_1 = np \quad (7.10.3)$$

$$\mu_2 = np(1 - p) = npq \quad (7.10.4)$$

$$\mu_3 = npq(q + p) \quad (7.10.5)$$

$$\mu_4 = npq(3npq + 6pq + 1) \quad (7.10.6)$$

7.10.3.2 Recurrence relations

The following recurrence relations hold:

$$f_{\text{NegBin}}(r, k + 1; p) = \frac{(r + k)(1 - p)}{k + 1} f_{\text{NegBin}}(r, k; p) \quad (7.10.7)$$

$$f_{\text{NegBin}}(r, k - 1; p) = \frac{k}{(r + k - 1)(1 - p)} f_{\text{NegBin}}(r, k; p) \quad (7.10.8)$$

7.10.4 Random Numbers

Function **NegativeBinomialDistRandom**(*Size* As *mpNum*, *r* As *mpNum*, *p* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given below.

7.10.4.1 Random Numbers: algorithms and formulas

Random numbers from a negative binomial distribution can be obtained using the algorithms outline for the beta distribution.

7.11 Normal Distribution

7.11.1 Definition

A random variable is said to follow a normal distribution with mean μ and variance σ^2 , if its pdf is given by 7.11.1. It is said to follow a standardized normal distribution if its pdf is given by 7.11.2.

7.11.2 Density and CDF

Function **NDist**(*x* As *mpNum*, **mean** As *mpNum*, **stdev** As *mpNum*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

The function **NDist** 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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.2.2.1 and 7.2.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**NORMDIST**(*x* As *mpReal*, **mean** As *mpNum*, **stdev** As *mpNum*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.NORMDIST** returns the CDF and of the Lognormal-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

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

WorksheetFunction.**NORM.DIST**(*x* As *mpReal*, **mean** As *mpNum*, **stdev** As *mpNum*, **Cumulative** As Boolean) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.NORM.DIST** returns the CDF and of the Lognormal-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

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

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.**NORMSDIST**(*x* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NORMSDIST returns the CDF and of the standard normal distribution

Parameter:

x: A real number. The numeric value at which to evaluate the distribution

WorksheetFunction.**NORM.S.DIST**(*x* As mpReal, *Cumulative* As Boolean) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NORM.S.DIST returns the CDF and of the standard normal distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, NORM.S.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.**GAUSS**(*x* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.GAUSS returns the CDF of the standard normal distribution

Parameter:

x: A real number. The numeric value at which to evaluate the distribution

WorksheetFunction.**PHI**(*x* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.PHI returns the pdf of the standard normal distribution

Parameter:

x: A real number. The numeric value at which to evaluate the distribution

7.11.2.1 Density

This function returns the pdf of the normal distribution with mean μ and variance σ^2 , which is given by

$$f_N(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \quad (7.11.1)$$

The pdf of the standardized normal distribution with mean 0 and variance 1 is given by

$$\phi(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2}, \quad (7.11.2)$$

These two functions are related by

$$f_N(x; \mu, \sigma^2) = \frac{1}{\sigma} \phi\left(\frac{x - \mu}{\sigma}\right), \text{ and } \phi(u) = \sigma f_N(\mu + \sigma u) \quad (7.11.3)$$

7.11.2.2 CDF

This functions returns the cdf of the normal distribution with mean μ and variance σ^2 , which is given by

$$F_N(x; \mu, \sigma^2) = \int_{-\infty}^x f_N(v) dv \quad (7.11.4)$$

The cdf of the standardized normal distribution with mean 0 and variance 1 is given by

$$\Phi(u) = \int_{-\infty}^u \phi(w) dw \quad (7.11.5)$$

These two functions are related by

$$F_N(x; \mu, \sigma^2) = \Phi\left(\frac{x - \mu}{\sigma}\right), \text{ and } \Phi(u) = F_N(\mu + \sigma u) \quad (7.11.6)$$

7.11.3 Quantiles

These functions return the quantile of the normal distribution with mean μ and variance σ^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*

NOT YET IMPLEMENTED

The function **NDistInv** 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 7.1.3.2 for the options for *Prob* and *Output*).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**NORMINV**(*Prob* As *mpReal*, *mean* As *mpNum*, *stdev* As *mpNum*) As *mpReal*

NOT YET IMPLEMENTED

The function **WorksheetFunction.NORMINV** returns the two-tailed inverse of the normal distribution

Parameters:

Prob: 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

WorksheetFunction.**NORM.INV**(*Prob* As mpReal, *mean* As mpNum, *stdev* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NORM.INV returns the left-tailed inverse of the normal distribution

Parameters:

Prob: 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

WorksheetFunction.**NORMSINV**(*Prob* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NORMSINV returns the two-tailed inverse of the standardized normal distribution

Parameter:

Prob: A real number

WorksheetFunction.**NORM.S.INV**(*Prob* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NORM.S.INV returns the left-tailed inverse of the standardized normal distribution

Parameter:

Prob: A real number

7.11.3.1 Quantiles: algorithms and formulas

$$F^{-1}(y) = \exp(a + b \cdot \text{normstdinv}(y)) \quad (7.11.7)$$

7.11.4 Properties

Function **NormalDistInfo**(*mean* As mpNum, *stdev* As mpNum, *Output* As String) As mpNum-List

NOT YET IMPLEMENTED

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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.11.4.1 Moments: algorithms and formulas

$$\begin{aligned}\kappa_1 &= \mu \\ \kappa_2 &= \sigma^2 \\ \kappa_r &= 0 \text{ for } r \geq 3.\end{aligned}$$

7.11.4.2 Differential Equation

Let $Z^{(m)}$ denote the m^{th} derivative of $Z(x)$. Then (Abramowitz & Stegun., 1970)

$$Z^{(1)} = -xZ(x) \quad (7.11.8)$$

$$Z^{(m+2)} + xZ^{(m+1)} + (m+1)Z^{(m)} = 0 \quad (7.11.9)$$

7.11.5 Random Numbers

Function **NormalRandom**(*Size* As *mpNum*, *mean* As *mpNum*, *stdev* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 7.6.5.

7.11.5.1 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 $\Phi^{-1}(\alpha)$.

7.12 Poisson Distribution

7.12.1 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 $\mu \geq 0$.

7.12.2 Density and CDF

Function **PoissonDist**(*x* As *mpNum*, *lambda* As *mpNum*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.1 for the options for *Output*. Algorithms and formulas are given in sections 7.4.2.1 and 7.4.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.POISSON(*x* As *mpReal*, *deg_freedom* As *mpReal*, *Tails* As *Integer*) As *mpReal*

NOT YET IMPLEMENTED

The function WorksheetFunction.POISSON returns the CDF and of the Poisson distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

deg_freedom: An integer greater 0, indicating the degrees of freedom

Tails: Specifies the number of distribution tails to return. If tails = 1, TDIST returns the one-tailed distribution. If tails = 2, TDIST returns the two-tailed distribution.

WorksheetFunction.POISSON.DIST(*x* As *mpReal*, *deg_freedom* As *mpReal*, *Cumulative* As *Boolean*) As *mpReal*

NOT YET IMPLEMENTED

The function WorksheetFunction.POISSON.DIST returns the CDF and of the Poisson distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

deg_freedom: An integer greater 0, indicating the degrees of freedom

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, POISSON.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

7.12.2.1 Density

$$f(k) = \frac{\mu^k}{k!} e^{-\mu} = \text{fcIgprefix}(1 + k, \mu) \quad (7.12.1)$$

7.12.2.2 CDF

$$F(k) = e^{-\mu} \sum_{i=0}^k \frac{\mu^i}{i!} = \text{igammaq}(1 + k, \mu) \quad (7.12.2)$$

7.12.3 Quantiles

Function **PoissonDistInv**(*Prob* As mpNum, *lambda* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

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 7.1.3.2 for the options for *Prob* and *Output*). Algorithms and formulas are given in section 7.12.3.1.

7.12.3.1 Quantiles: algorithms and formulas

The algorithms follow the one for the chisquare distribution.

7.12.4 Properties

Function **PoissonDistInfo**(*lambda* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.4.4.

7.12.4.1 Moments and Cumulants

The momemnts and cumulants are given by

$$\kappa_r = \lambda \quad (7.12.3)$$

$$\mu_1 = \mu_2 = \mu_3 = \lambda \quad (7.12.4)$$

$$\mu_4 = 3\lambda^2 + \lambda \quad (7.12.5)$$

7.12.5 Random Numbers

Function **PoissonDistRan**(*Size* As *mpNum*, *lambda* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 7.4.5.

7.13 Student's t-Distribution

7.13.1 Definition

If X is a random variable following a normal distribution with mean zero and variance unity and χ^2 is a random variable following an independent χ^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

7.13.2 Density and CDF

Function **TDist**(*x* As mpNum, *n* As mpNum, **Output** As String) As mpNumList

NOT YET IMPLEMENTED

The function **TDist** 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 7.13.1 for the options for *Output*. Algorithms and formulas are given in sections 7.13.2.1 and 7.13.2.2.

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**TDIST**(*x* As mpReal, **deg_freedom** As mpReal, **Tails** As Integer) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.TDIST** returns the CDF and of the central *t*-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

deg_freedom: An integer greater 0, indicating the degrees of freedom

Tails: Specifies the number of distribution tails to return. If tails = 1, TDIST returns the one-tailed distribution. If tails = 2, TDIST returns the two-tailed distribution.

WorksheetFunction.**T.DIST**(*x* As mpReal, **deg_freedom** As mpReal, **Cumulative** As Boolean) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.T.DIST** returns the CDF and of the central *t*-distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

deg_freedom: An integer greater 0, indicating the degrees of freedom

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

WorksheetFunction.T.DIST.RT(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.T.DIST.RT returns the complement of the CDF and of the central *t*-distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

WorksheetFunction.T.DIST.2T(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.T.DIST.2T returns the two-sided CDF of the central *t*-distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

7.13.2.1 Density

The density of a variable following a central Student's *t*-distribution with *n* degrees of freedom is given by

$$f_t(n, x) = \frac{\Gamma((n+1)/2)}{\sqrt{n\pi}\Gamma(n/2)} \left(\frac{n}{n+x^2}\right)^{(n+1)/2} \quad (7.13.1)$$

where $\Gamma(\cdot)$ denotes the Gamma function (see section 5.8.6.)

7.13.2.2 CDF: General formulas

The cdf of a variable following a central *t*-distribution with *n* degrees of freedom is defined as

$$\Pr[X \leq x] = F_t(n, x) = \int_0^x f_t(n, t) dt \quad (7.13.2)$$

The cdf of the central *t*-distribution is calculated for any positive degrees of freedom *n* using the relationships

$$2F_t(n, x) = F_F(1, n; x^2), \quad x \leq 0 \quad (7.13.3)$$

$$F_t(n, x) - F_t(n, -x) = F_F(1, n; x^2), \quad x \geq 0 \quad (7.13.4)$$

$$F_t(n, x) = 1 - F_t(n, -x) \quad (7.13.5)$$

where $F_F(1, n; x^2)$ denotes the cdf of the central *F*-distribution with 1 and *n* of freedom (see section 7.6.2.2).

7.13.2.3 CDF (central): Finite sum

The cdf can be expressed as a finite sum if *n* is an integer:

$$F_t(n, x) = \frac{1}{2} + z_n + (c_1 + c_3 + \cdots + c_{n-2}), \quad \text{for } n \text{ odd}, \quad (7.13.6)$$

$$\text{where } z_n = \frac{1}{\pi} \arctan\left(\frac{x}{\sqrt{n}}\right); a_n = \frac{1}{\sqrt{n\pi}}; b_n = \frac{n}{n+x^2}; c_1 = x a_n b_n; c_k = c_{k-2} b_n (1 - 1/k) \quad (7.13.7)$$

$$F_t(n, x) = \frac{1}{2} + (c_0 + c_2 + \cdots + c_{n-2}), \quad \text{for } n \text{ even,} \quad (7.13.8)$$

$$\text{where } d_n = \frac{1}{2\sqrt{n+x^2}}; b_n = \frac{n}{n+x^2}; c_0 = xd_n; c_k = c_{k-2}b_n(1-1/k) \quad (7.13.9)$$

7.13.3 Quantiles

Function **TDistInv**(*Prob* As mpNum, *n* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function **TDistInv** 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 7.1.3.2 for the options for *Prob* and *Output*).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**TINV**(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.TINV** returns the two-tailed inverse of the central *t*-distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

WorksheetFunction.**T.INV**(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.T.INV** returns the left-tailed inverse of the central *t*-distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

WorksheetFunction.**T.INV.2T**(*x* As mpReal, *deg_freedom* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.T.INV.2T** returns the two-tailed inverse of the central *t*-distribution

Parameters:

x: A real number

deg_freedom: An integer greater 0, indicating the degrees of freedom

7.13.3.1 Quantiles: algorithms and formulas

Let p be a right tail probability, z_α the α -quantile of the standard normal distribution, and n the degrees of freedom. Depending on n and p , one proceeds as follows:

$$t_{1,\alpha} = \tan(x), \quad \text{where } x = ((1 - \alpha) - 0.5)\pi. \quad (7.13.10)$$

$$t_{2,\alpha} = \sqrt{2x/(1-x)}, \quad \text{where } x = (2\alpha - 1)^2. \quad (7.13.11)$$

$$t_{4,\alpha} = 2\sqrt{\cos(\arccos(x)/3)/x - 1}, \quad \text{where } x = 2\sqrt{\alpha(1-\alpha)}. \quad (7.13.12)$$

Otherwise, the quantile is approximated as (Peizer & Pratt, 1968)

$$t_{n,\alpha} \approx 2\sqrt{n \exp(z_\alpha^2/d^2) - 1}, \quad \text{where } d = \frac{n - 2/3 + 1/(10n)}{\sqrt{n - 5/6}} \quad (7.13.13)$$

7.13.4 Properties

Function **TDistInfo**(*n* As mpNum, **Output** As String) As mpNumList

NOT YET IMPLEMENTED

The function **TDistInfo** 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.13.4.1 Moments: algorithms and formulas

The algebraic moments (defined for $n > r$) are given by

$$\mu'_r = \left(\frac{1}{2}n\right)^{r/2} \frac{\Gamma\left(\frac{1}{2}(n-r)\right)}{\Gamma\left(\frac{1}{2}n\right)}. \quad (7.13.14)$$

7.13.4.2 Derivatives (central)

$$f_t(n; x) = \frac{n}{x} \left[F_t\left(n+2; x\sqrt{1+2/n}\right) - F_t(n; x) \right], \quad x \neq 0 \quad (7.13.15)$$

7.13.4.3 Relationships to other distributions (central)

$$F_t(n, x) = F_F\left(n, n; \left[n + 2x^2 + 2x\sqrt{n+x^2}\right]/n\right) \quad (7.13.16)$$

$$F_t(n, x) = F_B\left(\frac{1}{2}n, \frac{1}{2}n; \frac{1}{2}(x+1)/\sqrt{n+x^2}\right) \quad (7.13.17)$$

where $F_F(\cdot)$ denotes the cdf of the central F -distribution (see section 7.6.2.2) and $F_B(\cdot)$ denotes the cdf of the central Beta-distribution (see section 7.2.2.2).

7.13.5 Random Numbers

Function **TDistRan**(*Size* As *mpNum*, *n* As *mpNum*, *Generator* As *String*, *Output* As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 7.1.3.6 for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section 7.13.5.

7.13.5.1 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_n* a chi-squared variable with *n* degrees of freedom. To obtain random numbers from these distributions see the appropriate sections.

7.13.6 Behrens-Fisher Problem

See [Golhar \(1972\)](#)

7.14 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 $(0, +\infty)$:

7.14.1 Density and CDF

Function **WeibullDist**(*x* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNum-List

NOT YET IMPLEMENTED

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 [7.1.3.1](#) for the options for *Output*. Algorithms and formulas are given in sections [7.2.2.1](#) and [7.2.2.2](#).

The following functions are provided for compatibility with established spreadsheet functions

WorksheetFunction.**WEIBULL**(*x* As mpReal, *a* As mpNum, *b* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.WEIBULL returns the CDF and of the Weibull distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

a: A real number greater 0, representing the numerator degrees of freedom

b: A real number greater 0, representing the denominator degrees of freedom

WorksheetFunction.**WEIBULL.DIST**(*x* As mpReal, *a* As mpNum, *b* As mpNum, *Cumulative* As Boolean) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.WEIBULL.DIST returns the CDF and of the Weibull distribution

Parameters:

x: A real number. The numeric value at which to evaluate the distribution

a: A real number greater 0, representing the numerator degrees of freedom

b: A real number greater 0, representing the denominator degrees of freedom

Cumulative : A logical value that determines the form of the function. If cumulative is TRUE, T.DIST returns the cumulative distribution function; if FALSE, it returns the probability density function

7.14.1.1 Density

$$f(x) = \frac{x}{b^2} \exp\left(-\frac{x^2}{2b^2}\right) \exp(-(x/b)^a) \quad (7.14.1)$$

7.14.1.2 CDF

$$F(x) = 1 - \exp(-(x/b)^a) = -\text{expm1}(-(x/b)^a) \quad (7.14.2)$$

7.14.2 Quantiles

Function **WeibullDistInv**(*Prob* As mpNum, *a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function WeibullDistInv 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 7.1.3.2 for the options for *Prob* and *Output*).

$$F^{-1}(y) = b(-\text{ln1p}(-y))^{1/a} \quad (7.14.3)$$

7.14.3 Properties

Function **WeibullDistInfo**(*a* As mpNum, *b* As mpNum, *Output* As String) As mpNumList

NOT YET IMPLEMENTED

The function WeibullDistInfo 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 7.1.3.3 for the options for *Output*. Algorithms and formulas are given in section 7.13.4.

7.14.3.1 Moments: algorithms and formulas

$$\mu'_r = \sum_{j=0}^r \binom{r}{j} \Gamma\left(\frac{r-j}{c} + 1\right) b^{r-j} \quad (7.14.4)$$

$$\mu_1 = b \Gamma\left(\frac{1}{c} + 1\right) \quad (7.14.5)$$

$$\mu_2 = b^2 \left[\Gamma \left(\frac{1}{c} + 1 \right) \Gamma^2 \left(\frac{1}{c} + 1 \right) \right] \quad (7.14.6)$$

See [Rinne \(2008\)](#) for further details.

7.14.4 Random Numbers

Function **WeibullDistRandom**(*Size* As *mpNum*, *a* As *mpNum*, *b* As *mpNum*, **Generator** As *String*, **Output** As *String*) As *mpNumList*

NOT YET IMPLEMENTED

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 [7.1.3.6](#) for the options for *Size*, *Generator* and *Output*. Algorithms and formulas are given in section [7.6.5](#).

Chapter 8

Statistical Functions

8.1 Frequencies and Percentages

8.1.1 Count, CountA

WorksheetFunction.**COUNT**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COUNT returns the number of cells that contain numbers, and counts numbers within the list of arguments.

Parameter:

x: An array of real numbers.

Use the COUNT function to get the number of entries in a number field that is in a range or array of numbers.

WorksheetFunction.**COUNTA**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COUNTA returns the number of cells that are not empty in a range.

Parameter:

x: An array of real numbers.

The COUNTA function counts the number of cells that are not empty in a range (range: Two or more cells on a sheet. The cells in a range can be adjacent or nonadjacent.).

WorksheetFunction.**COUNTBLANK**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COUNTBLANK returns the number of empty cells in a specified range of cells.

Parameter:

x: An array of real numbers.

WorksheetFunction.**COUNTIF**(*x* As *mpNum*[], *Criteria* As String) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COUNTIF returns the number of cells within a range that meet a single criterion that you specify.

Parameters:

x: An array of real numbers.

Criteria: A String specifying the criteria.

For example, you can count all the cells that start with a certain letter, or you can count all the cells that contain a number that is larger or smaller than a number you specify. For example, suppose you have a worksheet that contains a list of tasks in column A, and the first name of the person assigned to each task in column B. You can use the COUNTIF function to count how many times a person's name appears in column B and, in that way, determine how many tasks are assigned to that person.

WorksheetFunction.**COUNTIFS**(*x* As *mpNumList*, *Criteria* As String[]) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.COUNTIFS returns the number of cells within multiple ranges that meet all criteria that you specify.

Parameters:

x: An array of real numbers.

Criteria: An array of strings specifying the criteria.

Reference to Klemens ([Klemens, 2008](#)).

See [Ogita et al. \(2005\)](#)

8.1.2 Frequency and 1D Histogram

WorksheetFunction.**FREQUENCY**(*DataArray* As *mpNum*[], *BinsArray* As *mpNum*[]) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.FREQUENCY returns a vertical array of numbers, calculating how often values occur within a range of values.

Parameters:

DataArray: An array of a set of values for which you want to count frequencies.

BinsArray: An array of intervals into which you want to group the values in *DataArray*

For example, use FREQUENCY to count the number of test scores that fall within ranges of scores. Because FREQUENCY returns an array, it must be entered as an array formula.

If *DataArray* contains no values, FREQUENCY returns an array of zeros. If *BinsArray* contains no values, FREQUENCY returns the number of elements in *DataArray*.

FREQUENCY is entered as an array formula after you select a range of adjacent cells into which you want the returned distribution to appear. The number of elements in the returned array is one more than the number of elements in *BinsArray*. The extra element in the returned array

returns the count of any values above the highest interval. For example, when counting three ranges of values (intervals) that are entered into three cells, be sure to enter **FREQUENCY** into four cells for the results. The extra cell returns the number of values in **DataArray** that are greater than the third interval value. **FREQUENCY** ignores blank cells and text.

WorksheetFunction.Histogram(*DataArray* As *mpNum*[], *BinsArray* As *mpNum*[]) As *mpNum*

NOT YET IMPLEMENTED

The function **WorksheetFunction.Histogram** returns a vertical array of numbers, calculating how often values occur within a range of values.

Parameters:

DataArray: An array of a set of values for which you want to count frequencies.

BinsArray: An array of intervals into which you want to group the values in **DataArray**

This section covers Histogram, as implemented in Excel Toolpak

8.2 Transformations

8.2.1 Linear Transformations (CONVERT)

WorksheetFunction.**CONVERT**(*Number* As *mpNum*, *FromUnit* As *String*, *ToUnit* As *String*)
As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.CONVERT returns a number converted from one measurement system to another.

Parameters:

Number: The value in FromUnit to convert.

FromUnit: The units for *Number*.

ToUnit: The units for the result.

For example, CONVERT can translate a table of distances in miles to a table of distances in kilometers.

If the input data types are incorrect, CONVERT returns the #VALUE! error value.

If the unit does not exist, CONVERT returns the #N/A error value.

If the unit does not support an abbreviated unit prefix, CONVERT returns the #N/A error value.

If the units are in different groups, CONVERT returns the #N/A error value.

Unit names and prefixes are case-sensitive. CONVERT accepts the following text values (in quotation marks) for FromUnit and ToUnit:

8.2.1.1 Weight and mass

Gram "g"

Slug "sg"

Pound mass (avoirdupois) "lbm"

U (atomic mass unit) "u"

Ounce mass (avoirdupois) "ozm"

8.2.1.2 Distance

Meter "m"

Statute mile "mi"

Nautical mile "Nmi"

Inch "in"

Foot "ft"

Yard "yd"

Angstrom "ang"

Pica "pica"

8.2.1.3 Time

Year "yr"

Day "day"

Hour "hr"

Minute "mn"

Second "sec"

8.2.1.4 Pressure

Pascal "Pa" (or "p")

Atmosphere "atm" (or "at")

mm of Mercury "mmHg"

8.2.1.5 Force

Newton "N"

Dyne "dyn" (or "dy")

Pound force "lbf"

8.2.1.6 Energy

Joule "J"

Erg "e"

Thermodynamic calorie "c"

IT calorie "cal"

Electron volt "eV" (or "ev")

Horsepower-hour "HPh" (or "hh")

Watt-hour "Wh" (or "wh")

Foot-pound "flb"

BTU "BTU" (or "btu")

8.2.1.7 Power

Horsepower "HP" (or "h")

Watt "W" (or "w")

8.2.1.8 Magnetism

Tesla "T"

Gauss "ga"

8.2.1.9 Temperature

Degree Celsius "C" (or "cel")

Degree Fahrenheit "F" (or "fah")

Kelvin "K" (or "kel")

8.2.1.10 Liquid measure

Teaspoon "tsp"

Tablespoon "tbs"

Fluid ounce "oz"

Cup "cup"

U.S. pint "pt" (or "us_pt")

U.K. pint "uk_pt"

Quart "qt"

Gallon "gal"
Liter "l" (or "lt")

8.2.1.11 Prefix (Multiplier)

exa 1E+18 "E"
peta 1E+15 "P"
tera 1E+12 "T"
giga 1E+09 "G"
mega 1E+06 "M"
kilo 1E+03 "k"
hecto 1E+02 "h"
deka 1E+01 "e"
deci 1E-01 "d"
centi 1E-02 "c"
milli 1E-03 "m"
micro 1E-06 "u"
nano 1E-09 "n"
pico 1E-12 "p"
femto 1E-15 "f"
atto 1E-18 "a"

8.2.2 Standardization

WorksheetFunction.**STANDARDIZE**(*Number* As mpNum, *Mean* As mpNum, *StDev* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.STANDARDIZE returns a normalized value Z from a distribution with mean μ and standard deviation σ .

Parameters:

Number: The value you want to normalize.

Mean: The arithmetic mean μ of the distribution.

StDev: The standard deviation σ of the distribution.

The equation for the normalized value Z is: $Z = \frac{x - \mu}{\sigma}$.

$$Z = \frac{x - \mu}{\sigma}. \quad (8.2.1)$$

8.2.3 Trimming and Winsorizing

WorksheetFunction.**TRIMMEAN**(*X* As mpNum[], *Percent* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.TRIMMEAN returns the mean of the interior of a data set.

Parameters:

X: The array or range of values to trim and average.

Percent: The fractional number of data points to exclude from the calculation. Calculates the mean taken by excluding a percentage of data points from the top and bottom tails of a data set. You can use this function when you wish to exclude outlying data from your analysis. For example, if *Percent* = 0.2, 4 points are trimmed from a data set of 20 points (20×0.2): 2 from the top and 2 from the bottom of the set.

If *Percent* < 0 or *Percent* > 1, **TRIMMEAN** returns the #NUM! error value. **TRIMMEAN** rounds the number of excluded data points down to the nearest multiple of 2. If *Percent* = 0.1, 10 percent of 30 data points equals 3 points. For symmetry, **TRIMMEAN** excludes a single value from the top and bottom of the data set.

8.3 Sums, Means, Moments and Cumulants

8.3.1 Sum

WorksheetFunction.**SUM**(*x* As *mpNum*[]) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.SUM returns the sum of the numbers in an array

Parameter:

x: An array of real numbers.

WorksheetFunction.**SUMA**(*x* As *mpNumList*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMA returns the sum of the numbers in a list of arguments

Parameter:

x: An array of real numbers.

Text and FALSE in arguments are evaluated as zero; TRUE is evaluated as 1.

WorksheetFunction.**SUMIF**(*x* As *mpNum*[], *Criteria* As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMIF returns the sum of cells within a range that meet a single criterion that you specify.

Parameters:

x: An array of real numbers.

Criteria: A String specifying the criteria.

The **SUMIF** function You use the SUMIF function to sum the values in a range that meets criteria that you specify. For example, suppose that in a column that contains numbers, you want to sum only the values that are larger than 5. You can use the following formula:

=SUMIF(B2:B25, ">5")

WorksheetFunction.**SUMIFS**(*x* As *mpNumList*, *Criteria* As *String*[]) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.SUMIFS returns the sum of cells within multiple ranges that meet all criteria that you specify.

Parameters:

x: An array of real numbers.

Criteria: An array of strings specifying the criteria.

8.3.2 Arithmetic Mean

WorksheetFunction.**AVERAGE**(*x* As *mpNum*[]) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.AVERAGE returns the average (arithmetic mean) of the numbers in an array

Parameter:

x: An array of real numbers.

WorksheetFunction.**AVERAGEA**(*x* As *mpNumList*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.AVERAGEA returns the average (arithmetic mean) of the numbers in a list of arguments

Parameter:

x: An array of real numbers.

Text and FALSE in arguments are evaluated as zero; TRUE is evaluated as 1.

WorksheetFunction.**AVERAGEIF**(*x* As *mpNum*[], *Criteria* As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.AVERAGEIF returns the average (arithmetic mean) of cells within a range that meet a single criterion that you specify.

Parameters:

x: An array of real numbers.

Criteria: A String specifying the criteria.

The SUMIF function You use the SUMIF function to sum the values in a range that meets criteria that you specify. For example, suppose that in a column that contains numbers, you want to sum only the values that are larger than 5. You can use the following formula:

=AVERAGEIF(B2:B25, ">5")

WorksheetFunction.**AVERAGEIFS**(*x* As *mpNumList*, *Criteria* As *String*[]) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.AVERAGEIFS returns the average (arithmetic mean) of cells within multiple ranges that meet all criteria that you specify.

Parameters:

x: An array of real numbers.

Criteria: An array of strings specifying the criteria.

8.3.3 Geometric Mean

WorksheetFunction.**GEOMEAN**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.GEOMEAN returns the geometric mean of an array or range of positive data.

Parameter:

x: An array of real numbers.

For example, you can use GEOMEAN to calculate average growth rate given compound interest with variable rates. If any data point ≤ 0 , GEOMEAN returns the #NUM! error value. The equation for the geometric mean is:

$$GM = \sqrt[n]{x_1 \times x_2 \times x_3, \dots, \times x_n} \quad (8.3.1)$$

8.3.4 Harmonic Mean

WorksheetFunction.**HARMEAN**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.HARMEAN returns the harmonic mean of an array or range of positive data.

Parameter:

x: An array of real numbers.

The harmonic mean is the reciprocal of the arithmetic mean of reciprocals. If any data point ≤ 0 , HARMEAN returns the #NUM! error value. The equation for the harmonic mean is:

$$\frac{1}{HM} = \frac{1}{n} \sum_{i=1}^n \frac{1}{x_i} \quad (8.3.2)$$

8.3.5 Variance

WorksheetFunction.**VAR**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.VAR returns the sample variance s^2 of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The sample variance s^2 of a sample of size n is calculated using the formula

$$s^2 = \frac{\sum(x - \bar{x})^2}{(n - 1)} \quad (8.3.3)$$

where \bar{x} denotes the arithmetic mean of the sample.

WorksheetFunction.VAR.S(x As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function **WorksheetFunction.VAR.S** returns the sample variance s^2 of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The sample variance s^2 of a sample of size n is calculated as in equation 8.3.3.

WorksheetFunction.VARA(x As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function **WorksheetFunction.VARA** returns the sample variance s^2 of an array or range of data, including text entries and FALSE as 0 and TRUE as 1.

Parameter:

x: An array of real numbers.

The sample variance s^2 of a sample of size n is calculated as in equation 8.3.3.

WorksheetFunction.VARP(x As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function **WorksheetFunction.VARP** returns the population variance S^2 of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The population variance S^2 of a population of size n is calculated using the formula

$$S^2 = \frac{\sum(x - \bar{x})^2}{n} \quad (8.3.4)$$

where \bar{x} denotes the arithmetic mean of the population.

WorksheetFunction.VAR.P(x As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function **WorksheetFunction.VAR.P** returns the population variance S^2 of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The population variance S^2 of a population of size n is calculated as in equation 8.3.4.

WorksheetFunction.VARPA(x As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.VARPA` returns the population variance S^2 of an array or range of data, including text entries and FALSE as 0 and TRUE as 1.

Parameter:

x: An array of real numbers.

The population variance S^2 of a population of size n is calculated as in equation 8.3.4.

8.3.6 Standard Deviation

`WorksheetFunction.STDEV(x As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.STDEV` returns the sample standard deviation s of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The sample standard deviation s of a sample of size n is calculated using the formula

$$s = \sqrt{\frac{\sum(x - \bar{x})^2}{(n - 1)}} \quad (8.3.5)$$

where \bar{x} denotes the arithmetic mean of the sample.

`WorksheetFunction.STDEV.S(x As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.STDEV.S` returns the sample standard deviation s of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The sample standard deviation s of a sample of size n is calculated as in equation 8.3.3.

`WorksheetFunction.STDEVA(x As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.STDEVA` returns the sample standard deviation s of an array or range of data, including text entries and FALSE as 0 and TRUE as 1.

Parameter:

x: An array of real numbers.

The sample standard deviation s of a sample of size n is calculated as in equation 8.3.3.

`WorksheetFunction.STDEVP(x As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.STDEVP` returns the population standard deviation S of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The population standard deviation S of a population of size n is calculated using the formula

$$S = \sqrt{\frac{\sum(x - \bar{x})^2}{n}} \quad (8.3.6)$$

where \bar{x} denotes the arithmetic mean of the population.

WorksheetFunction.**STDEV.P**(*x* As mpNum $[]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.STDEV.P returns the population standard deviation S of an array or range of numerical data, ignoring non-numeric entries.

Parameter:

x: An array of real numbers.

The population standard deviation S of a population of size n is calculated as in equation 8.3.4.

WorksheetFunction.**STDEVPA**(*x* As mpNum $[]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.STDEVPA returns the population standard deviation S of an array or range of data, including text entries and FALSE as 0 and TRUE as 1.

Parameter:

x: An array of real numbers.

The population standard deviation S of a population of size n is calculated as in equation 8.3.4.

8.3.7 Average Deviation

WorksheetFunction.**AVEDEV**(*x* As mpNum $[]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.AVEDEV returns the average of the absolute deviations of data points from their mean.

Parameter:

x: An array of real numbers.

AVEDEV is a measure of the variability in a data set. The equation for AVEDEV is:

$$\text{AVEDEV} = \frac{1}{n} \sum |x - \bar{x}| \quad (8.3.7)$$

where \bar{x} denotes the arithmetic mean of the sample.

8.3.8 Sum of Squares of Deviations (DEVSQ)

WorksheetFunction.**DEVSQ**(*x* As mpNum $[]$) As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.DEVSQ` returns the sum of squares of deviations of data points from their mean.

Parameter:

x: An array of real numbers.

`DEVSQ` is a measure of the variability in a data set. The equation for `DEVSQ` is:

$$\text{DEVSQ} = \sum_{i=1}^n (x_i - \bar{x})^2 \quad (8.3.8)$$

where \bar{x} denotes the arithmetic mean of the sample.

8.3.9 Skewness

`WorksheetFunction.SKEW(x As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.SKEW` returns the skewness of a sample.

Parameter:

x: An array of real numbers.

Skewness characterizes the degree of asymmetry of a distribution around its mean. Positive skewness indicates a distribution with an asymmetric tail extending toward more positive values. Negative skewness indicates a distribution with an asymmetric tail extending toward more negative values. The equation for `SKEW` is:

$$\text{SKEW} = \frac{n}{(n-1)(n-2)s^3} \sum_{i=1}^n (x_i - \bar{x})^3 \quad (8.3.9)$$

where \bar{x} denotes the arithmetic mean and s denotes the standard deviation of the sample.

`WorksheetFunction.SKEW.P(x As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.SKEW.P` returns the skewness of a population.

Parameter:

x: An array of real numbers.

Skewness characterizes the degree of asymmetry of a distribution around its mean. Positive skewness indicates a distribution with an asymmetric tail extending toward more positive values. Negative skewness indicates a distribution with an asymmetric tail extending toward more negative values. The equation for `SKEW.P` is:

$$\text{SKEW.P} = \frac{1}{n\sigma^3} \sum_{i=1}^n (x_i - \bar{x})^3 \quad (8.3.10)$$

where \bar{x} denotes the arithmetic mean and σ denotes the standard deviation of the population.

8.3.10 Kurtosis

WorksheetFunction.KURT(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.KURT returns the kurtosis of a sample.

Parameter:

x: An array of real numbers.

Kurtosis characterizes the relative peakedness or flatness of a distribution compared with the normal distribution. Positive kurtosis indicates a relatively peaked distribution. Negative kurtosis indicates a relatively flat distribution. Kurtosis is defined as:

$$\text{KURT} = \left[\frac{n(n+1)}{(n-1)(n-2)(n-3)s^4} \sum_{i=1}^n (x_i - \bar{x})^4 \right] - \frac{3(n-1)^2}{(n-2)(n-3)} \quad (8.3.11)$$

where \bar{x} denotes the arithmetic mean and s denotes the standard deviation of the sample.

8.4 Min, Max, Median, Percentiles

8.4.1 Minimum

WorksheetFunction.**MIN**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.MIN returns the smallest number in an array, ignoring non-numerical values,

Parameter:

x: An array of real numbers.

WorksheetFunction.**MINA**(*x* As *mpNumList*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.MINA returns the smallest number in an array; Text and FALSE in arguments are evaluated as zero; TRUE is evaluated as 1.

Parameter:

x: An array of real numbers.

8.4.2 Maximum

WorksheetFunction.**MAX**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.MAX returns the largest number in an array, ignoring non-numerical values,

Parameter:

x: An array of real numbers.

WorksheetFunction.**MAXA**(*x* As *mpNumList*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.MAXA returns the largest number in an array; Text and FALSE in arguments are evaluated as zero; TRUE is evaluated as 1.

Parameter:

x: An array of real numbers.

8.4.3 Median

WorksheetFunction.**MEDIAN**(*x* As *mpNum[]*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.MEDIAN returns the median in an array, ignoring non-numerical values.

Parameter:

x: An array of real numbers.

The median is the number in the middle of a set of numbers.

8.4.4 Mode

WorksheetFunction.MODE(*x* As mpNum $[]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.MODE returns the most frequently occurring, or repetitive, value in an array or range of data.

Parameter:

x: An array of real numbers.

WorksheetFunction.MODE.SNGL(*x* As mpNum $[]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.MODE.SNGL returns the most frequently occurring, or repetitive, value in an array or range of data.

Parameter:

x: An array of real numbers.

WorksheetFunction.MODE.MULT(*x* As mpNum $[]$) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.MODE.MULT returns a vertical array of the most frequently occurring, or repetitive values in an array or range of data.

Parameter:

x: An array of real numbers.

For horizontal arrays, use TRANSPOSE(MODE.MULT(number1,number2,...)). This will return more than one result if there are multiple modes. Because this function returns an array of values, it must be entered as an array formula.

8.4.5 *K*-th Largest Number

WorksheetFunction.LARGE(*x* As mpNum $[]$, *k* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.LARGE returns the *k*th largest value in a data set.

Parameters:

x: The array or range of data for which you want to determine the *k*th largest value.

k: The position (from the largest) in the array or cell range of data to return.

You can use this function to select a value based on its relative standing. For example, you can use LARGE to return the highest, runner-up, or third-place score.

8.4.6 *K*-th Smallest Number

WorksheetFunction.**SMALL**(*x* As mpNum[], *k* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SMALL returns the *k*th smallest value in a data set.

Parameters:

- x*: The array or range of data for which you want to determine the *k*th smallest value.
- k*: The position (from the smallest) in the array or cell range of data to return.

8.4.7 Percentile

WorksheetFunction.**PERCENTILE**(*x* As mpNum[], *k* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.PERCENTILE returns the *k*-th percentile of values in a data set as a percentage (0..1, inclusive) of the data set.

Parameters:

- x*: The array or range of data with numeric values that defines relative standing.
- k*: The percentile value in the range 0..1, inclusive.

WorksheetFunction.**PERCENTILE.INC**(*x* As mpNum[], *k* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.PERCENTILE.INC returns the *k*-th percentile of values in a data set as a percentage (0..1, inclusive) of the data set.

Parameters:

- x*: The array or range of data with numeric values that defines relative standing.
- k*: The percentile value in the range 0..1, inclusive.

WorksheetFunction.**PERCENTILE.EXC**(*x* As mpNum[], *k* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.PERCENTILE.EXC returns the *k*-th percentile of values in a data set as a percentage (0..1, exclusive) of the data set.

Parameters:

- x*: The array or range of data with numeric values that defines relative standing.
- k*: The percentile value in the range 0..1, exclusive.

You can use this function to establish a threshold of acceptance. For example, you can decide to examine candidates who score above the 90th percentile.

8.4.8 PercentRank

WorksheetFunction.**PERCENTRANK**(*Data* As mpNum[], *x* As mpNum, *digits* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.PERCENTRANK returns the rank of a value in a data set as a percentage (0..1, inclusive) of the data set.

Parameters:

Data: The array or range of data for which you want to determine the k^{th} smallest value.

x: The value for which you want to know the rank.

digits: A value that identifies the number of significant digits for the returned percentage value. If omitted, three digits (0.xxx) are used.

WorksheetFunction.**PERCENTRANK.INC**(*Data* As mpNum[], *x* As mpNum, *digits* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.PERCENTRANK.INC returns the rank of a value in a data set as a percentage (0..1, inclusive) of the data set.

Parameters:

Data: The array or range of data for which you want to determine the k^{th} smallest value.

x: The value for which you want to know the rank.

digits: A value that identifies the number of significant digits for the returned percentage value. If omitted, three digits (0.xxx) are used.

WorksheetFunction.**PERCENTRANK.EXC**(*Data* As mpNum[], *x* As mpNum, *digits* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.PERCENTRANK.EXC returns the rank of a value in a data set as a percentage (0..1, exclusive) of the data set.

Parameters:

Data: The array or range of data for which you want to determine the k^{th} smallest value.

x: The value for which you want to know the rank.

digits: A value that identifies the number of significant digits for the returned percentage value. If omitted, three digits (0.xxx) are used.

These function can be used to evaluate the relative standing of a value within a data set. For example, you can use PERCENTRANK to evaluate the standing of an aptitude test score among all scores for the test.

8.4.9 Quartile

WorksheetFunction.**QUARTILE**(*x* As mpNum[], *Quart* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.QUARTILE` returns the quartile of a data set, based on percentile values from 0..1, inclusive.

Parameters:

x: The array or cell range of numeric values for which you want the quartile value.

Quart: Indicates which quartile to return.

`WorksheetFunction.QUARTILE.INC(x As mpNum[], Quart As mpNum) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.QUARTILE.INC` returns quartile of a data set, based on percentile values from 0..1, inclusive.

Parameters:

x: The array or cell range of numeric values for which you want the quartile value.

Quart: Indicates which quartile to return.

`WorksheetFunction.QUARTILE.EXC(x As mpNum[], Quart As mpNum) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.QUARTILE.EXC` returns quartile of a data set, based on percentile values from 0..1, exclusive.

Parameters:

x: The array or cell range of numeric values for which you want the quartile value.

Quart: Indicates which quartile to return.

Quart: Indicates which value to return:

0: Minimum value (not for QUARTILE.EXC)

1: First quartile (25th percentile)

2: Median value (50th percentile)

3: Third quartile (75th percentile)

4: Maximum value (not for QUARTILE.EXC)

8.4.10 Rank

`WorksheetFunction.RANK(x As mpNum, Data As mpNum[], order As mpNum) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.RANK` returns the rank of a number in a list of numbers.

Parameters:

x: The number whose rank you want to find.

Data: The array or cell range of numeric values for which you want the rank value.

order: A number specifying how to rank number.

This function gives duplicate numbers the same rank. However, the presence of duplicate numbers affects the ranks of subsequent numbers. For example, in a list of integers sorted in ascending order, if the number 10 appears twice and has a rank of 5, then 11 would have a rank of 7 (no number would have a rank of 6).

WorksheetFunction.RANK.EQ(*x* As mpNum, *Data* As mpNum[], *order* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function **WorksheetFunction.RANK.EQ** returns the rank of a number in a list of numbers.

Parameters:

x: The number whose rank you want to find.

Data: The array or cell range of numeric values for which you want the rank value.

order: A number specifying how to rank number.

This function gives duplicate numbers the same rank. However, the presence of duplicate numbers affects the ranks of subsequent numbers. For example, in a list of integers sorted in ascending order, if the number 10 appears twice and has a rank of 5, then 11 would have a rank of 7 (no number would have a rank of 6).

WorksheetFunction.RANK.AVG(*x* As mpNum, *Data* As mpNum[], *order* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function **WorksheetFunction.RANK.AVG** returns the (average) rank of a number in a list of numbers.

Parameters:

x: The number whose rank you want to find.

Data: The array or cell range of numeric values for which you want the rank value.

order: A number specifying how to rank number.

This function returns the rank of a number in a list of numbers: its size relative to other values in the list; if more than one value has the same rank, the average rank is returned.

Order: A number specifying how to rank number:

0: ranks number as if ref were a list sorted in descending order.

1: ranks number as if ref were a list sorted in ascending order.

8.4.11 Prob

WorksheetFunction.PROB(*XRange* As mpNum[], *ProbRange* As mpNum[], *LowerLimit* As mpNum, *UpperLimit* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function **WorksheetFunction.PROB** returns the probability that values in a range are between two limits.

Parameters:

XRange: The range of numeric values of *x* with which there are associated probabilities.

ProbRange: A set of probabilities associated with values in *XRange*.

LowerLimit: The lower bound on the value for which you want a probability.

UpperLimit: The optional upper bound on the value for which you want a probability.

If *UpperLimit* is not supplied, returns the probability that values in *XRange* are equal to *LowerLimit*.

8.5 Summary Tables of Aggregates

8.5.1 SUBTOTAL

WorksheetFunction.**SUBTOTAL**(*FunctionNum* As Integer, *Data* As mpNumList) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SUBTOTAL returns a subtotal in a list or database.

Parameters:

FunctionNum: The number that specifies which function to use in calculating subtotals within a list.

Data: An array of real numbers.

It is generally easier to create a list with subtotals by using the Subtotal command in the Outline group on the Data tab in the Excel desktop application. Once the subtotal list is created, you can modify it by editing the SUBTOTAL function.

FunctionNum: The number 1 to 11 (includes hidden values) or 101 to 111 (ignores hidden values) that specifies which function to use in calculating subtotals within a list.

1 (101): AVERAGE

2 (102): COUNT

3 (103): COUNTA

4 (104): MAX

5 (105): MIN

6 (106): PRODUCT

7 (107): STDEV

8 (108): STDEVP

9 (109): SUM

10 (110): VAR

11 (111): VARP

8.5.2 AGGREGATE

WorksheetFunction.**AGGREGATE**(*FunctionNum* As Integer, *Options* As Integer, *Data* As mpNumList, *k* As Integer) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.AGGREGATE returns a subtotal in a list or database.

Parameters:

FunctionNum: The number that specifies which function to use in calculating subtotals within a list.

Options: A numerical value that determines which values to ignore in the evaluation range for the function

Data: An array of real numbers.

k: A selection parameter required for the certain functions.

The AGGREGATE function can apply different aggregate functions to a list or database with the option to ignore hidden rows and error values.

FunctionNum Required. A number 1 to 19 that specifies which function to use:

- 1: AVERAGE
- 2: COUNT
- 3: COUNTA
- 4: MAX
- 5: MIN
- 6: PRODUCT
- 7: STDEV.S
- 8: STDEV.P
- 9: SUM
- 10: VAR.S
- 11: VAR.P
- 12: MEDIAN
- 13: MODE.SNGL
- 14: LARGE
- 15: SMALL
- 16: PERCENTILE.INC
- 17: QUARTILE.INC
- 18: PERCENTILE.EXC
- 19: QUARTILE.EXC

Options Required. A numerical value that determines which values to ignore in the evaluation range for the function:

- 0: or omitted Ignore nested SUBTOTAL and AGGREGATE functions
- 1: Ignore hidden rows, nested SUBTOTAL and AGGREGATE functions
- 2: Ignore error values, nested SUBTOTAL and AGGREGATE functions
- 3: Ignore hidden rows, error values, nested SUBTOTAL and AGGREGATE functions
- 4: Ignore nothing
- 5: Ignore hidden rows
- 6: Ignore error values
- 7: Ignore hidden rows and error values

Data1 Required. The first numeric argument for functions that take multiple numeric arguments for which you want the aggregate value.

k: Required for the following functions:

- LARGE(array,k)
- SMALL(array,k)
- PERCENTILE.INC(array,k)
- QUARTILE.INC(array,quart)
- PERCENTILE.EXC(array,k)
- QUARTILE.EXC(array,quart)

8.6 Confidence intervals and tests

8.6.1 Confidence Interval for the mean, with known variance

WorksheetFunction.**CONFIDENCE**(*Alpha* As mpNum, *SteDev* As mpNum, *N* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.CONFIDENCE returns the confidence interval for a population mean.

Parameters:

Alpha: The significance level used to compute the confidence level. The confidence level equals $100*(1 - \alpha)\%$, or in other words, an alpha of 0.05 indicates a 95 percent confidence level.

SteDev: The population standard deviation for the data range and is assumed to be known.

N: The sample size.

WorksheetFunction.**CONFIDENCE.NORM**(*Alpha* As mpNum, *SteDev* As mpNum, *N* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.CONFIDENCE.NORM returns the confidence interval for a population mean.

Parameters:

Alpha: The significance level used to compute the confidence level. The confidence level equals $100*(1 - \alpha)\%$, or in other words, an alpha of 0.05 indicates a 95 percent confidence level.

SteDev: The population standard deviation for the data range and is assumed to be known.

N: The sample size.

The confidence interval is calculated as

$$\bar{x} \pm z_{\alpha} \left(\frac{\sigma}{\sqrt{n}} \right) \quad (8.6.1)$$

8.6.2 Confidence Interval for the mean, with unknown variance

WorksheetFunction.**CONFIDENCE.T**(*Alpha* As mpNum, *SteDev* As mpNum, *N* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.CONFIDENCE.T returns the confidence interval for a population mean.

Parameters:

Alpha: The significance level used to compute the confidence level. The confidence level equals $100*(1 - \alpha)\%$, or in other words, an alpha of 0.05 indicates a 95 percent confidence level.

SteDev: The population standard deviation for the data range

N: The sample size.

The confidence interval is calculated as

$$\bar{x} \pm t_\alpha \left(\frac{\sigma}{\sqrt{n}} \right) \quad (8.6.2)$$

8.6.3 Gauss z-Tests

WorksheetFunction.ZTEST(*X* As mpNum[], *Mean* As mpNum, *Sigma* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.ZTEST returns the one-tailed P-value of a z-test.

Parameters:

X: The array or range of data against which to test Mean.

Mean: The value to test.

Sigma: The population (known) standard deviation. If omitted, the sample standard deviation is used.

WorksheetFunction.Z.TEST(*X* As mpNum[], *Mean* As mpNum, *Sigma* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.Z.TEST returns the one-tailed P-value of a z-test.

Parameters:

X: The array or range of data against which to test Mean.

Mean: The value to test.

Sigma: The population (known) standard deviation. If omitted, the sample standard deviation is used.

$$ZTEST(X, \mu_0, \sigma) = 1 - \Phi \left(\frac{\bar{x} - \mu_0}{\sigma / \sqrt{n}} \right) \quad (8.6.3)$$

where \bar{x} denotes the mean of X and n denotes the sample size of X .

Placeholder for z-test for 2 samples

8.6.4 Student's t-Test, 2 samples

WorksheetFunction.TTEST(*X* As mpNum[], *Y* As mpNum[], *Tails* As Integer, *Type* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.TTEST returns the probability associated with a Student's t-Test.

Parameters:

X: the first data set.

Y: the second data set.

Tails: specifies the number of distribution tails. If tails = 1, TTEST uses the one-tailed distribution. If tails = 2, TTEST uses the two-tailed distribution.

Type: the kind of t-Test to perform. type = 1 paired, type = 2 unpaired, equal variance (homoscedastic), type = 3 unpaired, unequal variance (heteroscedastic).

WorksheetFunction.**T.TEST**(*X* As mpNum[], *Y* As mpNum[], *Tails* As Integer, *Type* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.T.TEST returns the probability associated with a Student's t-Test.

Parameters:

X: the first data set.

Y: the second data set.

Tails: specifies the number of distribution tails. If tails = 1, TTEST uses the one-tailed distribution. If tails = 2, TTEST uses the two-tailed distribution.

Type: the kind of t-Test to perform. type = 1 paired, type = 2 unpaired, equal variance (homoscedastic), type = 3 unpaired, unequal variance (heteroscedastic).

8.6.5 F-Test (variances of 2 independent samples)

WorksheetFunction.**FTEST**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.FTEST returns the two-tailed probability that the variances in array1 and array2 are not significantly different.

Parameters:

X: the first data set.

Y: the second data set.

WorksheetFunction.**F.TEST**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.F.TEST returns the two-tailed probability that the variances in array1 and array2 are not significantly different.

Parameters:

X: the first data set.

Y: the second data set.

8.6.6 Anova: Single Factor

Function **ANOVA1**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function ANOVA1 returns the two-tailed probability that the variances in array1 and array2 are not significantly different.

Parameters:

X: the first data set.

Y: the second data set.

8.6.7 Anova: Two Factors (with or without replication)

Function **ANOVA2**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function ANOVA2 returns the two-tailed probability that the variances in array1 and array2 are not significantly different.

Parameters:

X: the first data set.

Y: the second data set.

8.6.8 Chi-Square-Test (Homogeneity)

WorksheetFunction.**CHITEST**(*A* As mpNum[], *E* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.CHITEST returns the probability that a value of the χ^2 statistic at least as high as the value calculated by the above formula could have happened by chance under the assumption of independence.

Parameters:

A: The range of data that contains observations to test against expected values.

E: The range of data that contains the ratio of the product of row totals and column totals to the grand total.

WorksheetFunction.**CHISQ.TEST**(*A* As mpNum[], *E* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.CHISQ.TEST returns the probability that a value of the χ^2 statistic at least as high as the value calculated by the above formula could have happened by chance under the assumption of independence.

Parameters:

A: The range of data that contains observations to test against expected values.

E: The range of data that contains the ratio of the product of row totals and column totals to the grand total.

The χ^2 test first calculates a χ^2 statistic using the formula:

$$\chi^2 = \sum_{i=1}^r \sum_{j=1}^c \frac{(A_{ij} - E_{ij})^2}{E_{ij}}, \quad (8.6.4)$$

where:

A_{ij} = actual frequency in the *i*-th row, *j*-th column

E_{ij} = expected frequency in the *i*-th row, *j*-th column

r = number of rows

c = number of columns

The test then uses the χ^2 distribution with an appropriate number of degrees of freedom, df . If $r > 1$ and $c > 1$, then $df = (r - 1)(c - 1)$. If $r = 1$ and $c > 1$, then $df = c - 1$ or if $r > 1$ and $c = 1$, then $df = r - 1$. $r = c = 1$ is not allowed.

8.7 Covariance and Correlation

8.7.1 Covariance

WorksheetFunction.**COVAR**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.COVAR returns the sample covariance $\text{cov}(X, Y)$

Parameters:

X: An array of real numbers.

Y: An array of real numbers.

The sample covariance $\text{cov}(X, Y)$ of a sample of size n is calculated using the formula

$$\text{cov}(X, Y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{(n - 1)} \quad (8.7.1)$$

where \bar{x} and \bar{y} denote the arithmetic means of the samples *X* and *Y*.

WorksheetFunction.**COVARIANCE.S**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.COVARIANCE.S returns the sample covariance $\text{cov}(X, Y)$

Parameters:

X: An array of real numbers.

Y: An array of real numbers.

The sample covariance $\text{cov}(X, Y)$ of a sample of size n is calculated as in equation 8.7.1

WorksheetFunction.**COVARIANCE.P**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.COVARIANCE.P returns the sample covariance $\text{cov}(X, Y)$

Parameters:

X: An array of real numbers.

Y: An array of real numbers.

The population covariance $\text{COV}(X, Y)$ of a population of size n is calculated using the formula

$$\text{COV}(X, Y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n} \quad (8.7.2)$$

where \bar{x} and \bar{y} denote the arithmetic means of the populations *X* and *Y*.

8.7.2 Correlation

WorksheetFunction.**CORREL**(*X* As mpNum[], *Y* As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.CORREL` returns the Pearson product moment correlation coefficient $r = \text{corr}(X, Y)$

Parameters:

X : An array of real numbers.

Y : An array of real numbers.

The Pearson product moment correlation coefficient $r = \text{corr}(X, Y)$ of a sample of size n is calculated using the formula

$$r = \text{corr}(X, Y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}} \quad (8.7.3)$$

where \bar{x} and \bar{y} denote the arithmetic means of the samples X and Y .

`WorksheetFunction.PEARSON(X As mpNum[], Y As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.PEARSON` returns the Pearson product moment correlation coefficient $r = \text{corr}(X, Y)$

Parameters:

X : An array of real numbers.

Y : An array of real numbers.

The sample covariance $\text{cov}(X, Y)$ of a sample of size n is calculated as in equation 8.7.3

`WorksheetFunction.RSQ(X As mpNum[], Y As mpNum[])` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.RSQ` returns r^2 , the square of the Pearson product moment correlation coefficient r , with $r = \text{corr}(X, Y)$

Parameters:

X : An array of real numbers.

Y : An array of real numbers.

r is calculated as in equation 8.7.3

8.7.3 Fisher's z-transform

`WorksheetFunction.FISHER(X As mpNum)` As mpNum

NOT YET IMPLEMENTED

The function `WorksheetFunction.FISHER` returns Fisher's z-transform

Parameter:

X : Areal numbers

The Fisher z -transform is defined by

$$Z(a) = \frac{1}{2} \log \left(\frac{1+a}{1-a} \right) = \text{atanh}(a) \quad (8.7.4)$$

WorksheetFunction.**FISHERINV**(*X* As mpNum) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.FISHERINV returns Fisher's inverse z-transform

Parameter:

X: Areal numbers

The inverse Fisher *z*-transform is defined by

$$Z^{-1}(a) = \frac{e^{2a} - 1}{e^{2a} + 1} = \tanh(a) \quad (8.7.5)$$

8.8 Linear Regression

8.8.1 INTERCEPT

WorksheetFunction.INTERCEPT(**X** As mpNum[], **Y** As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.INTERCEPT returns the point at which a line will intersect the y-axis by using linear regression.

Parameters:

X: An array of real numbers.

Y: An array of real numbers.

The equation for the intercept of the regression line, a , is:

$$a = \bar{y} - b\bar{x}, \text{ where } b = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}, \quad (8.8.1)$$

and \bar{x} and \bar{y} denote the arithmetic means of the samples X and Y .

8.8.2 SLOPE

WorksheetFunction.SLOPE(**X** As mpNum[], **Y** As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.SLOPE returns the slope of the linear regression line through data points in X and Y .

Parameters:

X: An array of real numbers.

Y: An array of real numbers.

The slope is the vertical distance divided by the horizontal distance between any two points on the line, which is the rate of change along the regression line. The equation for the slope of the regression line is:

$$b = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}, \quad (8.8.2)$$

and \bar{x} and \bar{y} denote the arithmetic means of the samples X and Y .

8.8.3 Forecast

WorksheetFunction.FORECAST(**X** As mpNum[], **Y** As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.FORECAST returns an y_0 -value for a given x_0 -value, using linear regression.

Parameters:

X: An array of real numbers.

Y: An array of real numbers.

Calculates an y_0 -value for a given x_0 -value, using linear regression. The equation for FORECAST is

$$y_0 = f(x_0) = a + bx_0, \text{ where } a = \bar{y} - b\bar{x}, \text{ and } b = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}, \quad (8.8.3)$$

and \bar{x} and \bar{y} denote the arithmetic means of the samples X and Y .

8.8.4 SteYX

WorksheetFunction.**STEYX**(X As mpNum[], Y As mpNum[]) As mpNum

NOT YET IMPLEMENTED

The function WorksheetFunction.STEYX returns the standard error of the predicted y -value for each x in the regression.

Parameters:

X : An array of real numbers.

Y : An array of real numbers.

The standard error is a measure of the amount of error in the prediction of y for an individual x . The equation for STEYX is

$$\text{STEYX} = \sqrt{\frac{1}{n-2} \left[\sum_{i=1}^n (y_i - \bar{y})^2 - \frac{[\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})]^2}{\sum_{i=1}^n (x_i - \bar{x})^2} \right]}, \quad (8.8.4)$$

and \bar{x} and \bar{y} denote the arithmetic means of the samples X and Y .

8.9 Database related functions

The Database related functions share the following arguments:

Table: The range of cells that makes up the list or database. A database is a list of related data in which rows of related information are records, and columns of data are fields. The first row of the list contains labels for each column.

Field: Indicates which column is used in the function. Enter the column label enclosed between double quotation marks, such as "Age" or "Yield," or a number (without quotation marks) that represents the position of the column within the list: 1 for the first column, 2 for the second column, and so on.

Criteria: The range of cells that contains the conditions that you specify. You can use any range for the criteria argument, as long as it includes at least one column label and at least one cell below the column label in which you specify a condition for the column.

8.9.1 DGET

WorksheetFunction.**DGET**(*Table* As *mpNum[]*, *Field* As *String*, *Criteria* As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DGET returns a single value from a column of a list or database that matches conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

8.9.2 DPRODUCT

WorksheetFunction.**DPRODUCT**(*Table* As *mpNum[]*, *Field* As *String*, *Criteria* As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DPRODUCT returns the product of the values from a column of a list or database that matches conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

8.9.3 DCount, DCountA

WorksheetFunction.**DCOUNT**(*Table* As *mpNum[]*, *Field* As *String*, *Criteria* As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function `WorksheetFunction.DCOUNT` returns the number of cells that contain numbers in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

If field is omitted, `DCOUNT` counts all records in the database that match the criteria.

`WorksheetFunction.DCOUNTA(Table As mpNum[], Field As String, Criteria As String) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.DCOUNTA` returns the number of nonblank cells that contain numbers in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

If field is omitted, `DCOUNTA` counts all records in the database that match the criteria.

8.9.4 DSum

`WorksheetFunction.DSUM(Table As mpNum[], Field As String, Criteria As String) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.DSUM` returns the sum of cells that contain numbers in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

8.9.5 DAverage

`WorksheetFunction.DAVERAGE(Table As mpNum[], Field As String, Criteria As String) As mpNum`

NOT YET IMPLEMENTED

The function `WorksheetFunction.DAVERAGE` returns the arithmetic mean of cells that contain numbers in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

8.9.6 Variance

WorksheetFunction.DVAR(**Table** As *mpNum[]*, **Field** As *String*, **Criteria** As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DVAR returns the sample variance of cells that contain numbers in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

WorksheetFunction.DVARP(**Table** As *mpNum[]*, **Field** As *String*, **Criteria** As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DVARP returns the population variance of cells that contain numbers in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

8.9.7 Standard Deviation

WorksheetFunction.DSTDEV(**Table** As *mpNum[]*, **Field** As *String*, **Criteria** As *String*) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DSTDEV returns the sample standard deviation of cells that contain numbers in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

WorksheetFunction.DSTDEVP(**Table** As *mpNum[]*, **Field** As *String*, **Criteria** As *String*) As *mp-Num*

NOT YET IMPLEMENTED

The function WorksheetFunction.DSTDEVP returns the population standard deviation of cells that contain numbers in a field (column) of records in a list or database that match conditions that

you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

8.9.8 Minimum

WorksheetFunction.DMIN(***Table*** As *mpNum*[], ***Field*** As String, ***Criteria*** As String) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DMIN returns the smallest number in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

8.9.9 Maximum

WorksheetFunction.DMAX(***Table*** As *mpNum*[], ***Field*** As String, ***Criteria*** As String) As *mpNum*

NOT YET IMPLEMENTED

The function WorksheetFunction.DMAX returns the largest number in a field (column) of records in a list or database that match conditions that you specify.

Parameters:

Table: An array of real numbers, using Strings as headers.

Field: Indicates which column is used in the function.

Criteria: A String containing the criteria.

Part III

Special Functions

Chapter 9

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:

```
>>> mp.dps = 15; mp.pretty = True
>>> fac(100)
9.33262154439442e+157
>>> print int(_) # most digits are wrong
93326215443944150965646704795953882578400970373184098831012889540582227238570431
295066113089288327277825849664006524270554535976289719382852181865895959724032
>>> mp.dps = 160
>>> fac(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915
6089414639761565182862536979208272237582511852109168640000000000000000000000000000000000.0
```

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.

9.1 Factorials

9.1.1 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
```

9.1.2 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

$$x!! = \begin{cases} 1 \cdot 3 \cdots (x-2) \cdot x & \text{for } x \text{ odd} \\ 2 \cdot 4 \cdots (x-2) \cdot x & \text{for } x \text{ even} \end{cases} \quad (9.1.1)$$

and more generally by [1]

$$x!! = 2^{x/2} \left(\frac{\pi}{2}\right)^{(\cos(\pi x)-1)/4} \Gamma\left(\frac{x}{2} + 1\right) \quad (9.1.2)$$

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)
```

9.2 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.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \quad (9.2.1)$$

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
```

9.3 Pochhammer symbol, Rising and falling factorials

9.3.1 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

$$\text{poch1}(a, x) = \frac{(a)_x - 1}{x}, \quad (9.3.1)$$

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.

9.3.2 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,

$$x^{(n)} = x(x + 1) \cdots (x + n - 1) = \frac{\Gamma(x + n)}{\Gamma(x)} \quad (9.3.2)$$

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)
```

9.3.3 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,

$$x_{(n)} = x(x - 1) \cdots (x - n + 1) = \frac{\Gamma(x + 1)}{\Gamma(x - n + 1)} \quad (9.3.3)$$

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)
```

9.4 Super- and hyperfactorials

9.4.1 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:

$$sf(n) = \prod_{k=1}^n k! \quad (9.4.1)$$

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
```

9.4.2 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

$$H(n) = \prod_{k=1}^n k^k \quad (9.4.2)$$

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.

$$H(z) = \frac{\Gamma(z + 1)^z}{G(z)}. \quad (9.4.3)$$

The extension to complex numbers can also be done via the integral representation

$$H(z) = (2\pi)^{-z/2} \exp \left[\binom{z+1}{2} + \int_0^z \log(t!) dt \right]. \quad (9.4.4)$$

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)
```

9.4.3 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:

$$G(z+1) = (2\pi)^{z/2} e^{[z(z+1)+\gamma z^2]/2} \prod_{n=1}^{\infty} \left[\left(1 + \frac{z}{n}\right)^n e^{-z+z^2/(2n)} \right] \quad (9.4.5)$$

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)
```

9.5 Gamma functions

9.5.1 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

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \quad (9.5.1)$$

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
```

9.5.2 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
```

9.5.3 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:

$$\frac{\Gamma(a_0)\Gamma(a_1)\cdots\Gamma(a_p)}{\Gamma(b_0)\Gamma(b_1)\cdots\Gamma(b_p)} \quad (9.5.2)$$

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

$$\lim_{\epsilon \rightarrow 0} \frac{\Gamma(a_0 + \epsilon)\Gamma(a_1 + \epsilon)\cdots\Gamma(a_p + \epsilon)}{\Gamma(b_0 + \epsilon)\Gamma(b_1 + \epsilon)\cdots\Gamma(b_p + \epsilon)} \quad (9.5.3)$$

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
```

9.5.4 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

$$\ln \Gamma(z) \approx \frac{\ln(2\pi)}{2} + \left(z - \frac{1}{2}\right) \ln(z) - z + O(z^{-1}) \quad (9.5.4)$$

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)
```

9.5.5 Generalized incomplete gamma function

Function **gammaintc(z As mpNum, a As mpNum, b As mpNum, Keywords As String) As mpNum**

The function `gammaintc` 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:

$$\Gamma(z, a, b) = \int_a^b t^{z-1} e^{-t} dt \quad (9.5.5)$$

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`, `gammaintc()` computes the 'regularized' incomplete gamma function

$$P(z, a, b) = \frac{\Gamma(z, a, b)}{\Gamma(z)}. \quad (9.5.6)$$

Examples

We can compare with numerical quadrature to verify that `gammaintc()` computes the integral in the definition:

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> gammaintc(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)
```

9.5.6 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:

$$\frac{\partial}{\partial x} P(a, x) = \frac{e^{-x} x^{a-1}}{\Gamma(a)}. \quad (9.5.7)$$

9.5.7 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

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt \quad (9.5.8)$$

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

$$Q(a, x) = \frac{1}{\Gamma(a)} \int_x^{\infty} t^{a-1} e^{-t} dt \quad (9.5.9)$$

for $a \geq 0$ and $x \geq 0$.

9.5.8 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

$$\Gamma(a, x) = \int_0^x t^{a-1} e^{-t} dt \quad (9.5.10)$$

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

$$\gamma(a, x) = \int_x^{\infty} t^{a-1} e^{-t} dt \quad (9.5.11)$$

for $a \geq 0$ and $x \geq 0$.

Note: in Boost, the functions are referred to as TgammaLower and TgammaUpper.

9.5.9 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 γ^* , defined as

$$\gamma^*(a, x) = e^{-x} \frac{M(1, a + 1, x)}{\Gamma(a + 1)} \quad (9.5.12)$$

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:

$$\gamma^*(a, x) = \frac{x^{-a}}{\Gamma(a)} \gamma(a, x) = x^{-a} P(a, x). \quad (9.5.13)$$

9.5.10 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.

9.6 Polygamma functions and harmonic numbers

9.6.1 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:

$$\psi^{(m)}(z) = \left(\frac{d}{dz} \right)^{m+1} \log \Gamma(z). \quad (9.6.1)$$

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)`.

9.6.2 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)`.

9.6.3 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

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \quad (9.6.2)$$

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
```

9.7 Beta Functions

9.7.1 Beta function B(a, b)

9.7.2 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

$$B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1} dt \quad (9.7.1)$$

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
```

9.7.3 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.

9.7.4 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,

$$I_{x_1}^{x_2}(a, b) = \int_{x_1}^{x_2} t^{a-1}(1-t)^{b-1} dt \quad (9.7.2)$$

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
```

9.7.5 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 $0 \leq x \leq 1$:

$$B_x(a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt. \quad (9.7.3)$$

There are some special cases

$$B_0(a, b) = 0, \quad B_1(a, b) = B(a, b), \quad B_x(a, 1) = \frac{x^a}{a}, \quad B_x(1, b) = \frac{1 - (1-x)^b}{b}, \quad (9.7.4)$$

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

$$B_x(a, b) = \frac{x^a}{a} {}_2F_1(a, 1-b, a+1, x), \quad -a \notin \mathbb{N} \quad (9.7.5)$$

else if $b \neq 0$ is not a negative integer, the result is

$$B_x(a, b) = B(a, b) - \frac{(1-x)^b x^a}{b} {}_2F_1(1, a+b, b+1, 1-x), \quad -b \notin \mathbb{N}. \quad (9.7.6)$$

9.7.6 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 $0 \leq x \leq 1$:

$$I_x(a, b) = \frac{B_x(a, b)}{B(a, b)}, \quad B_x(a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt. \quad (9.7.7)$$

There are some special cases

$$I_0(a, b) = 0, \quad I_1(a, b) = 1, \quad I_x(a, 1) = x^a, \quad (9.7.8)$$

and the symmetry relation $I_x(a, b) = 1 - I_{1-x}(b, a)$, which is used for $x > a/(a+b)$.

Chapter 10

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).

10.1 Exponential integrals

10.1.1 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

$$\text{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t} dt \quad (10.1.1)$$

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)

10.1.2 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

$$E_1(x) = \int_z^\infty \frac{e^t}{t} dt \quad (10.1.2)$$

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
```

10.1.3 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

$$E_n(x) = \int_1^\infty \frac{e^{-zt}}{t^n} dt \quad (10.1.3)$$

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)
```

10.1.4 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 $p \in \mathbb{R}$

$$E_p(x) = x^{p-1} \int_x^\infty \frac{e^{-t}}{t^p} dt = \int_1^\infty \frac{e^{-xt}}{t^p} dt \quad (10.1.4)$$

with $x > 0$ for $p \leq 1$, and $x \geq 0$ for $p > 1$.

10.2 Logarithmic integral

10.2.1 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

$$\text{li}(x) = \int_0^x \frac{1}{\log(t)} dt \quad (10.2.1)$$

The logarithmic integral has a singularity at $x = 1$.

Alternatively, **li(x, offset=True)** computes the offset logarithmic integral (used in number theory)

$$\text{Li}(x) = \int_2^x \frac{1}{\log(t)} dt \quad (10.2.2)$$

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)
```

10.3 Trigonometric integrals

10.3.1 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

$$\text{Ci}(x) = \int_x^{\infty} \frac{\cos(t)}{t} dt = \gamma + \log(x) + \int_0^x \frac{\cos(t) - 1}{t} dt \quad (10.3.1)$$

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)
```

10.3.2 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

$$\text{Si}(x) = \int_0^x \frac{\sin(t)}{t} dt \quad (10.3.2)$$

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)
```

10.4 Hyperbolic integrals

10.4.1 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

$$\text{Chi}(x) = \int_x^{\infty} \frac{\cosh(t)}{t} dt = \gamma + \log(x) + \int_0^x \frac{\cosh(t) - 1}{t} dt \quad (10.4.1)$$

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)
```

10.4.2 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

$$\text{Shi}(x) = \int_0^x \frac{\sinh(t)}{t} dt \quad (10.4.2)$$

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)
```

10.5 Error functions

10.5.1 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,

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt \quad (10.5.1)$$

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

10.5.2 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)
```

10.5.3 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:

$$\text{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(t^2) dt \quad (10.5.2)$$

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)
```

10.5.4 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
```

10.6 The normal distribution

10.6.1 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 μ and variance σ^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
```

10.6.2 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 μ and variance σ^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
```

10.7 Fresnel integrals

10.7.1 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

$$S(x) = \int_0^x \sin\left(\frac{\pi t^2}{2}\right) dt \quad (10.7.1)$$

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)
```

10.7.2 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

$$C(x) = \int_0^x \cos\left(\frac{\pi t^2}{2}\right) dt \quad (10.7.2)$$

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)
```

10.8 Other Special Functions

10.8.1 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
```

10.8.2 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:

$$a_0 = a; \quad b_0 = b; \quad a_{n+1} = \frac{1}{2}(a_n + b_n); \quad b_{n+1} = \sqrt{a_n b_n}. \quad (10.8.1)$$

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 11

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).

11.1 Bessel functions

11.1.1 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 ν , $x \geq 0$ if ν is not an integer.

Parameters:

x: A real number.

ν: A real number.

11.1.2 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 ν , $x > 0$.

Parameters:

x: A real number.

ν: A real number.

11.1.3 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

$$x^2y'' + xy' + (x^2 - n^2)y = 0 \quad (11.1.1)$$

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 ${}_0F_1$:

$$J_n(x) = \frac{x^n}{2^n \Gamma(n+1)} {}_0F_1 \left(n+1, -\frac{x^2}{4} \right) \quad (11.1.2)$$

With derivative = *m* $\neq 0$, the *m*-th derivative

$$\frac{d^m}{dx^m} J_n(x) \quad (11.1.3)$$

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()**.

11.1.4 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

$$Y_n(x) = \frac{J_n(x) \cos(\pi n) - J_{-n}(x)}{\sin(\pi n)} \quad (11.1.4)$$

For *n* an integer, this formula should be understood as a limit. With derivative = *m* $\neq 0$, the *m*-th derivative

$$\frac{d^m}{dx^m} Y_n(x) \quad (11.1.5)$$

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
```

11.1.5 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

$$I_n(x) = i^{-n} J_n(ix) \quad (11.1.6)$$

With derivative = $m \neq 0$, the m -th derivative

$$\frac{d^m}{dx^m} I_n(x) \quad (11.1.7)$$

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
```

11.1.6 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

$$K_n(x) = \frac{\pi}{2} \frac{I_{-n}(x) - I_n(x)}{\sin(\pi n)} \quad (11.1.8)$$

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)
```

11.2 Bessel function zeros

11.2.1 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 ν and m .

The zeros are interlaced according to the inequalities

$$j'_{\nu,k} < j_{\nu,k} < j'_{\nu,k+1} \quad (11.2.1)$$

$$j_{\nu,1} < j_{\nu+1,2} < j_{\nu,2} < j'_{\nu,k+1} < j_{\nu+1,2} < j_{\nu,3} \quad (11.2.2)$$

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
```

11.2.2 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

$$y'_{\nu,k} < y_{\nu,k} < y'_{\nu,k+1} \quad (11.2.3)$$

$$y_{\nu,1} < y_{\nu+1,2} < y_{\nu,2} < y'_{\nu,k+1} < y_{\nu+1,2} < y_{\nu,3} \quad (11.2.4)$$

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
```

11.3 Hankel functions

11.3.1 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

$$H_n^{(1)}(x) = J_n(x) + iY_n(x). \quad (11.3.1)$$

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)
```

11.3.2 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

$$H_n^{(2)}(x) = J_n(x) - iY_n(x). \quad (11.3.2)$$

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)
```

11.4 Kelvin functions

11.4.1 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

$$J_n(xe^{3\pi i/4}) = \text{ber}_n(x) + i\text{bei}_n(x). \quad (11.4.1)$$

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)
```

11.4.2 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()**.

11.4.3 Kelvin function ker

Function **ker(*n* As mpNum, *z* As mpNum) As mpNum**

The function `ker` returns the Kelvin function `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

$$e^{-\pi i/2} K_n(xe^{3\pi i/4}) = \text{ker}_n(x) + i\text{kei}_n(x). \quad (11.4.2)$$

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)
```

11.4.4 Kelvin function `kei`

Function **kei(*n* As mpNum, *z* As mpNum)** As mpNum

The function `kei` returns the Kelvin function `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()`.

11.5 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]):

$$\mathbf{H}_\nu(x) = \left(\frac{1}{2}x\right)^{\nu+1} \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{1}{2}x\right)^{2k}}{\Gamma\left(k + \frac{3}{2}\right) \Gamma\left(k + \nu + \frac{3}{2}\right)} \quad (11.5.1)$$

$$\mathbf{L}_\nu(x) = \left(\frac{1}{2}x\right)^{\nu+1} \sum_{k=0}^{\infty} \frac{\left(\frac{1}{2}x\right)^{2k}}{\Gamma\left(k + \frac{3}{2}\right) \Gamma\left(k + \nu + \frac{3}{2}\right)} \quad (11.5.2)$$

11.5.1 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

$$\mathbf{H}_n(z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{\Gamma\left(k + \frac{3}{2}\right) \Gamma\left(k + n + \frac{3}{2}\right)} \left(\frac{z}{2}\right)^{2k+n+1} \quad (11.5.3)$$

which is a solution to the Struve differential equation

$$z^2 f''(z) + z f'(z) + (z^2 - n^2) f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!} \quad (11.5.4)$$

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, 1000000000000000)
3153915652525200060.308937
>>> struveh(2.5, -1000000000000000)
(0.0 - 3153915652525200060.308937j)
>>> struveh(1+j, 1000000+4000000j)
(-3.066421087689197632388731e+1737173 - 1.596619701076529803290973e+1737173j)
```

11.5.2 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

$$L_n(z) = -ie^{-n\pi i/2} H_n(iz) \quad (11.5.5)$$

which solves to the modified Struve differential equation

$$z^2 f''(z) + z f'(z) + (z^2 + n^2) f(z) = \frac{2z^{n+1}}{\pi(2n-1)!!} \quad (11.5.6)$$

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)
```

11.6 Anger-Weber functions

11.6.1 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

$$\mathbf{J}_\nu(z) = \frac{1}{\pi} \int_0^\pi \cos(\nu t - z \sin(t)) dt \quad (11.6.1)$$

which is an entire function of both the parameter ν and the argument z . It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z} f'(z) + \left(1 - \frac{\nu^2}{z^2}\right) f(z) = \frac{(z - \nu)}{\pi z^2} \sin(\pi\nu). \quad (11.6.2)$$

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
```

11.6.2 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

$$\mathbf{E}_\nu(z) = \frac{1}{\pi} \int_0^\pi \sin(\nu t - z \sin(t)) dt \quad (11.6.3)$$

which is an entire function of both the parameter ν and the argument z . It solves the inhomogeneous Bessel differential equation

$$f''(z) + \frac{1}{z} f'(z) + \left(1 - \frac{\nu^2}{z^2}\right) f(z) = \frac{1}{\pi z^2} (z + \nu + (z - \nu) \cos(\pi\nu)). \quad (11.6.4)$$

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
```

11.7 Lommel functions

11.7.1 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

$$s_{\mu,\nu} = \frac{z^{\mu+1}}{(\mu - \nu + 1)(\mu + \nu + 1)} {}_1F_2 \left(1; \frac{\mu - \nu + 3}{2}, \frac{\mu + \nu + 3}{2}; -\frac{z^2}{4} \right) \quad (11.7.1)$$

which solves the inhomogeneous Bessel equation

$$z^2 f''(z) + z f'(z) + (z^2 - \nu^2) f(z) = z^{\mu+1}. \quad (11.7.2)$$

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
```

11.7.2 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

$$S_{\mu,\nu}(z) = s_{\mu,\nu}(z) + 2^{\mu-1} \Gamma\left(\frac{1}{2}(\mu - \nu + 1)\right) \Gamma\left(\frac{1}{2}(\mu + \nu + 1)\right) \times \left[\sin\left(\frac{1}{2}(\mu - \nu)\pi\right) J_{\nu}(z) - \cos\left(\frac{1}{2}(\mu - \nu)\pi\right) Y_{\nu}(z) \right] \quad (11.7.3)$$

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
```

11.8 Airy and Scorer functions

11.8.1 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

$$\text{Ai}(0) = \frac{1}{3^{2/3}\Gamma\left(\frac{2}{3}\right)}; \quad \text{Ai}'(0) = \frac{1}{3^{1/3}\Gamma\left(\frac{1}{3}\right)} \quad (11.8.1)$$

Other common ways of defining the Ai-function include integrals such as

$$\text{Ai}(x) = \frac{1}{\pi} \int_0^\infty \cos\left(\frac{1}{3}t^3 + xt\right) dt, \quad x \in \mathbb{R} \quad (11.8.2)$$

$$\text{Ai}(z) = \frac{\sqrt{3}}{2\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} - \frac{z^3}{3t^3}\right) dt. \quad (11.8.3)$$

The Airy function $\text{Ai}(x)$ can also be defined as

$$\text{Ai}(x) = \frac{1}{\pi} \sqrt{\frac{x}{3}} K_{1/3}(z), \quad (x > 0) \quad (11.8.4)$$

$$\text{Ai}(x) = \frac{1}{3^{2/3}\Gamma(2/3)}, \quad (x = 0) \quad (11.8.5)$$

$$\text{Ai}(x) = \frac{1}{2}\sqrt{-x} \left(J_{1/3}(z) - \frac{1}{\sqrt{3}} Y_{1/3}(z) \right), \quad (x < 0) \quad (11.8.6)$$

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 α -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

$$f_0(z) = \text{Ai}(z); \quad f_n(z) = \int_0^z f_{n-1}(t) dt. \quad (11.8.7)$$

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

```

11.8.2 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

$$Bi(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{2}{3}\right)}; \quad Bi'(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{1}{3}\right)} \quad (11.8.8)$$

The Airy function $\text{Bi}(x)$, can also be defined as

$$\text{Bi}(x) = \sqrt{x} \left(\frac{2}{\sqrt{3}} I_{1/3}(z) + \frac{1}{\pi} K_{1/3}(z) \right), \quad (x > 0) \quad (11.8.9)$$

$$\text{Bi}(x) = \frac{1}{3^{1/6} \Gamma(2/3)}, \quad (x = 0) \quad (11.8.10)$$

$$\text{Bi}(x) = -\frac{1}{2} \sqrt{-x} \left(\frac{1}{\sqrt{3}} J_{1/3}(z) + Y_{1/3}(z) \right), \quad (x < 0) \quad (11.8.11)$$

Like the Ai -function (see `airyai()`), the 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 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
```

11.8.3 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
```

11.8.4 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 β_k . Also the conjugate $\bar{\beta}_k$ is a zero.

Optionally, with derivative=1, the corresponding zero b'_k or β'_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
```

11.8.5 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

$$Gi(z) = Ai(z) \int_0^z Bi(t)dt + Bi(z) \int_z^\infty Ai(t)dt \quad (11.8.12)$$

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
```

11.8.6 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

$$Hi(z) = Bi(z) \int_{-\infty}^z Ai(t)dt - Ai(z) \int_{-\infty}^z Bi(t)dt \quad (11.8.13)$$

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
```

11.9 Coulomb wave functions

11.9.1 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

$$F_l(\eta, z) = C_l(\eta) z^{l+1} e^{iz} {}_1F_1(l + 1 - i\eta, 2l + 2, 2iz) \quad (11.9.1)$$

where the normalization constant $C_l(\eta)$ is as calculated by `coulombc()`. This function solves the differential equation

$$f''(z) + \left(1 - \frac{2\eta}{z} - \frac{l(l+1)}{z^2}\right) f(z) = 0. \quad (11.9.2)$$

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

$$f(z) = C_1 F_l(\eta, z) + C_2 G_l(\eta, z) \quad (11.9.3)$$

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, η 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)
```

11.9.2 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

$$G_l(\eta, z) = \frac{F_l(\eta, z) \cos(\chi) - F_{-l-1}(\eta, z)}{\sin(\chi)} \quad (11.9.4)$$

where

$$\chi = \sigma_l - \sigma_{l-1} - (l + 1/2)\pi \quad (11.9.5)$$

and

$$\sigma_l(\eta) = (\ln \Gamma(1 + l + i\eta) - \ln \Gamma(1 + l - i\eta))/(2i) \quad (11.9.6)$$

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
```

11.9.3 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

$$C_l(\eta) = 2^l \exp(-\pi\eta/2 + [\ln \Gamma(1 + l + i\eta) + \ln \Gamma(1 + l - i\eta)]/2 - \ln \Gamma(2l + 2)) \quad (11.9.7)$$

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.

11.10 Parabolic cylinder functions

11.10.1 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

$$y'' + \left(n + \frac{1}{2} - \frac{1}{4}z^2\right)y = 0. \quad (11.10.1)$$

and can be represented in terms of Hermite polynomials (see hermite()) as

$$D_n(z) = 2^{-n/2} e^{-z^2/4} H_n\left(\frac{z}{\sqrt{2}}\right) \quad (11.10.2)$$

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
```

11.10.2 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

$$U(a, z) = 2^{-\frac{1}{4} - \frac{a}{2}} e^{-\frac{1}{4}z^2} U\left(\frac{a}{2} + \frac{1}{4}, \frac{1}{2}, \frac{1}{2}z^2\right) \quad (11.10.3)$$

or, for arbitrary z ,

$$e^{-\frac{1}{4}z^2} U(a, z) = U(a, 0) {}_1F_1\left(-\frac{a}{2} + \frac{1}{4}, \frac{1}{2}, -\frac{1}{2}z^2\right) + U'(a, 0) {}_1F_1\left(-\frac{a}{2} + \frac{3}{4}, \frac{3}{2}, -\frac{1}{2}z^2\right) \quad (11.10.4)$$

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
```

11.10.3 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

$$V(a, z) = \frac{\Gamma(a + \frac{1}{2})(U(a, -z) - \sin(\pi a)U(a, z))}{\pi} \quad (11.10.5)$$

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

```

11.10.4 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 12

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

$$\int_S P_n(x) P_m(x) w(x) = \begin{cases} c_n \neq 0 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases} \quad (12.0.1)$$

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.

12.1 Legendre functions

12.1.1 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

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n \quad (12.1.1)$$

Alternatively, they can be computed recursively using

$$P_0(x) = 1; \quad P_1(x) = x; \quad (n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \quad (12.1.2)$$

A third definition is in terms of the hypergeometric function ${}_2F_1$, whereby they can be generalized to arbitrary n :

$$P_n(x) = {}_2F_1\left(-n, n+1, 1, \frac{1-x}{2}\right) \quad (12.1.3)$$

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]
```

12.1.2 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

$$P_n^m = \frac{1}{\Gamma(1-m)} \frac{(1+z)^{m/2}}{(1-z)^{m/2}} {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right). \quad (12.1.4)$$

With type=3 instead of type=2, the alternative definition

$$\hat{P}_n^m = \frac{1}{\Gamma(1-m)} \frac{(1+z)^{m/2}}{(z-1)^{m/2}} {}_2F_1\left(-n, n+1, 1-m, \frac{1-z}{2}\right). \quad (12.1.5)$$

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

$$(1-z^2)f''(z) - 2zf'(z) + \left(n(n+1) - \frac{m^2}{1-z^2}\right)f(z) = 0 \quad (12.1.6)$$

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
```

12.1.3 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

$$Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} \left(\cos(\pi m) P_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} P_n^{-m}(z) \right) \quad (12.1.7)$$

$$\hat{Q}_n^m(z) = \frac{\pi}{2 \sin(\pi m)} e^{\pi i m} \left(\hat{P}_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} \hat{P}_n^{-m}(z) \right) \quad (12.1.8)$$

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)
```

12.1.4 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 $Y_l^m(\theta, \phi)$,

$$Y_l^m(\theta, \phi) = \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}} P_l^m(\cos(\theta)) e^{im\phi} \quad (12.1.9)$$

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, θ, ϕ 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)
```

12.2 Chebyshev polynomials

12.2.1 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

$$T_n(\cos(x)) = \cos(nx). \quad (12.2.1)$$

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]:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x). \end{aligned} \quad (12.2.2)$$

If $n > 64$ the following trigonometric and hyperbolic identities [1, 22.3.15]:

$$T_n(x) = \cos(n \arccos(x)), \quad |x| < 1 \quad (12.2.3)$$

$$T_n(x) = \cosh(n \operatorname{arccosh}(x)), \quad |x| > 1 \quad (12.2.4)$$

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
```

12.2.2 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

$$U_n(\cos(x)) = \frac{\sin((n+1)x)}{\sin(x)} \quad (12.2.5)$$

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]:

$$\begin{aligned} U_0(x) &= 1 \\ U_1(x) &= 2x \\ U_{n+1}(x) &= 2xU_n(x) - U_{n-1}(x). \end{aligned} \quad (12.2.6)$$

If $n > 64$ the following trigonometric and hyperbolic identities [1, 22.3.15]:

$$U_n(x) = \frac{\sin((n+1) \arccos(x))}{\sin(\arccos(x))}, \quad |x| < 1 \quad (12.2.7)$$

$$U_n(x) = \frac{\sin((n+1) \operatorname{arccosh}(x))}{\sin(\operatorname{arccosh}(x))}, \quad |x| > 1 \quad (12.2.8)$$

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
```

12.3 Jacobi and Gegenbauer polynomials

12.3.1 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

$$P_0^{(a,b)} = 1, \quad 2P_1^{(a,b)} = (a - b) + (a + b + 2)x, \quad (12.3.1)$$

and for $n > 1$ there are the somewhat complicated recurrence relations from [30] (18.9.1) and (18.9.2):

$$\begin{aligned} P_{n+1}^{(a,b)} &= (A_n x + B_n) P_n^{(a,b)} - C_n P_{n+1}^{(a,b)} \\ A_n &= \frac{(2n + a + b + 1)(2n + a + b + 2)}{2(n + 1)(n + a + b + 1)} \\ B_n &= \frac{(a^2 - b^2)(2n + a + b + 1)}{2(n + 1)(n + a + b + 1)(2n + a + b)} \\ C_n &= \frac{(n + a)(n + b)(2n + a + b + 2)}{(n + 1)(n + a + b + 1)(2n + a + b)}. \end{aligned} \quad (12.3.2)$$

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:

$$P_n^{(a,b)} = \binom{n+a}{n} {}_2F_1 \left(-n, 1 + a + b + n, a + 1, \frac{1-x}{2} \right). \quad (12.3.3)$$

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
```

12.3.2 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

$$R_n^m(r) = (-1)^{(n-m)/2} r^m P_{(n-m)/2}^{(m,0)}(1 - 2r^2). \quad (12.3.4)$$

12.3.3 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,

$$C_n^{(a)}(z) = \binom{n+2a-1}{n} {}_2F_1 \left(-n, n+2a; a + \frac{1}{2}, \frac{1}{2}(1-z) \right). \quad (12.3.5)$$

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]:

$$\begin{aligned} C_0^{(a)}(x) &= 1 \\ C_1^{(a)}(x) &= 2ax \\ nC_n^{(a)}(x) &= 2(n+a-1)x C_{n-1}^{(a)}(x) - (n+2a-2) C_{n-2}^{(a)}(x). \end{aligned} \quad (12.3.6)$$

For *a* = 0 the result can be expressed in Chebyshev polynomials:

$$C_0^{(0)}(x) = 1, \quad C_n^{(0)}(x) = 2/n T_n(x). \quad (12.3.7)$$

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)
```

12.4 Hermite and Laguerre polynomials

12.4.1 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

$$H_0(z) = 1; \quad H_1(z) = 2z; \quad H_{n+1} = 2zH_n(z) - 2nH_{n-1}(z). \quad (12.4.1)$$

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]:

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= 2x \\ H_n(x) &= 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x). \end{aligned} \quad (12.4.2)$$

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

$$H_n(z) = (2z)^n {}_2F_0\left(-\frac{n}{2}, \frac{1-n}{2}, -\frac{1}{z^2}\right) \quad (12.4.3)$$

for $\Re z > 0$, or generally

$$H_n(z) = 2^n \sqrt{\pi} \left(\frac{1}{\Gamma(\frac{1-n}{2})} {}_1F_1\left(-\frac{n}{2}, \frac{1}{2}, z^2\right) - \frac{2z}{\Gamma(-\frac{n}{2})} {}_1F_1\left(-\frac{1-n}{2}, \frac{3}{2}, z^2\right) \right) \quad (12.4.4)$$

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)
```

12.4.2 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]

$$L_n^{(a)}(x) = \binom{n+a}{n} M(-n, a+1, x), \quad (12.4.5)$$

otherwise the standard recurrence formulas are used:

$$\begin{aligned} L_0^{(a)}(x) &= 1 \\ L_1^{(a)}(x) &= -x + 1 + a \\ nL_n^{(a)}(x) &= (2n + a - 1 - x)L_{n-1}^{(a)}(x) - (n + a - 1)L_{n-2}^{(a)}(x). \end{aligned} \quad (12.4.6)$$

Gives the generalized (associated) Laguerre polynomial, defined by

$$L_n^{\alpha}(z) = \frac{\Gamma(n + b + 1)}{\Gamma(b + 1)\Gamma(n + 1)} {}_1F_1(-n, a + 1, z). \quad (12.4.7)$$

With $a = 0$ and n a nonnegative integer, this reduces to an ordinary Laguerre polynomial, the sequence of which begins

$$L_0(z) = 1, \quad L_1(z) = 1 - z, \quad L_2(z) = z^2 - 2z + 1, \dots \quad (12.4.8)$$

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
```

12.4.3 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

$$L_n(x) = L_n^{(0)}(x). \quad (12.4.9)$$

12.4.4 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

$$L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x). \quad (12.4.10)$$

The $L_n^m(x)$ are computed using the generalized Laguerre polynomials

$$L_n^m(x) = L_n^{(m)}(x). \quad (12.4.11)$$

Chapter 13

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 \(1997\)](#).

13.1 Confluent Hypergeometric Limit Function

13.1.1 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

$${}_0F_1(a, z) = \sum_{k=0}^{\infty} \frac{1}{(a)_k} \frac{z^k}{k!}. \quad (13.1.1)$$

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

$${}_0F_1(b; x) = {}_0F_1(-; b; x) = \sum_{n=0}^{\infty} \frac{x^n}{(b)_n n!} \quad (13.1.2)$$

where $(a)_k$ is the Pochammer symbol (see section 9.3)

The function is calculated by treating ${}_0F_1(b; 0) = 1$ as special case, and otherwise using the following relations to Bessel functions:

$${}_0F_1(b; x) = \Gamma(b) (+x)^{(1-b)/2} I_{b-1} (2\sqrt{+x}), \quad x > 0, \quad (13.1.3)$$

$${}_0F_1(b; x) = \Gamma(b) (-x)^{(1-b)/2} J_{b-1} (2\sqrt{-x}), \quad x < 0, \quad (13.1.4)$$

`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
```

13.1.2 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]

$${}_0\tilde{F}_1(b; x) = \frac{1}{\Gamma(b)} {}_0F_1(b; x), \quad (b \neq 0, -1, -2, \dots) \quad (13.1.5)$$

and by the corresponding limit if $b = 0, -1, -2, \dots, = -n$, with the value

$${}_0\tilde{F}_1(-n; x) = x^{n+1} {}_0F_1(n+2; x), \quad n \in \mathbb{N} \quad (13.1.6)$$

where $\Gamma(\cdot)$ is the Gamma function (see section 5.8.6).

13.2 Kummer's Confluent Hypergeometric Functions and related functions

13.2.1 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

$${}_1F_1(a, b; z) = \sum_{k=0}^{\infty} \frac{(a)_k}{(b)_k} \cdot \frac{z^k}{k!} \quad (13.2.1)$$

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

$$zf''(z) + (b - z)f'(z) - af(z) = 0. \quad (13.2.2)$$

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

$${}_1F_1(a, b; z) = M(a, b; z) = \sum_{n=0}^{\infty} \frac{(a)_n}{(b)_n} \cdot \frac{z^n}{n!} \quad (13.2.3)$$

where $(a)_k$ is the Pochammer symbol (see section 9.3)

The function has the following integral representation

$${}_1F_1(a, b; z) = B(a, b - a)^{-1} \int_0^1 e^{zt} t^{a-1} (1 - t)^{b-a-1} dt, \quad \Re b > \Re a > 0 \quad (13.2.4)$$

The following closed-form approximation based on a Laplace approximation has been derived by [Butler & Wood \(2002\)](#):

$${}_1F_1(a, b; z) \approx \frac{G_1(a, b; z)}{G_1(a, b; 0)}, \quad \text{where} \quad (13.2.5)$$

$$G_1(a, b, c; z) = w^{-1/2} y^a (1 - y)^{b-a} e^x y,$$

$$w = a(1 - y)^2 + (b - a)y^2$$

$$y = [(x - b) + \sqrt{(x - b)^2 + 4ax}]/2x, \text{ if } x \neq 0, \text{ and } y = a/b \text{ otherwise.}$$

`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
```

13.2.2 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]

$${}_1\tilde{F}_1(a; b; z) = \frac{1}{\Gamma(b)} {}_1F_1(a; b; z) = M(a, b; z) = \frac{1}{\Gamma(b)} M(a; b; z), \quad (b \neq 0, -1, -2, \dots) \quad (13.2.6)$$

and by the corresponding limit if $b = 0, -1, -2, \dots, = -n$, with the value

$${}_1\tilde{F}_1(a; b; z) = \frac{(a)_{n+1}}{(n+1)!} x^{n+1} {}_1F_1(a+n+1; n+2; z), \quad n \in \mathbb{N} \quad (13.2.7)$$

where $\Gamma(\cdot)$ is the Gamma function (see section 5.8.6) and $(a)_k$ is the Pochammer symbol (see section 9.3)

The function has the following integral representation

$${}_1\tilde{F}_1(a, b; -m; z) = \frac{1}{\Gamma(a)\Gamma(b-a)} \int_0^1 e^{zt} t^{a-1} (1-t)^{b-a-1} dt, \quad \Re b > \Re a > 0 \quad (13.2.8)$$

13.2.3 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

$$U(a, b; x) = \frac{\Gamma(1-b)}{\Gamma(1+a-b)} M(a, b; c; z) + \frac{\Gamma(1-b)}{\Gamma(a)} x^{1-b} M(1+a-b, 2-b; x) \quad (13.2.9)$$

where $\Gamma(\cdot)$ is the Gamma function (see section 5.8.6).

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)
```

13.2.4 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

$$_2F_0(a, b; z) = \sum_{n=0}^{\infty} (a)_n (b)_n \frac{z^n}{n!} \quad (13.2.10)$$

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 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
```

13.3 Whittaker functions M and W

13.3.1 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

$$\frac{d^2}{dz^2} + \left(-\frac{1}{4} + \frac{k}{z} + \frac{\frac{1}{4} - m^2}{z^2} \right) f = 0. \quad (13.3.1)$$

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 :

$$M(k, m, z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} {}_1F_1\left(\frac{1}{2} + m - k, 1 + 2m, z\right) \quad (13.3.2)$$

$$W(k, m, z) = e^{-\frac{1}{2}z} z^{\frac{1}{2}+m} U\left(\frac{1}{2} + m - k, 1 + 2m, z\right) \quad (13.3.3)$$

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
```

13.3.2 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
```

13.4 Gauss Hypergeometric Function

13.4.1 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

$${}_2F_1(a, b; c; z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \cdot \frac{z^k}{k!} \quad (13.4.1)$$

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

$${}_2F_1(a, b; c; x) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \cdot \frac{x^k}{k!} \quad (13.4.2)$$

where $(a)_k$ is the Pochhammer symbol (see section 9.3)

The function has the following integral representation

$${}_2F_1(a, b; c; z) = B(a, c - a)^{-1} \int_0^1 \frac{t^{b-1} (1-t)^{c-b-1}}{(1-zt)^a} dt, \quad \Re c > \Re b > 0 \quad (13.4.3)$$

where $B(\cdot, \cdot)$ is the Beta function (see section 5.8.7)

The following closed-form approximation based on a Laplace approximation has been derived by [Butler & Wood \(2002\)](#):

$${}_2F_1(a, b; c; z) \approx \frac{G_2(a, b, c; z)}{G_2(a, b, c; 0)}, \quad \text{where} \quad (13.4.4)$$

$$G_2(a, b, c; z) = w^{-1/2} y^a (1-y)^{c-a} (1-xy)^{-b},$$

$$w = a(1-y)^2 + (c-a)y^2 - bx^2y^2(1-y)^2/(1-xy)^2$$

$$y = [\tau + \sqrt{\tau^2 - 4ax(c-b)}]/[2x(b-c)], \text{ if } x \neq 0, \text{ and } y = a/c \text{ otherwise.}$$

$$\tau = x(b-a) - c.$$

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
```

13.4.2 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]

$${}_2\tilde{F}_1(a, b; c; z) = \frac{1}{\Gamma(c)} {}_2F_1(a, b; c; z) = \mathbf{F}(a, b; c; z), \quad (c \neq 0, -1, -2, \dots) \quad (13.4.5)$$

and by the corresponding limit if $c = 0, -1, -2, \dots, = -m$, with the value

$${}_2\tilde{F}_1(a, b; -m; z) = \frac{(a)_{m+1}(b)_{m+1}}{(m+1)!} x^{m+1} {}_2F_1(a+m+1, a+m+1; m+2; z) \quad (13.4.6)$$

where $\Gamma(\cdot)$ is the Gamma function (see section 5.8.6) and $(a)_k$ is the Pochammer symbol (see section 9.3)

The function has the following integral representation

$${}_2\tilde{F}_1(a, b; c; z) = \frac{1}{\Gamma(b)\Gamma(c-b)} \int_0^1 \frac{t^{b-1}(1-t)^{c-b-1}}{(1-zt)^a} dt, \quad \Re c > \Re b > 0 \quad (13.4.7)$$

13.5 Additional Hypergeometric Functions

13.5.1 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)
```

13.5.2 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)

```

13.5.3 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)

```

13.5.4 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

$${}_3F_2(a_1, a_2, a_3; b_1, b_2; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k (a_3)_k}{(b_1)_k (b_2)_k} \frac{z^k}{k!}, \quad (13.5.1)$$

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
```

13.6 Generalized hypergeometric functions

13.6.1 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

$${}_pF_q(a_1, \dots, a_p; b_1, \dots, b_q; z) = \sum_{n=0}^{\infty} \frac{(a_1)_n (a_2)_n \dots (a_p)_n}{(b_1)_n (b_2)_n \dots (b_q)_n} \frac{z^n}{n!}, \quad (13.6.1)$$

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.

13.6.2 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

$$\sum_{r=1}^N \left[\prod_{k=1}^{l_r} (w_{r,k})^{c_{r,k}} \frac{\prod_{k=1}^{m_r} \Gamma(\alpha_{r,k})}{\prod_{k=1}^{n_r} \Gamma(\beta_{r,k})} {}^{p_r}F_{q_r}(a_{r,1}, \dots, a_{r,p}; b_{r,1}, \dots, b_{r,q}; z_r) \right] \quad (13.6.2)$$

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 α , β , 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

$$(a-1) \frac{\Gamma(a-3)}{\Gamma(a-4)} {}^1F_1(a, a-1, z) = e^z(a-4)(a+z-1) \quad (13.6.3)$$

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
```

13.7 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

$$G_{p,q}^{m,n}\left(z; r \left| \begin{matrix} a_1, \dots, a_n, a_{n+1}, \dots, a_p \\ b_1, \dots, b_m, b_{m+1}, \dots, b_q \end{matrix} \right. \right) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j + s) \prod_{j=1}^n \Gamma(1 - a_j - s)}{\prod_{j=n+1}^p \Gamma(a_j + s) \prod_{j=m+1}^q \Gamma(1 - b_j - s)} z^{-s/r} ds \quad (13.7.1)$$

for an appropriate choice of the contour *L* (see references).

There are *p* elements *a_j*. The argument *a_s* should be a pair of lists, the first containing the *n* elements *a₁, ..., a_n* and the second containing the *p - n* elements *a_{n+1}, ..., a_p*.

There are *q* elements *a_j*. The argument *b_s* should be a pair of lists, the first containing the *m* elements *b₁, ..., b_m* and the second containing the *q - m* elements *b_{m+1}, ..., b_q*.

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:

$$e^z = G_{0,1}^{1,0}\left(-z \left| \begin{matrix} 1 \\ 1 \end{matrix} \right. \right) \quad (13.7.2)$$

```
>>> meijerg([],[], [[0],[]], -z)
9.487735836358525720550369
>>> exp(z)
9.487735836358525720550369
```

The natural logarithm

$$\log(1+z) = G_{2,2}^{1,2} \left(-z \left| \begin{matrix} a_1, a_2 \\ b_1, b_2 \end{matrix} \right. \right) \quad (13.7.3)$$

```
>>> meijerg([[1,1],[]], [[1],[0]], z)
1.178654996341646117219023
>>> log(1+z)
1.178654996341646117219023
```

A rational function

$$\frac{z}{z+1} = G_{2,2}^{1,2} \left(-z \left| \begin{matrix} a_1, a_2 \\ b_1, b_2 \end{matrix} \right. \right) \quad (13.7.4)$$

```
>>> meijerg([[1,1],[]], [[1],[1]], z)
0.6923076923076923076923077
>>> z/(z+1)
0.6923076923076923076923077
```

The sine and cosine functions:

$$\frac{1}{\sqrt{\pi}} \sin(2\sqrt{z}) = G_{0,2}^{1,0} \left(z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (13.7.5)$$

$$\frac{1}{\sqrt{\pi}} \cos(2\sqrt{z}) = G_{0,2}^{1,0} \left(z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (13.7.6)$$

```
>>> 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:

$$J_\alpha(2\sqrt{z}) = G_{0,2}^{1,0} \left(z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (13.7.7)$$

$$Y_\alpha(2\sqrt{z}) = G_{1,3}^{2,0} \left(z \left| \begin{matrix} a_1 \\ b_1, b_2, b_3 \end{matrix} \right. \right) \quad (13.7.8)$$

$$(-z)^{\alpha/2} z^{-\alpha/2} I_\alpha(2\sqrt{z}) = G_{0,2}^{2,0} \left(-z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (13.7.9)$$

$$2K_\alpha(2\sqrt{z}) = G_{0,2}^{2,0} \left(z \left| \begin{matrix} b_1, b_2 \end{matrix} \right. \right) \quad (13.7.10)$$

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:

$$\sqrt{\pi} z^{2(\alpha-1)} \operatorname{erfc}(z) = G_{1,2}^{2,0} \left(z, \frac{1}{2} \middle| \begin{matrix} a_1 \\ b_1, b_2 \end{matrix} \right) \quad (13.7.11)$$

```

>>> 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

```

13.8 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

$${}_A H_B(a_1; \dots, a_A; b_1, \dots, b_B) = \sum_{n=-\infty}^{\infty} \frac{(a_1)_n \dots (a_A)_n}{(b_1)_n \dots (b_B)_n} z^n \quad (13.8.1)$$

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)
```

13.9 Hypergeometric functions of two variables

13.9.1 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

$$\sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{P((a), m, n)}{Q((b))} \frac{x^m y^n}{m!, n!} \quad (13.9.1)$$

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

$$F_1 = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b)_m (c)_n}{(d)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (13.9.2)$$

$$F_4 = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b)_{m+n}}{(d)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (13.9.3)$$

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)
2.666666666666666666666666666667
>>> 1/(1-x)/(1-y)
2.666666666666666666666666666667
>>> 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
```

13.9.2 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,

$$F_1(a_1, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b_1)_m (b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (13.9.4)$$

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)
```

13.9.3 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

$$F_2(a_1, b_1, b_2, c_1, c_2, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (b_1)_m (b_2)_n}{(c_1)_m (c_2)_n} \frac{x^m y^n}{m! n!} \quad (13.9.5)$$

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)
```

13.9.4 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

$$F_3(a_1, a_2, b_1, b_2, c, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a_1)_m (a_2)_n (b_1)_m (b_2)_n}{(c)_{m+n}} \frac{x^m y^n}{m!, n!} \quad (13.9.6)$$

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
```

13.9.5 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

$$F_4(a, b, c_1, c_2, x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{(a)_{m+n} (a_2)_n (b)_{m+n}}{(c_1)_m (c_2)_n} \frac{x^m y^n}{m!, n!} \quad (13.9.7)$$

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 14

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

$$f(z + 2\omega_1) = f(z + 2\omega_2) = f(z) \quad (14.0.1)$$

for some half-periods ω_1, ω_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 τ , which usually must satisfy $\Im[\tau] > 0$. These quantities can be expressed in terms of each other using the following relations:

$$m = k^2; \quad \tau = i \frac{K(1-m)}{K(m)}; \quad q = e^{i\pi\tau}; \quad k = \frac{\vartheta_2^4(q)}{\vartheta_2^4(q)} \quad (14.0.2)$$

In addition, an alternative definition is used for the nome in number theory, which we here denote by \bar{q} :

$$\bar{q} = q^2 = e^{2i\pi\tau} \quad (14.0.3)$$

14.1 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, τ, \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, τ :

```
>>> 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, τ, \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, τ, \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 τ .

Parameter:

Keywords: m=x; k=x; q=x; qbar=x.

Returns the elliptic half-period ratio τ , 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)
```

14.2 Legendre elliptic integrals

14.2.1 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

$$K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \frac{\pi}{2} {}_2F_1\left(\frac{1}{2}, \frac{1}{2}, 1, m\right) \quad (14.2.1)$$

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)
```

14.2.2 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

$$F(\phi, m) = \int_0^{\phi} \frac{dt}{\sqrt{1 - m \sin^2 t}} \quad (14.2.2)$$

or equivalently

$$F(\phi, m) = \int_0^{\sin \phi} \frac{dt}{\sqrt{1-t^2}\sqrt{1-mt^2}} \quad (14.2.3)$$

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

$$F(\phi + n\pi, m) = 2nK(m) + F(\phi, m), n \in \mathbb{Z}. \quad (14.2.4)$$

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)
```

14.2.3 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

$$E(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \frac{\pi}{2} {}_2F_1\left(\frac{1}{2}, -\frac{1}{2}, 1, m\right) \quad (14.2.5)$$

```
>>> 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)
```

14.2.4 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

$$E(\phi, m) = \int_0^{\phi} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}} dt \quad (14.2.6)$$

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

$$E(\phi + n\pi, m) = 2nE(m) + F(\phi, m), n \in \mathbb{Z}. \quad (14.2.7)$$

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)
```

14.2.5 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
```

14.2.6 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

$$\Pi(n; \phi, m) = \int_0^\phi \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{dt}{(1 - nt^2) \sqrt{1 - t^2} \sqrt{1 - mt^2}} \quad (14.2.8)$$

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

$$\Pi(n, \phi + k\pi, m) = 2k\Pi(n, m) + \Pi(n, \phi, m), k \in \mathbb{Z}. \quad (14.2.9)$$

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)
```

14.3 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 \(1995\)](#).

14.3.1 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

$$R_F(x, y, z) = \frac{1}{2} \int_0^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}} \quad (14.3.1)$$

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)
```

14.3.2 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

$$R_C(x, y) = R_F(x, y, y) = \frac{1}{2} \int_0^\infty \frac{dt}{(t+y)\sqrt{(t+x)}} \quad (14.3.2)$$

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

$$R_C(x, y) = \begin{cases} \frac{1}{\sqrt{y-x}} \cos^{-1} \left(\sqrt{x/y} \right), & x < y \\ \frac{1}{\sqrt{y}}, & x = y \\ \frac{1}{\sqrt{y-x}} \cosh^{-1} \left(\sqrt{x/y} \right), & x > y \end{cases} \quad (14.3.3)$$

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)
```

14.3.3 Symmetric elliptic integral of the third kind, RJ

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

$$R_J(x, y, z, p) = \frac{3}{2} \int_0^{\infty} \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}} \quad (14.3.4)$$

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)
```

14.3.4 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
```

14.3.5 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

$$R_G(x, y, z) = \frac{1}{4} \int_0^\infty \frac{t}{\sqrt{(t+x)(t+y)(t+z)}} \left(\frac{x}{t+x} + \frac{y}{t+y} + \frac{z}{t+z} \right) dt \quad (14.3.5)$$

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
```

14.4 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:

$$\vartheta_1(z, q) = 2q^{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin(2n+1)z \quad (14.4.1)$$

$$\vartheta_2(z, q) = 2q^{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos(2n+1)z \quad (14.4.2)$$

$$\vartheta_3(z, q) = 1 + 2 \sum_{n=0}^{\infty} q^{n^2} \cos(2nz) \quad (14.4.3)$$

$$\vartheta_4(z, q) = 1 + 2 \sum_{n=0}^{\infty} (-1)^n q^{n^2} \cos(2nz) \quad (14.4.4)$$

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 τ 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)
```

14.5 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

$$x = \int_0^{\text{sn}} \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}}, \quad \text{sn}^2 + \text{cn}^2 = 1, \quad k^2\text{sn}^2 + \text{cn}^2 = 1 \quad (14.5.1)$$

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

$$\text{sn}(u, m) = \frac{\vartheta_3(0, q)\vartheta_1(t, q)}{\vartheta_2(0, q)\vartheta_4(t, q)} \quad (14.5.2)$$

$$\text{cn}(u, m) = \frac{\vartheta_4(0, q)\vartheta_2(t, q)}{\vartheta_2(0, q)\vartheta_4(t, q)} \quad (14.5.3)$$

$$\text{dn}(u, m) = \frac{\vartheta_4(0, q)\vartheta_3(t, q)}{\vartheta_3(0, q)\vartheta_4(t, q)} \quad (14.5.4)$$

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

$$\text{cd}(u, m) = \frac{1}{\text{sn}(u, m)} \quad (14.5.5)$$

and 'cd' identifies the function

$$\text{ns}(u, m) = \frac{\text{cn}(u, m)}{\text{dn}(u, m)} \quad (14.5.6)$$

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
```

14.6 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 τ in the upper half-plane as

$$J(\tau) = \frac{g_2^3(\tau)}{g_2^3(\tau) - 27g_3^2(\tau)} \quad (14.6.1)$$

where g_2 and g_3 are the modular invariants of the Weierstrass elliptic function,

$$g_2(\tau) = 60 \sum_{(m,n) \in \mathbb{Z}^2 \setminus (0,0)} (m\tau + n)^{-4} \quad (14.6.2)$$

$$g_3(\tau) = 140 \sum_{(m,n) \in \mathbb{Z}^2 \setminus (0,0)} (m\tau + n)^{-6} \quad (14.6.3)$$

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 15

Zeta functions, L-series and polylogarithms

This section includes the Riemann zeta functions and associated functions pertaining to analytic number theory.

15.1 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.

$$\zeta(s) = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \dots \quad (15.1.1)$$

or, with $a \neq 1$, the more general Hurwitz zeta function

$$\zeta(s, a) = \sum_{k=0}^{\infty} \frac{1}{(a+k)^s}. \quad (15.1.2)$$

Optionally, **zeta**(*s*, *a*, *n*) computes the *n*-th derivative with respect to *s*,

$$\zeta^{(n)}(s, a) = (-1)^n \sum_{k=0}^{\infty} \frac{\log^n(a+k)}{(a+k)^s}. \quad (15.1.3)$$

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
```

15.2 Dirichlet L-series

15.2.1 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

$$\eta(s) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k^s} = 1 - \frac{1}{2^s} + \frac{1}{3^s} - \frac{1}{4^s} + \dots \quad (15.2.1)$$

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
```

15.2.2 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 $\eta(s) - 1$.

15.2.3 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

$$\beta(s) = \sum_{n=1}^{\infty} \frac{(-1)^n}{(2n+1)^s} = 2^{-s} \Phi\left(-1, s, \frac{1}{2}\right) \quad (15.2.2)$$

15.2.4 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

$$\lambda(s) = \sum_{n=0}^{\infty} (2n+1)^s \quad (15.2.3)$$

and by analytic continuation for $s < 1$. The function is calculated as

$$\lambda(s) = (1 - 2^{-s})\eta(s). \quad (15.2.4)$$

15.2.5 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

$$L(s, \chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s} \quad (15.2.5)$$

where χ 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, χ 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
```

15.3 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 γ_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:

$$\zeta(s) = \frac{1}{s-1} \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n (s-1)^n \quad (15.3.1)$$

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]:

$$\gamma_n(a) = \frac{\log^n a}{2a} \frac{\log^{n+1}(a)}{n+1} + \frac{2}{a} \Re \int_0^\infty \frac{(x/a - i) \log^n(a - ix)}{(1 + x^2/a^2)(e^{2\pi x} - 1)} dx \quad (15.3.2)$$

Examples

The zeroth Stieltjes constant is just Euler's constant γ :

```
>>> 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
```

15.4 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)
```

15.5 Riemann-Siegel Z function and related functions

15.5.1 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

$$Z(t) = e^{i\theta(t)} \zeta(1/2 + it) \quad (15.5.1)$$

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
```

15.5.2 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

$$\theta(t) = \frac{\log \Gamma(\frac{1+2it}{4}) - \log \Gamma(\frac{1-2it}{4})}{2i} - \frac{\log \pi}{2} t. \quad (15.5.2)$$

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
```

15.5.3 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
```

15.5.4 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
```

15.6 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

$$\Phi(z, s, a) = \sum_{k=0}^{\infty} \frac{z^k}{(a+k)^s} \quad (15.6.1)$$

and generally by the recurrence $\Phi(z, s, a) = z\Phi(z, s, a+1) + a^{-s}$ along with the integral representation valid for $\Re a > 0$

$$\Phi(z, s, a) = \frac{1}{2a^s} + \int_0^{\infty} \frac{z^t}{(a+t)^s} dt - 2 \int_0^{\infty} \frac{\sin(t \log z) - s \arctan(t/a)}{(a^2 + t^2)^{s/2} (e^{2\pi t} - 1)} dt. \quad (15.6.2)$$

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)
```

15.6.1 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

$$F_s(x) = \frac{1}{\Gamma(s+1)} \int_0^\infty \frac{t^s}{e^{t-x} + 1} dt \quad (15.6.3)$$

and by analytic continuation for $s \leq -1$ using polylogarithms

$$F_s(x) = -\text{Li}_{s+1}(-e^x) = e^x \Phi(-e^x, s+1, 1). \quad (15.6.4)$$

15.6.2 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.

15.6.3 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.

15.6.4 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.

15.6.5 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

$$\chi_s(x) = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)^s}. \quad (15.6.5)$$

The function can be expressed as

$$\chi_s(x) = 2^{-s} x \Phi\left(x^2, s, \frac{1}{2}\right) = \frac{1}{2} (\text{Li}_s(x) - \text{Li}_s(-x)). \quad (15.6.6)$$

For large $s > 22.8$ the function adds up to three terms of the sum, for $s = 0$ or $s = 1$ the *Li*_{*s*} relation is used, otherwise the result is computed with Lerch's transcendent.

15.6.6 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

$$\text{Ti}_2(x) = \int_0^x \frac{\arctan(t)}{t} dt. = \frac{1}{4} x \Phi\left(-x^2, 2, \frac{1}{2}\right) \quad (15.6.7)$$

For $x > 1$ the relation

$$\text{Ti}_2(x) = \text{Ti}_2\left(\frac{1}{x}\right) + \frac{\pi}{2} \ln(x) \quad (15.6.8)$$

is used, and for $x < 0$ the result is $\text{Ti}_2(x) = -\text{Ti}_2(-x)$.

15.7 Polylogarithms and Clausen functions

15.7.1 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

$$\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}. \quad (15.7.1)$$

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)
```

15.7.2 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

$$\text{dilog}(x) = \Re \text{Li}_2(x) = -\Re \int_0^x \frac{\ln(1-t)}{t} dt. \quad (15.7.2)$$

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.

15.7.3 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

$$D_n(x) = \frac{n}{x^n} \int_0^x \frac{t^n}{e^t - 1} dt \quad (n > 0, x \geq 0). \quad (15.7.3)$$

$$D_k(x) = \frac{k}{x^{k+1}} \left[(-1)^k k! \zeta(k+1) + \sum_{m=0}^k (-1)^{k-m+1} \frac{k!}{m!} x^m \text{Li}_{k-m+1}(e^x) \right] - \frac{k}{k+1}, \quad (15.7.4)$$

where Li_s denotes the polylogarithm (see [Dubinov & Dubinova \(2008\)](#)).

15.7.4 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

$$\text{Cl}_s(z) = \sum_{k=1}^{\infty} \frac{\sin(kz)}{k^s}. \quad (15.7.5)$$

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

$$\text{Cl}_s(z) = \frac{1}{2i} [\text{Li}_s(e^{iz}) - \text{Li}_s(e^{-iz})] \quad (15.7.6)$$

$$\text{Cl}_s(z) = \Im [\text{Li}_s(e^{iz})], \quad (s, z \in \mathbb{R}), \quad (15.7.7)$$

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
```

15.7.5 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

$$\widetilde{\text{Cl}}_s(z) = \sum_{k=1}^{\infty} \frac{\cos(kz)}{k^s}. \quad (15.7.8)$$

This function is complementary to the Clausen sine function $\text{clsin}()$. In terms of the polylogarithm,

$$\widetilde{\text{Cl}}_s(z) = \frac{1}{2} [\text{Li}_s(e^{iz}) - \text{Li}_s(e^{-iz})] \quad (15.7.9)$$

$$\widetilde{\text{Cl}}_s(z) = \Re [\text{Li}_s(e^{iz})], \quad (s, z \in \mathbb{R}), \quad (15.7.10)$$

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
```

15.7.6 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

$$E_s(z) = \sum_{k=1}^{\infty} \frac{k^s}{k!} z^k. \quad (15.7.11)$$

$E_s(z)$ 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,

$$E_s(z) = e^z B_s(z) - \text{sinc}(\pi s). \quad (15.7.12)$$

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)
```

15.8 Zeta function variants

15.8.1 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

$$P(s) = \sum_p \frac{1}{p^s} \quad (15.8.1)$$

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)
```

15.8.2 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

$$Z(s) = \sum_{n=1}^{\infty} \frac{1}{\tau_n^s} \quad (15.8.2)$$

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 16

Number-theoretical, combinatorial and integer functions

For factorial-type functions, including binomial coefficients, double factorials, etc., see the separate section Factorials and gamma functions.

16.1 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

$$F(z) = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}} \quad (16.1.1)$$

where ϕ 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)
```

16.2 Bernoulli numbers and polynomials

16.2.1 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

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}, \quad |t| < 2\pi. \quad (16.2.1)$$

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]

$$(-1)^{n+1} B_{2n} \approx \frac{2(2n)!}{(2\pi)^{2n}}, \quad (16.2.2)$$

gives an asymptotic recursion formula

$$B_{2n+2} \approx -\frac{(2n+1)(2n+2)}{(2\pi)^2} B_{2n}, \quad (16.2.3)$$

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
```

16.2.2 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]

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!}, \quad |t| < 2\pi. \quad (16.2.4)$$

or the simple explicit representation [30, 24.2.5]

$$B_n(x) = \sum_{n=0}^{\infty} \binom{n}{k} B_k(x) x^{n-k}. \quad (16.2.5)$$

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
```

16.3 Euler numbers and polynomials

16.3.1 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

$$\text{sech}(t) = \sum_{n=0}^{\infty} \frac{E_n}{n!} t^n. \quad (16.3.1)$$

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
```

16.3.2 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

$$\frac{2e^{zt}}{e^t + 1} = \sum_{n=0}^{\infty} E_n(z) \frac{t^n}{n!}. \quad (16.3.2)$$

The Euler polynomials may also be represented in terms of Bernoulli polynomials (see `bernpoly()`) using various formulas, for example

$$En(z) = \frac{2}{n+1} \left(B_n(z) - 2^{n+2} B_n(z/2) \right) \quad (16.3.3)$$

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
```

16.4 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

$$B_0(x) = 1; \quad B_1(x) = x; \quad B_2(x) = x^2 + x; \quad B_3(x) = x^3 + 3x^2 + x. \quad (16.4.1)$$

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

$$B_n(x) = e^{-x} (\text{sinc}(\pi n) + E_n(x)) \quad (16.4.2)$$

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)
```

16.5 Stirling numbers

16.5.1 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

$$x(x-1)(x-2)\cdots(x-n+1) = \sum_{k=0}^n s(n, k)x^k. \quad (16.5.1)$$

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
```

16.5.2 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

$$x^n = \sum_{k=0}^n S(n, k)x(x-1)(x-2)\cdots(x-k+1). \quad (16.5.2)$$

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
```

16.6 Prime counting functions

16.6.1 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
```

16.6.2 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

$$|\pi(x)| - \text{li}(x) < \frac{\sqrt{x} \log x}{8\pi} \quad (16.6.1)$$

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
```

16.6.3 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,

$$R(x) = 1 + \sum_{k=1}^{\infty} \frac{\log^k x}{kk!\zeta(k+1)}. \quad (16.6.2)$$

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
```

16.7 Miscellaneous functions

16.7.1 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

$$\Phi_n(x) = \prod_{\zeta}(x - \zeta) \quad (16.7.1)$$

where ζ ranges over all primitive n -th roots of unity (see `unitroots()`). An equivalent representation, used for computation, is

$$\Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)} \quad (16.7.2)$$

where $\mu(m)$ denotes the Moebius function. The cyclotomic polynomials are integer polynomials, the first of which can be written explicitly as

$$\Phi_0(x) = 1; \quad \Phi_1(x) = x - 1; \quad \Phi_2(x) = x + 1; \quad \Phi_3(x) = x^3 + x^2 + 1 \quad (16.7.3)$$

$$\Phi_4(x) = x^2 + 1; \quad \Phi_5(x) = x^4 + x^3 + x^2 + x + 1; \quad \Phi_6(x) = x^2 - x + 1. \quad (16.7.4)$$

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]
```

16.7.2 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 17

q-functions

17.1 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

$$(a; q)_n = \prod_{k=0}^{n-1} (1 - aq^k) \quad (17.1.1)$$

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

$$\phi(q) = (q; q)_\infty = \prod_{k=1}^{\infty} (1 - q^k) = \sum_{k=-\infty}^{\infty} (-1)^k q^{(3k^2-k)/2} \quad (17.1.2)$$

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)
```

17.2 q-gamma and factorial

17.2.1 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

$$\Gamma_q(z) = \frac{(q; q)_\infty}{(q^z; q)_\infty} (1 - q)^{1-z}. \quad (17.2.1)$$

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)
```

17.2.2 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

$$[n]_q! = (1 + q)(1 + q + q^2) \cdots (1 + q + \cdots + q^{n-1}) \quad (17.2.2)$$

or more generally

$$[z]_q! = \frac{(q; q)_z}{(1 - q)^z} \quad (17.2.3)$$

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)
```

17.3 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

$$\text{qhyper}(A, B, q, z) = \sum_{n=0}^{\infty} \frac{(a_1; q)_n, \dots, (a_r; q)_n}{(b_1; q)_n, \dots, (b_s; q)_n} \left((-1)^n q^{\binom{n}{2}} \right)^{1+s-r} \frac{z^n}{(q; q)_n} \quad (17.3.1)$$

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 18

Matrix functions

18.1 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

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots \quad (18.1.1)$$

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
```

18.2 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)]
```

18.3 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)]
```

18.4 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
```

18.5 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]
```

18.6 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 19

Eigensystems and related Decompositions

19.1 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

$$A = USV; \quad U'U = 1; \quad VV' = 1, \quad (19.1.1)$$

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]
```

19.2 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

$$Q' A Q = R; \quad Q' Q = Q Q' = 1, \quad (19.2.1)$$

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]
```

19.3 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]
```

19.4 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*:

$$Q' A Q = \text{diag}(E); \quad Q' Q = Q' Q = 1. \quad (19.4.1)$$

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

Numerical Calculus

Chapter 20

Polynomials

See also `taylor()` and `chebyfit()` for approximation of functions by polynomials.

20.1 Polynomial evaluation

Function **`polyval(coeffs As mpNum, x As mpNum, Keywords As String) As mpNum`**

The function `polyval` returns a polynomial

Parameters:

coeffs: A list of coefficients (real or complex numbers).

x: A real or complex number.

Keywords: `derivative=False`.

Given coefficients $[c_n, \dots, c_2, c_1, c_0]$ and a number x , `polyval()` evaluates the polynomial

$$P(x) = c_n x^n + \dots + c_2 x^2 + c_1 x + c_0 \quad (20.1.1)$$

If `derivative=True` is set, `polyval()` simultaneously evaluates $P(x)$ with the derivative, $P'(x)$, and returns the tuple $(P(x), P'(x))$.

```
>>> from mpFormulaPy import *
>>> mp.pretty = True
>>> polyval([3, 0, 2], 0.5)
2.75
>>> polyval([3, 0, 2], 0.5, derivative=True)
(2.75, 3.0)
```

The coefficients and the evaluation point may be any combination of real or complex numbers.

20.2 Polynomial roots

Function **polyroots(coeffs As mpNum, Keywords As String) As mpNum**

The function `polyroots` returns all roots (real or complex) of a given polynomial.

Parameters:

coeffs: A real or complex number.

Keywords: `maxsteps=50`, `cleanup=True`, `extraprec=10`, `error=False`.

The roots are returned as a sorted list, where real roots appear first followed by complex conjugate roots as adjacent elements. The polynomial should be given as a list of coefficients, in the format used by `polyval()`. The leading coefficient must be nonzero.

With `error=True`, `polyroots()` returns a tuple `(roots, err)` where `err` is an estimate of the maximum error among the computed roots.

Examples

Finding the three real roots of $x^3 - x^2 - 14x + 24$:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(polyroots([1,-1,-14,24]), 4)
[-4.0, 2.0, 3.0]
```

Finding the two complex conjugate roots of $4x^2 + 3x + 2$, with an error estimate:

```
>>> roots, err = polyroots([4,3,2], error=True)
>>> for r in roots:
...     print(r)
...
(-0.375 + 0.59947894041409j)
(-0.375 - 0.59947894041409j)
>>>
>>> err
2.22044604925031e-16
>>>
>>> polyval([4,3,2], roots[0])
(2.22044604925031e-16 + 0.0j)
>>> polyval([4,3,2], roots[1])
(2.22044604925031e-16 + 0.0j)
```

The following example computes all the 5th roots of unity; that is, the roots of $x^5 - 1$:

```
>>> mp.dps = 20
>>> for r in polyroots([1, 0, 0, 0, 0, -1]):
...     print(r)
...
1.0
(-0.8090169943749474241 + 0.58778525229247312917j)
(-0.8090169943749474241 - 0.58778525229247312917j)
(0.3090169943749474241 + 0.95105651629515357212j)
(0.3090169943749474241 - 0.95105651629515357212j)
```

Precision and conditioning

The roots are computed to the current working precision accuracy. If this accuracy cannot be achieved in *maxsteps* steps, then a *NoConvergence* exception is raised.

The algorithm internally is using the current working precision extended by *extraprec*. If *NoConvergence* was raised, that is caused either by not having enough extra precision to achieve convergence (in which case increasing *extraprec* should fix the problem) or too low *maxsteps* (in which case increasing *maxsteps* should fix the problem), or a combination of both.

The user should always do a convergence study with regards to *extraprec* to ensure accurate results. It is possible to get convergence to a wrong answer with too low *extraprec*.

Provided there are no repeated roots, *polyroots()* can typically compute all roots of an arbitrary polynomial to high precision:

```
>>> mp.dps = 60
>>> for r in polyroots([1, 0, -10, 0, 1]):
...     print r
...
-3.14626436994197234232913506571557044551247712918732870123249
-0.317837245195782244725757617296174288373133378433432554879127
0.317837245195782244725757617296174288373133378433432554879127
3.14626436994197234232913506571557044551247712918732870123249
>>>
>>> sqrt(3) + sqrt(2)
3.14626436994197234232913506571557044551247712918732870123249
>>> sqrt(3) - sqrt(2)
0.317837245195782244725757617296174288373133378433432554879127
```

Algorithm

polyroots() implements the Durand-Kerner method [1], which uses complex arithmetic to locate all roots simultaneously. The Durand-Kerner method can be viewed as approximately performing simultaneous Newton iteration for all the roots. In particular, the convergence to simple roots is quadratic, just like Newton's method.

Although all roots are internally calculated using complex arithmetic, any root found to have an imaginary part smaller than the estimated numerical error is truncated to a real number (small real parts are also chopped). Real roots are placed first in the returned list, sorted by value. The remaining complex roots are sorted by their real parts so that conjugate roots end up next to each other.

Chapter 21

Root-finding and optimization

21.1 Root-finding

Function **findroot**(*f* As *mpNum*, *x0* As *mpNum*, **Keywords** As *String*) As *mpNum*

The function `findroot` returns a solution to $f(x) = 0$, using *x0* as starting point or interval for *x*.

Parameters:

f: A one dimensional function

x0: A real or complex number.

Keywords: `solver=Secant`, `tol=None`, `verbose=False`, `verify=True`. Many more, see below.

Multidimensional overdetermined systems are supported. You can specify them using a function or a list of functions.

If the found root does not satisfy $|f(x)|^2 < tol$, an exception is raised (this can be disabled with `verify=False`).

Arguments

f: one dimensional function

x0: starting point, several starting points or interval (depends on solver)

tol: the returned solution has an error smaller than this

verbose: print additional information for each iteration if true

verify: verify the solution and raise a `ValueError` if

solver: a generator for *f* and *x0* returning approximative solution and error

maxsteps: after how many steps the solver will cancel

df: first derivative of *f* (used by some solvers)

d2f: second derivative of *f* (used by some solvers)

multidimensional: force multidimensional solving

J: Jacobian matrix of *f* (used by multidimensional solvers)

norm: used vector norm (used by multidimensional solvers)

solver has to be callable with $(f, x_0, \text{**kwargs})$ and return an generator yielding pairs of approximative solution and estimated error (which is expected to be positive).

You can use the following string aliases: 'secant', 'mnewton', 'halley', 'muller', 'illinois', 'pegasus', 'anderson', 'ridder', 'anewton', 'bisect'. See `mpFormulaPy.calculus.optimization` for their documentation.

Examples

The function `findroot()` locates a root of a given function using the secant method by default. A simple example use of the secant method is to compute π as the root of $\sin x$ closest to $x_0 = 3$:

```
>>> from mpFormulaPy import *
>>> mp.dps = 30; mp.pretty = True
>>> findroot(sin, 3)
3.14159265358979323846264338328
```

The secant method can be used to find complex roots of analytic functions, although it must in that case generally be given a nonreal starting value (or else it will never leave the real line):

```
>>> mp.dps = 15
>>> findroot(lambda x: x**3 + 2*x + 1, j)
(0.226698825758202 + 1.46771150871022j)
```

A nice application is to compute nontrivial roots of the Riemann zeta function with many digits (good initial values are needed for convergence):

```
>>> mp.dps = 30
>>> findroot(zeta, 0.5+14j)
(0.5 + 14.1347251417346937904572519836j)
```

The secant method can also be used as an optimization algorithm, by passing it a derivative of a function. The following example locates the positive minimum of the gamma function:

```
>>> mp.dps = 20
>>> findroot(lambda x: diff(gamma, x), 1)
1.4616321449683623413
```

Finally, a useful application is to compute inverse functions, such as the Lambert W function which is the inverse of we^w , given the first term of the solution's asymptotic expansion as the initial value. In basic cases, this gives identical results to `mpFormulaPy`'s built-in `lambertw` function:

```
>>> def lambert(x):
...     return findroot(lambda w: w*exp(w) - x, log(1+x))
...
>>> mp.dps = 15
>>> lambert(1); lambertw(1)
0.567143290409784
0.567143290409784
>>> lambert(1000); lambert(1000)
5.2496028524016
5.2496028524016
```

Multidimensional functions are also supported:

```
>>> f = [lambda x1, x2: x1**2 + x2,
... lambda x1, x2: 5*x1**2 - 3*x1 + 2*x2 - 3]
>>> findroot(f, (0, 0))
[-0.618033988749895]
[-0.381966011250105]
>>> findroot(f, (10, 10))
[ 1.61803398874989]
[-2.61803398874989]
```

You can verify this by solving the system manually.

Please note that the following (more general) syntax also works:

```
>>> def f(x1, x2):
...     return x1**2 + x2, 5*x1**2 - 3*x1 + 2*x2 - 3
...
>>> findroot(f, (0, 0))
[-0.618033988749895]
[-0.381966011250105]
```

Multiple roots

For multiple roots all methods of the Newtonian family (including secant) converge slowly. Consider this example:

```
>>> f = lambda x: (x - 1)**99
>>> findroot(f, 0.9, verify=False)
0.918073542444929
```

Even for a very close starting point the secant method converges very slowly. Use `verbose=True` to illustrate this.

It is possible to modify Newton's method to make it converge regardless of the root's multiplicity:

```
>>> findroot(f, -10, solver="mnewton")
1.0
```

This variant uses the first and second derivative of the function, which is not very efficient.

Alternatively you can use an experimental Newtonian solver that keeps track of the speed of convergence and accelerates it using Steffensen's method if necessary:

```
>>> findroot(f, -10, solver="anewton", verbose=True)
x: -9.888888888888888889
error: 0.11111111111111111111
converging slowly
x: -9.77890011223344556678
error: 0.10998877665544332211
converging slowly
x: -9.67002233332199662166
error: 0.10887778911448945119
converging slowly
accelerating convergence
x: -9.5622443299551077669
error: 0.107778003366888854764
```

```

converging slowly
x: 0.999999999999999214
error: 10.562244329955107759
x: 1.0
error: 7.8598304758094664213e-18
ZeroDivisionError: canceled with x = 1.0
1.0

```

Complex roots

For complex roots it is recommended to use Muller's method as it converges very quickly even for real starting points :

```

>>> findroot(lambda x: x**4 + x + 1, (0, 1, 2), solver="muller")
(0.727136084491197 + 0.934099289460529j)

```

Intersection methods

When you need to find a root in a known interval, it is highly recommended to use an intersection-based solver like 'anderson' or 'ridder'. Usually they converge faster and more reliable. They have however problems with multiple roots and usually need a sign change to find a root:

```

>>> findroot(lambda x: x**3, (-1, 1), solver="anderson")
0.0

```

Be careful with symmetric functions:

```

>>> findroot(lambda x: x**2, (-1, 1), solver="anderson")
Traceback (most recent call last):
...
ZeroDivisionError

```

It fails even for better starting points, because there is no sign change:

```

>>> findroot(lambda x: x**2, (-1, .5), solver='anderson')
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (1 > 2.1684e-19)
Try another starting point or tweak arguments.

```

21.2 Solvers

21.2.1 Secant

1d-solver generating pairs of approximative root and error. Needs starting points x_0 and x_1 close to the root. x_1 defaults to $x_0 + 0.25$.

Pro: converges quickly

Contra: converges slowly for multiple roots

21.2.2 Newton

1d-solver generating pairs of approximative root and error. Needs starting points x_0 close to the root.

Pro: converges fast. Sometimes more robust than secant with bad second starting point

Contra: converges slowly for multiple roots. Needs first derivative. 2 function evaluations per iteration

21.2.3 MNewton

1d-solver generating pairs of approximative root and error. Needs starting point x_0 close to the root. Uses modified Newton's method that converges fast regardless of the multiplicity of the root.

Pro: converges quickly for multiple roots.

Contra: needs first and second derivative of f . 3 function evaluations per iteration

21.2.4 Halley

1d-solver generating pairs of approximative root and error. Needs a starting point x_0 close to the root. Uses Halley's method with cubic convergence rate.

Pro: converges even faster than Newton's method. Useful when computing with many digits

Contra: needs first and second derivative of f . 3 function evaluations per iteration. Converges slowly for multiple roots

21.2.5 Muller

1d-solver generating pairs of approximative root and error. Needs starting points x_0 , x_1 and x_2 close to the root. x_1 defaults to $x_0 + 0.25$; x_2 to $x_1 + 0.25$. Uses Muller's method that converges towards complex roots.

Pro: converges quickly (somewhat faster than secant). Can find complex roots

Contra: converges slowly for multiple roots. May have complex values for real starting points and real roots.

21.2.6 Bisection

1d-solver generating pairs of approximative root and error. Uses bisection method to find a root of f in $[a, b]$. Might fail for multiple roots (needs sign change).

Pro: robust and reliable

Contra: converges slowly. Needs sign change

21.2.7 Illinois

1d-solver generating pairs of approximative root and error.

Uses Illinois method or similar to find a root of f in $[a, b]$. Might fail for multiple roots (needs sign change). Combines bisect with secant (improved regula falsi).

The only difference between the methods is the scaling factor m , which is used to ensure convergence (you can choose one using the 'method' keyword):

Illinois method ('illinois'): $m = 0.5$

Pegasus method ('pegasus'): $m = fb/(fb + fz)$

Anderson-Bjoerk method ('anderson'): $m = 1 - fz/fb$ if positive else 0.5

Pro: converges very fast

Contra: has problems with multiple roots. Needs sign change

21.2.8 Pegasus

1d-solver generating pairs of approximative root and error.

Uses Pegasus method to find a root of f in $[a, b]$.

Wrapper for illinois to use method='pegasus'.

21.2.9 Anderson

1d-solver generating pairs of approximative root and error.

Uses Anderson-Bjoerk method to find a root of f in $[a, b]$.

Wrapper for illinois to use method='anderson'.

21.2.10 Ridder

1d-solver generating pairs of approximative root and error. Ridder's method to find a root of f in $[a, b]$. Is told to perform as well as Brent's method while being simpler.

Pro: very fast. Simpler than Brent's method

Contra: two function evaluations per step. Has problems with multiple roots. Needs sign change

21.2.11 ANewton

EXPERIMENTAL 1d-solver generating pairs of approximative root and error.

Uses Newton's method modified to use Steffensen's method when convergence is slow (i.e. for multiple roots.)

21.2.12 MDNewton

Find the root of a vector function numerically using Newton's method.

f is a vector function representing a nonlinear equation system.

x_0 is the starting point close to the root.

J is a function returning the Jacobian matrix for a point.

Supports overdetermined systems.

Use the 'norm' keyword to specify which norm to use. Defaults to max-norm. The function to calculate the Jacobian matrix can be given using the keyword 'J'. Otherwise it will be calculated numerically.

Please note that this method converges only locally. Especially for high- dimensional systems it is not trivial to find a good starting point being close enough to the root.

It is recommended to use a faster, low-precision solver from SciPy [1] or OpenOpt [2] to get an initial guess. Afterwards you can use this method for root-polishing to any precision.

Chapter 22

Sums, products, limits and extrapolation

The functions listed here permit approximation of infinite sums, products, and other sequence limits. Use `mpFormulaPy.fsum()` and `mpFormulaPy.fprod()` for summation and multiplication of finite sequences.

22.1 Summation

22.1.1 One-dimensional Summation

Function **`nsum(f As mpNum, interval As mpNum, Keywords As String) As mpNum`**

The function `nsum` returns a one dimensional (possibly infinite) sum.

Parameters:

f: A one dimensional function

interval: A real interval.

Keywords: `method=r+s`, `tol=eps`, `verbose=False`, `maxterms=10*dps`. Many more, see below.

This function computes the sum

$$S = \sum_{k=a}^b f(k) \tag{22.1.1}$$

where $(a, b) = \text{interval}$, and where $a = -\infty$ and/or $b = \infty$ are allowed, or more generally

$$S = \sum_{k_1=a_1}^{b_1} \cdots \sum_{k_n=a_n}^{b_n} f(k_1, \dots, k_n) \tag{22.1.2}$$

if multiple intervals are given.

Two examples of infinite series that can be summed by `nsum()`, where the first converges rapidly and the second converges slowly, are:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> nsum(lambda n: 1/fac(n), [0, inf])
2.71828182845905
>>> nsum(lambda n: 1/n**2, [1, inf])
```

1.64493406684823

When appropriate, `nsum()` applies convergence acceleration to accurately estimate the sums of slowly convergent series. If the series is finite, `nsum()` currently does not attempt to perform any extrapolation, and simply calls `fsum()`.

Multidimensional infinite series are reduced to a single-dimensional series over expanding hypercubes; if both infinite and finite dimensions are present, the finite ranges are moved innermost. For more advanced control over the summation order, use nested calls to `nsum()`, or manually rewrite the sum as a single-dimensional series.

Options

tol: Desired maximum final error. Defaults roughly to the epsilon of the working precision.

method: Which summation algorithm to use (described below). Default: 'richardson+shanks'.

maxterms: Cancel after at most this many terms. Default: 10^*dps .

steps: An iterable giving the number of terms to add between each extrapolation attempt. The default sequence is [10, 20, 30, 40, ...]. For example, if you know that approximately 100 terms will be required, efficiency might be improved by setting this to [100, 10]. Then the first extrapolation will be performed after 100 terms, the second after 110, etc.

verbose: Print details about progress.

ignore: If enabled, any term that raises `ArithmeticError` or `ValueError` (e.g. through division by zero) is replaced by a zero. This is convenient for lattice sums with a singular term near the origin.

Methods

Unfortunately, an algorithm that can efficiently sum any infinite series does not exist. `nsum()` implements several different algorithms that each work well in different cases.

The `method` keyword argument selects a method.

The default method is 'r+s', i.e. both Richardson extrapolation and Shanks transformation is attempted. A slower method that handles more cases is 'r+s+e'. For very high precision summation, or if the summation needs to be fast (for example if multiple sums need to be evaluated), it is a good idea to investigate which one method works best and only use that.

'richardson' / 'r':

Uses Richardson extrapolation. Provides useful extrapolation when $f(k) \sim P(k)/Q(k)$ or when $f(k) \sim (-1)^k P(k)/Q(k)$ for polynomials P and Q .

See `richardson()` for additional information.

'shanks' / 's':

Uses Shanks transformation. Typically provides useful extrapolation when $f(k) \sim c^k$ or when successive terms alternate signs. Is able to sum some divergent series. See `shanks()` for additional information.

'levin' / 'l':

Uses the Levin transformation. It performs better than the Shanks transformation for logarithmic convergent or alternating divergent series. The 'levin_variant'-keyword selects the variant. Valid choices are "u", "t", "v" and "all" whereby "all" uses all three u,t and v simultaneously (This is good for performance comparison in conjunction with "verbose=True"). Instead of the Levin

transform one can also use the Sidi-S transform by selecting the method 'sidi'. See `levin()` for additional details.

'alternating' / 'a':

This is the convergence acceleration of alternating series developed by Cohen, Villegas and Zagier. See `cohen_alt()` for additional details.

'euler-maclaurin' / 'e':

Uses the Euler-Maclaurin summation formula to approximate the remainder sum by an integral. This requires high-order numerical derivatives and numerical integration. The advantage of this algorithm is that it works regardless of the decay rate of f , as long as f is sufficiently smooth. See `sumem()` for additional information.

'direct' / 'd':

Does not perform any extrapolation. This can be used (and should only be used for) rapidly convergent series. The summation automatically stops when the terms decrease below the target tolerance.

Basic examples

A finite sum:

```
>>> nsum(lambda k: 1/k, [1, 6])
2.45
```

Summation of a series going to negative infinity and a doubly infinite series:

```
>>> nsum(lambda k: 1/k**2, [-inf, -1])
1.64493406684823
>>> nsum(lambda k: 1/(1+k**2), [-inf, inf])
3.15334809493716
```

`nsum()` handles sums of complex numbers:

```
>>> nsum(lambda k: (0.5+0.25j)**k, [0, inf])
(1.6 + 0.8j)
```

The following sum converges very rapidly, so it is most efficient to sum it by disabling convergence acceleration:

```
>>> mp.dps = 1000
>>> a = nsum(lambda k: -(-1)**k * k**2 / fac(2*k), [1, inf], method="direct")
>>> b = (cos(1)+sin(1))/4
>>> abs(a-b) < mpf('1e-998')
True
```

Examples with Richardson extrapolation

Richardson extrapolation works well for sums over rational functions, as well as their alternating counterparts:

```
>>> mp.dps = 50
>>> nsum(lambda k: 1 / k**3, [1, inf], method="richardson")
1.2020569031595942853997381615114499907649862923405
>>> zeta(3)
```

```

1.2020569031595942853997381615114499907649862923405
>>> nsum(lambda n: (n + 3)/(n**3 + n**2), [1, inf], method="richardson")
2.9348022005446793094172454999380755676568497036204
>>> pi**2/2-2
2.9348022005446793094172454999380755676568497036204
>>> nsum(lambda k: (-1)**k / k**3, [1, inf], method="richardson")
-0.90154267736969571404980362113358749307373971925537
>>> -3*zeta(3)/4
-0.90154267736969571404980362113358749307373971925538

```

Examples with Shanks extrapolation

The Shanks transformation works well for geometric series and typically provides excellent acceleration for Taylor series near the border of their disk of convergence. Here we apply it to a series for $\log(2)$, which can be seen as the Taylor series for $\log(1 + x)$ with $x = 1$:

```

>>> nsum(lambda k: -(-1)**k/k, [1, inf], method="shanks")
0.69314718055994530941723212145817656807550013436025
>>> log(2)
0.69314718055994530941723212145817656807550013436025

```

Here we apply it to a slowly convergent geometric series:

```

>>> nsum(lambda k: mpf("0.995")**k, [0, inf], method="shanks")
200.0

```

Finally, Shanks' method works very well for alternating series where $f(k) = (-1)^k g(k)$, and often does so regardless of the exact decay rate of $g(k)$:

```

>>> mp.dps = 15
>>> nsum(lambda k: (-1)**(k+1) / k**1.5, [1, inf], method="shanks")
0.765147024625408
>>> (2-sqrt(2))*zeta(1.5)/2
0.765147024625408

```

The following slowly convergent alternating series has no known closed-form value. Evaluating the sum a second time at higher precision indicates that the value is probably correct:

```

>>> nsum(lambda k: (-1)**k / log(k), [2, inf], method="shanks")
0.924299897222939
>>> mp.dps = 30
>>> nsum(lambda k: (-1)**k / log(k), [2, inf], method="shanks")
0.924299897222938855595957018136

```

Examples with Levin transformation

The following example calculates EulerâŽs constant as the constant term in the Laurent expansion of $\zeta(s)$ at $s=1$. This sum converges extremely slow because of the logarithmic convergence behaviour of the Dirichlet series for ζ .

```

>>> mp.dps = 30
>>> z = mp.mpf(10) ** (-10)
>>> a = mp.nsum(lambda n: n**(-(1+z)), [1, mp.inf], method = "levin") - 1 / z
>>> print(mp.chop(a - mp.euler, tol = 1e-10))

```

0.0

Now we sum the zeta function outside its range of convergence (attention: This does not work at the negative integers!):

```
>>> mp.dps = 15
>>> w = mp.nsum(lambda n: n ** (2 + 3j), [1, mp.inf], method = "levin", levin_variant = "v")
>>> print(mp.chop(w - mp.zeta(-2-3j)))
0.0
```

The next example resummates an asymptotic series expansion of an integral related to the exponential integral.

```
>>> mp.dps = 15
>>> z = mp.mpf(10)
>>> # exact = mp.quad(lambda x: mp.exp(-x)/(1+x/z), [0,mp.inf])
>>> exact = z * mp.exp(z) * mp.expint(1,z) # this is the symbolic expression for the integral
>>> w = mp.nsum(lambda n: (-1) ** n * mp.fac(n) * z ** (-n), [0, mp.inf], method = "sidi", levin_variant = "t")
>>> print(mp.chop(w - exact))
0.0
```

Following highly divergent asymptotic expansion needs some care. Firstly we need copious amount of working precision. Secondly the stepsize must not be chosen too large, otherwise nsum may miss the point where the Levin transform converges and reach the point where only numerical garbage is produced due to numerical cancellation.

```
>>> mp.dps = 15
>>> z = mp.mpf(2)
>>> # exact = mp.quad(lambda x: mp.exp( -x * x / 2 - z * x ** 4), [0,mp.inf]) * 2 / mp.sqrt(2 * mp.pi)
>>> exact = mp.exp(mp.one / (32 * z)) * mp.besselk(mp.one / 4, mp.one / (32 * z)) / (4 * mp.sqrt(z * mp.pi)) # this is the symbolic expression for the integral
>>> w = mp.nsum(lambda n: (-z)**n * mp.fac(4 * n) / (mp.fac(n) * mp.fac(2 * n) * (4 ** n)),
...   [0, mp.inf], method = "levin", levin_variant = "t", workprec = 8*mp.prec, steps = [2] + [1 for x in xrange(1000)])
>>> print(mp.chop(w - exact))
0.0
```

The hypergeometric function can also be summed outside its range of convergence:

```
>>> mp.dps = 15
>>> z = 2 + 1j
>>> exact = mp.hyp2f1(2 / mp.mpf(3), 4 / mp.mpf(3), 1 / mp.mpf(3), z)
>>> f = lambda n: mp.rf(2 / mp.mpf(3), n) * mp.rf(4 / mp.mpf(3), n) * z**n / (mp.rf(1 / mp.mpf(3), n) * mp.fac(n))
>>> v = mp.nsum(f, [0, mp.inf], method = "levin", steps = [10 for x in xrange(1000)])
>>> print(mp.chop(exact-v))
0.0
```

Examples with Cohen's alternating series resummation

The next example sums the alternating zeta function:

```
>>> v = mp.nsum(lambda n: (-1)**(n-1) / n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.log(2)))
0.0
```

The derivate of the alternating zeta function outside its range of convergence:

```
>>> v = mp.nsum(lambda n: (-1)**n * mp.log(n) * n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.diff(lambda s: mp.alzteta(s), -1)))
0.0
```

Examples with Euler-Maclaurin summation

The sum in the following example has the wrong rate of convergence for either Richardson or Shanks to be effective.

```
>>> f = lambda k: log(k)/k**2.5
>>> mp.dps = 15
>>> nsum(f, [1, inf], method="euler-maclaurin")
0.38734195032621
>>> -diff(zeta, 2.5)
0.38734195032621
```

Increasing steps improves speed at higher precision:

```
>>> f = lambda k: log(k)/k**2.5
>>> mp.dps = 50
>>> nsum(f, [1, inf], method="euler-maclaurin", steps=[250])
0.38734195032620997271199237593105101319948228874688
>>> -diff(zeta, 2.5)
0.38734195032620997271199237593105101319948228874688
```

Divergent series

The Shanks transformation is able to sum some divergent series. In particular, it is often able to sum Taylor series beyond their radius of convergence (this is due to a relation between the Shanks transformation and Pade approximations; see `pade()` for an alternative way to evaluate divergent Taylor series). Furthermore the Levintransform examples above contain some divergent series resummation.

Here we apply it to $\log(1 + x)$ far outside the region of convergence:

```
>>> mp.dps = 50
>>> nsum(lambda k: -(-9)**k/k, [1, inf], method="shanks")
2.3025850929940456840179914546843642076011014886288
>>> log(10)
2.3025850929940456840179914546843642076011014886288
```

A particular type of divergent series that can be summed using the Shanks transformation is geometric series. The result is the same as using the closed-form formula for an infinite geometric series:

```
>>> mp.dps = 15
```

```

>>> for n in range(-8, 8):
... if n == 1:
... continue
... print("%s %s %s" % (mpf(n), mpf(1)/(1-n),
... nsum(lambda k: n**k, [0, inf], method="shanks")))
...
-8.0 0.1111111111111111 0.1111111111111111
-7.0 0.125 0.125
-6.0 0.142857142857143 0.142857142857143
-5.0 0.1666666666666667 0.1666666666666667
-4.0 0.2 0.2
-3.0 0.25 0.25
-2.0 0.3333333333333333 0.3333333333333333
-1.0 0.5 0.5
0.0 1.0 1.0
2.0 -1.0 -1.0
3.0 -0.5 -0.5
4.0 -0.3333333333333333 -0.3333333333333333
5.0 -0.25 -0.25
6.0 -0.2 -0.2
7.0 -0.1666666666666667 -0.1666666666666667

```

22.1.2 Two-dimensional Summation

Function **nsum2d**(*f* As *mpNum*, *interval1* As *mpNum*, *interval2* As *mpNum*, *Keywords* As *String*) As *mpNum*

The function **nsum2d** returns a two dimensional (possibly infinite) sum.

Parameters:

f: A one dimensional function

interval1: A real interval.

interval2: A real interval.

Keywords: method=r+s, tol=eps, verbose=False, maxterms=10*dps. Many more, see below.

Any combination of finite and infinite ranges is allowed for the summation indices:

```

>>> mp.dps = 15
>>> nsum2d(lambda x,y: x+y, [2,3], [4,5])
28.0
>>> nsum2d(lambda x,y: x/2**y, [1,3], [1,inf])
6.0
>>> nsum2d(lambda x,y: y/2**x, [1,inf], [1,3])
6.0
>>> nsum3d(lambda x,y,z: z/(2**x*2**y), [1,inf], [1,inf], [3,4])
7.0
>>> nsum3d(lambda x,y,z: y/(2**x*2**z), [1,inf], [3,4], [1,inf])
7.0
>>> nsum3d(lambda x,y,z: x/(2**z*2**y), [3,4], [1,inf], [1,inf])
7.0

```

Some nice examples of double series with analytic solutions or reductions to single-dimensional series (see [1]):

```

>>> nsum2d(lambda m, n: 1/2** (m*n), [1,inf], [1,inf])
1.60669515241529
>>> nsum(lambda n: 1/(2**n-1), [1,inf])
1.60669515241529
>>> nsum2d(lambda i,j: (-1)**(i+j)/(i**2+j**2), [1,inf], [1,inf])
0.278070510848213
>>> pi*(pi-3*ln2)/12
0.278070510848213
>>> nsum2d(lambda i,j: (-1)**(i+j)/(i+j)**2, [1,inf], [1,inf])
0.129319852864168
>>> altzeta(2) - altzeta(1)
0.129319852864168
>>> nsum2d(lambda i,j: (-1)**(i+j)/(i+j)**3, [1,inf], [1,inf])
0.0790756439455825
>>> altzeta(3) - altzeta(2)
0.0790756439455825
>>> nsum2d(lambda m,n: m**2*n/(3**m*(n*3**m+m*3**n)), [1,inf], [1,inf])
0.28125
>>> mpf(9)/32
0.28125
>>> nsum2d(lambda i,j: fac(i-1)*fac(j-1)/fac(i+j), [1,inf], [1,inf], workprec=400)
1.64493406684823
>>> zeta(2)
1.64493406684823

```

22.1.3 Three-dimensional Summation

Function **nsum3d**(*f* As *mpNum*, **interval1** As *mpNum*, **interval2** As *mpNum*, **interval3** As *mpNum*, **Keywords** As String) As *mpNum*

The function **nsum3d** returns a three dimensional (possibly infinite) sum .

Parameters:

f: A one dimensional function

interval1: A real interval.

interval2: A real interval.

interval3: A real interval.

Keywords: method=r+s, tol=eps, verbose=False, maxterms=10*dps. Many more, see below.

A hard example of a multidimensional sum is the Madelung constant in three dimensions (see [2]). The defining sum converges very slowly and only conditionally, so **nsum()** is lucky to obtain an accurate value through convergence acceleration. The second evaluation below uses a much more efficient, rapidly convergent 2D sum:

```

>>> nsum3d(lambda x,y,z: (-1)**(x+y+z)/(x*x+y*y+z*z)**0.5,
... [-inf,inf], [-inf,inf], [-inf,inf], ignore=True)
-1.74756459463318
>>> nsum2d(lambda x,y: -12*pi*sech(0.5*pi * \

```

```
... sqrt((2*x+1)**2+(2*y+1)**2))**2, [0,inf], [0,inf])
-1.74756459463318
```

Another example of a lattice sum in 2D:

```
>>> nsum(lambda x,y: (-1)**(x+y) / (x**2+y**2), [-inf,inf],
... [-inf,inf], ignore=True)
-2.1775860903036
>>> -pi*ln2
-2.1775860903036
```

An example of an Eisenstein series:

```
>>> nsum(lambda m,n: (m+n*1j)**(-4), [-inf,inf], [-inf,inf],
... ignore=True)
(3.1512120021539 + 0.0j)
```

22.1.4 Euler-Maclaurin formula

Function **sumem**(*f* As *mpNum*, *interval* As *mpNum*, *Keywords* As *String*) As *mpNum*

The function **sumem** returns an infinite series of an analytic summand *f* using the Euler-Maclaurin formula

Parameters:

f: A one dimensional function

interval: A real interval.

Keywords: tol=None, reject=10, integral=None, adiffs=None, bdiffs=None, verbose=False, error=False, fastabort=False

Uses the Euler-Maclaurin formula to compute an approximation accurate to within tol (which defaults to the present epsilon) of the sum

$$S = \sum_{k=a}^b f(k) \quad (22.1.3)$$

where (a, b) are given by interval and or may be infinite. The approximation is

$$S \sim \int_a^b f(x)dx + \frac{f(a) + f(b)}{2} + \sum_{k=1}^{\infty} \frac{B_{2k}}{(2k)!} (f^{(2k-1)}(b) - f^{(2k-1)}(a)) \quad (22.1.4)$$

The last sum in the Euler-Maclaurin formula is not generally convergent (a notable exception is if *f* is a polynomial, in which case Euler-Maclaurin actually gives an exact result).

The summation is stopped as soon as the quotient between two consecutive terms falls below reject. That is, by default (reject = 10), the summation is continued as long as each term adds at least one decimal.

Although not convergent, convergence to a given tolerance can often be "forced" if $b = \infty$ by summing up to $a + N$ and then applying the Euler-Maclaurin formula to the sum over the range $(a + N + 1, \dots, \infty)$. This procedure is implemented by **nsum()**.

By default numerical quadrature and differentiation is used. If the symbolic values of the integral and endpoint derivatives are known, it is more efficient to pass the value of the integral explicitly as integral and the derivatives explicitly as adiffs and bdiffs. The derivatives should be given as iterables that yield $f(a), f'(a), f''(a), \dots$ (and the equivalent for b).

Examples

Summation of an infinite series, with automatic and symbolic integral and derivative values (the second should be much faster):

```
>>> from mpFormulaPy import *
>>> mp.dps = 50; mp.pretty = True
>>> sumem(lambda n: 1/n**2, [32, inf])
0.03174336652030209012658168043874142714132886413417
>>> I = mpf(1)/32
>>> D = adiffs=(-1)**n*fac(n+1)*32**(-2-n) for n in range(999)
>>> sumem(lambda n: 1/n**2, [32, inf], integral=I, adiffs=D)
0.03174336652030209012658168043874142714132886413417
```

An exact evaluation of a finite polynomial sum:

```
>>> sumem(lambda n: n**5-12*n**2+3*n, [-100000, 200000])
10500155000624963999742499550000.0
>>> print(sum(n**5-12*n**2+3*n for n in range(-100000, 200001)))
10500155000624963999742499550000
```

22.1.5 Abel-Plana formula

Function **sumap**(*f* As *mpNum*, *interval* As *mpNum*, *Keywords* As *String*) As *mpNum*

The function **sumap** returns an infinite series of an analytic summand *f* using the Abel-Plana formula.

Parameters:

f: A one dimensional function

interval: A real interval.

Keywords: integral=None, error=False

Evaluates an infinite series of an analytic summand *f* using the Abel-Plana formula

$$\sum_{k=0}^{\infty} f(k) = \int_0^{\infty} f(t)dt + \frac{1}{2}f(0) + i \int_0^{\infty} \frac{f(it) - f(-it)}{e^{2\pi t} - 1} dt \quad (22.1.5)$$

Unlike the Euler-Maclaurin formula (see **sumem()**), the Abel-Plana formula does not require derivatives. However, it only works when $|f(it) - f(-it)|$ does not increase too rapidly with *t*.

Examples

The Abel-Plana formula is particularly useful when the summand decreases like a power of *k*; for example when the sum is a pure zeta function:

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> sumap(lambda k: 1/k**2.5, [1,inf])
1.34148725725091717975677
```

```
>>> zeta(2.5)
1.34148725725091717975677
>>> sumap(lambda k: 1/(k+1j)**(2.5+2.5j), [1,inf])
(-3.385361068546473342286084 - 0.7432082105196321803869551j)
>>> zeta(2.5+2.5j, 1+1j)
(-3.385361068546473342286084 - 0.7432082105196321803869551j)
```

If the series is alternating, numerical quadrature along the real line is likely to give poor results, so it is better to evaluate the first term symbolically whenever possible:

```
>>> n=3; z=-0.75
>>> I = expint(n,-log(z))
>>> chop(sumap(lambda k: z**k / k**n, [1,inf], integral=I))
-0.6917036036904594510141448
>>> polylog(n,z)
-0.6917036036904594510141448
```

22.2 Products

Function **nprod**(*f* As *mpNum*, *interval* As *mpNum*, *Keywords* As *String*) As *mpNum*

The function **nprod** returns a one dimensional (possibly infinite) product.

Parameters:

f: A one dimensional function

interval: A real interval.

Keywords: nsum=False.

Computes the product

$$P = \prod_{k=a}^b f(k) \quad (22.2.1)$$

where $= (a, b)$ interval, and where $a = -\infty$ and/or $b = \infty$ are allowed.

By default, **nprod()** uses the same extrapolation methods as **nsum()**, except applied to the partial products rather than partial sums, and the same keyword options as for **nsum()** are supported. If *nsum=True*, the product is instead computed via **nsum()** as

$$P = \exp \left(\sum_{k=a}^b \log (f(k)) \right). \quad (22.2.2)$$

This is slower, but can sometimes yield better results. It is also required (and used automatically) when Euler-Maclaurin summation is requested.

Examples

A simple finite product:

```
>>> from mpFormulaPy import *
>>> mp.dps = 25; mp.pretty = True
>>> nprod(lambda k: k, [1, 4])
24.0
```

A large number of infinite products have known exact values, and can therefore be used as a reference. Most of the following examples are taken from MathWorld [1].

A few infinite products with simple values are:

```
>>> 2*nprod(lambda k: (4*k**2)/(4*k**2-1), [1, inf])
3.141592653589793238462643
>>> nprod(lambda k: (1+1/k)**2/(1+2/k), [1, inf])
2.0
>>> nprod(lambda k: (k**3-1)/(k**3+1), [2, inf])
0.66666666666666666666666666666667
>>> nprod(lambda k: (1-1/k**2), [2, inf])
0.5
```

Next, several more infinite products with more complicated values:

```
>>> nprod(lambda k: exp(1/k**2), [1, inf]); exp(pi**2/6)
5.180668317897115748416626
5.180668317897115748416626
>>> nprod(lambda k: (k**2-1)/(k**2+1), [2, inf]); pi*csch(pi)
```

```

0.2720290549821331629502366
0.2720290549821331629502366
>>> nprod(lambda k: (k**4-1)/(k**4+1), [2, inf])
0.8480540493529003921296502
>>> pi*sinh(pi)/(cosh(sqrt(2)*pi)-cos(sqrt(2)*pi))
0.8480540493529003921296502
>>> nprod(lambda k: (1+1/k+1/k**2)**2/(1+2/k+3/k**2), [1, inf])
1.848936182858244485224927
>>> 3*sqrt(2)*cosh(pi*sqrt(3)/2)**2*csch(pi*sqrt(2))/pi
1.848936182858244485224927
>>> nprod(lambda k: (1-1/k**4), [2, inf]); sinh(pi)/(4*pi)
0.9190194775937444301739244
0.9190194775937444301739244
>>> nprod(lambda k: (1-1/k**6), [2, inf])
0.9826842777421925183244759
>>> (1+cosh(pi*sqrt(3)))/(12*pi**2)
0.9826842777421925183244759
>>> nprod(lambda k: (1+1/k**2), [2, inf]); sinh(pi)/(2*pi)
1.838038955187488860347849
1.838038955187488860347849
>>> nprod(lambda n: (1+1/n)**n * exp(1/(2*n)-1), [1, inf])
1.447255926890365298959138
>>> exp(1+euler/2)/sqrt(2*pi)
1.447255926890365298959138

```

The following two products are equivalent and can be evaluated in terms of a Jacobi theta function. Pi can be replaced by any value (as long as convergence is preserved):

```

>>> nprod(lambda k: (1-pi**-k)/(1+pi**-k), [1, inf])
0.3838451207481672404778686
>>> nprod(lambda k: tanh(k*log(pi)/2), [1, inf])
0.3838451207481672404778686
>>> jtheta(4,0,1/pi)
0.3838451207481672404778686

```

This product does not have a known closed form value:

```

>>> nprod(lambda k: (1-1/2**k), [1, inf])
0.2887880950866024212788997

```

A product taken from $-\infty$:

```

>>> nprod(lambda k: 1-k**(-3), [-inf,-2])
0.8093965973662901095786805
>>> cosh(pi*sqrt(3)/2)/(3*pi)
0.8093965973662901095786805

```

A doubly infinite product:

```

>>> nprod(lambda k: exp(1/(1+k**2)), [-inf, inf])
23.41432688231864337420035
>>> exp(pi/tanh(pi))
23.41432688231864337420035

```

A product requiring the use of Euler-Maclaurin summation to compute an accurate value:

```
>>> nprod(lambda k: (1-1/k**2.5), [2, inf], method="e")
0.696155111336231052898125
```

22.3 Limits

Function **limit**(*f* As mpNum, **interval** As mpNum, **Keywords** As String) As mpNum

The function `limit` returns an estimate of the limit.

Parameters:

f. A one dimensional function

interval: A real interval.

Keywords: direction=1, exp=False.

Computes an estimate of the limit

$$\lim_{t \rightarrow x} f(t) \quad (22.3.1)$$

where x may be finite or infinite.

For finite x , `limit()` evaluates $f(x + d/n)$ for consecutive integer values of n , where the approach direction d may be specified using the `direction` keyword argument. For infinite x , `limit()` evaluates values of $f(\text{sign}(x) \cdot n)$.

If the approach to the limit is not sufficiently fast to give an accurate estimate directly, `limit()` attempts to find the limit using Richardson extrapolation or the Shanks transformation. You can select between these methods using the `method` keyword (see documentation of `nsum()` for more information).

Options

The following options are available with essentially the same meaning as for `nsum()`: `tol`, `method`, `maxterms`, `steps`, `verbose`.

If the option `exp=True` is set, f will be sampled at exponentially spaced points $n = 2^1, 2^2, 2^3, \dots$ instead of the linearly spaced points $1, 2, 3, \dots$. This can sometimes improve the rate of convergence so that `limit()` may return a more accurate answer (and faster). However, do note that this can only be used if f supports fast and accurate evaluation for arguments that are extremely close to the limit point (or if infinite, very large arguments).

Examples

A basic evaluation of a removable singularity:

```
>>> from mpFormulaPy import *
>>> mp.dps = 30; mp.pretty = True
>>> limit(lambda x: (x-sin(x))/x**3, 0)
0.1666666666666666666666666667
```

Computing the exponential function using its limit definition:

```
>>> limit(lambda n: (1+3/n)**n, inf)
20.0855369231876677409285296546
>>> exp(3)
20.0855369231876677409285296546
```

A limit for π :

```
>>> f = lambda n: 2**4*n+1*fac(n)**4/(2*n+1)/fac(2*n)**2
>>> limit(f, inf)
3.14159265358979323846264338328
```

Calculating the coefficient in Stirling's formula:

```
>>> limit(lambda n: fac(n) / (sqrt(n)*(n/e)**n), inf)
2.50662827463100050241576528481
>>> sqrt(2*pi)
2.50662827463100050241576528481
```

Evaluating Euler's constant γ using the limit representation

$$\gamma = \lim_{n \rightarrow \infty} \left[\left(\sum_{k=1}^n \frac{1}{k} \right) - \log(n) \right] \quad (22.3.2)$$

which converges notoriously slowly):

```
>>> f = lambda n: sum([mpf(1)/k for k in range(1,int(n)+1)]) - log(n)
>>> limit(f, inf)
0.577215664901532860606512090082
>>> +euler
0.577215664901532860606512090082
```

With default settings, the following limit converges too slowly to be evaluated accurately. Changing to exponential sampling however gives a perfect result:

```
>>> f = lambda x: sqrt(x**3+x**2)/(sqrt(x**3)+x)
>>> limit(f, inf)
0.992831158558330281129249686491
>>> limit(f, inf, exp=True)
1.0
```

22.4 Extrapolation

The following functions provide a direct interface to extrapolation algorithms. `nsum()` and `limit()` essentially work by calling the following functions with an increasing number of terms until the extrapolated limit is accurate enough.

The following functions may be useful to call directly if the precise number of terms needed to achieve a desired accuracy is known in advance, or if one wishes to study the convergence properties of the algorithms.

22.4.1 Richardson's algorithm

Function **richardson(*seq* As *mpNum*) As *mpNum***

The function `richardson` returns the N -term Richardson extrapolate for the limit.

Parameter:

seq: a list of the first N elements of a slowly convergent infinite sequence.

Given a list *seq* of the first N elements of a slowly convergent infinite sequence, `richardson()` computes the N -term Richardson extrapolate for the limit.

`richardson()` returns (v, c) where v is the estimated limit and c is the magnitude of the largest weight used during the computation. The weight provides an estimate of the precision lost to cancellation. Due to cancellation effects, the sequence must be typically be computed at a much higher precision than the target accuracy of the extrapolation.

Applicability and issues

The N -step Richardson extrapolation algorithm used by `richardson()` is described in [1].

Richardson extrapolation only works for a specific type of sequence, namely one converging like partial sums of $P(1)/Q(1) + P(2)/Q(2) + \dots$ where P and Q are polynomials. When the sequence does not converge at such a rate `richardson()` generally produces garbage.

Richardson extrapolation has the advantage of being fast: the N -term extrapolate requires only $O(N)$ arithmetic operations, and usually produces an estimate that is accurate to $O(N)$ digits. Contrast with the Shanks transformation (see `shanks()`), which requires $O(N^2)$ operations.

`richardson()` is unable to produce an estimate for the approximation error. One way to estimate the error is to perform two extrapolations with slightly different N and comparing the results. Richardson extrapolation does not work for oscillating sequences. As a simple workaround, `richardson()` detects if the last three elements do not differ monotonically, and in that case applies extrapolation only to the even-index elements.

Example

Applying Richardson extrapolation to the Leibniz series for π :

```
>>> from mpFormulaPy import *
>>> mp.dps = 30; mp.pretty = True
>>> S = [4*sum(mpf(-1)**n/(2*n+1) for n in range(m))
... for m in range(1,30)]
>>> v, c = richardson(S[:10])
>>> v
3.2126984126984126984126984127
>>> nprint([v-pi, c])
```

```
[0.0711058, 2.0]
>>> v, c = richardson(S[:30])
>>> v
3.14159265468624052829954206226
>>> npprint([v-pi, c])
[1.09645e-9, 20833.3]
```

22.4.2 Shanks' algorithm

Function **shanks**(*seq* As *mpNum*, *Keywords* As *String*) As *mpNum*

The function **shanks** returns the N -term Shanks extrapolate for the limit.

Parameters:

seq: a list of the first N elements of a slowly convergent infinite sequence.

Keywords: table=None, randomized=False.

Given a list *seq* of the first N elements of a slowly convergent infinite sequence (A_k) , **shanks()** computes the iterated Shanks transformation $S(A), S(S(A)), \dots, S^{N/2}(A)$. The Shanks transformation often provides strong convergence acceleration, especially if the sequence is oscillating.

The iterated Shanks transformation is computed using the Wynn epsilon algorithm (see [1]). **shanks()** returns the full epsilon table generated by Wynn's algorithm, which can be read off as follows:

The table is a list of lists forming a lower triangular matrix, where higher row and column indices correspond to more accurate values.

The columns with even index hold dummy entries (required for the computation) and the columns with odd index hold the actual extrapolates.

The last element in the last row is typically the most accurate estimate of the limit.

The difference to the third last element in the last row provides an estimate of the approximation error.

The magnitude of the second last element provides an estimate of the numerical accuracy lost to cancellation.

For convenience, so the extrapolation is stopped at an odd index so that **shanks**(*seq*) [-1][-1] always gives an estimate of the limit.

Optionally, an existing table can be passed to **shanks()**. This can be used to efficiently extend a previous computation after new elements have been appended to the sequence. The table will then be updated in-place.

The Shanks transformation

The Shanks transformation is defined as follows (see [2]): given the input sequence (A_0, A_1, \dots) , the transformed sequence is given by

$$S(A_k) = \frac{A_{k+1}A_{k-1} - A_k^2}{A_{k+1} + A_{k-1} - 2A_k} \quad (22.4.1)$$

The Shanks transformation gives the exact limit A_∞ in a single step if $A_k = A + aq^k$. Note in particular that it extrapolates the exact sum of a geometric series in a single step.

Applying the Shanks transformation once often improves convergence substantially for an arbitrary sequence, but the optimal effect is obtained by applying it iteratively: $S(S(A_k)), S(S(S(A_k))), \dots$

WynnâŽs epsilon algorithm provides an efficient way to generate the table of iterated Shanks transformations. It reduces the computation of each element to essentially a single division, at the cost of requiring dummy elements in the table. See [1] for details.

Precision issues

Due to cancellation effects, the sequence must be typically be computed at a much higher precision than the target accuracy of the extrapolation.

If the Shanks transformation converges to the exact limit (such as if the sequence is a geometric series), then a division by zero occurs. By default, shanks() handles this case by terminating the iteration and returning the table it has generated so far. With *randomized=True*, it will instead replace the zero by a pseudorandom number close to zero. (TODO: find a better solution to this problem.)

Examples

We illustrate by applying Shanks transformation to the Leibniz series for π :

```
>>> from mpFormulaPy import *
>>> mp.dps = 50
>>> S = [4*sum(mpf(-1)**n/(2*n+1) for n in range(m))
... for m in range(1,30)]
>>>
>>> T = shanks(S[:7])
>>> for row in T:
...     nprint(row)
...
[-0.75]
[1.25, 3.16667]
[-1.75, 3.13333, -28.75]
[2.25, 3.14524, 82.25, 3.14234]
[-2.75, 3.13968, -177.75, 3.14139, -969.937]
[3.25, 3.14271, 327.25, 3.14166, 3515.06, 3.14161]
```

The extrapolated accuracy is about 4 digits, and about 4 digits may have been lost due to cancellation:

```
>>> L = T[-1]
>>> nprint([abs(L[-1] - pi), abs(L[-1] - L[-3]), abs(L[-2])])
[2.22532e-5, 4.78309e-5, 3515.06]
```

Now we extend the computation:

```
>>> T = shanks(S[:25], T)
>>> L = T[-1]
>>> nprint([abs(L[-1] - pi), abs(L[-1] - L[-3]), abs(L[-2])])
[3.75527e-19, 1.48478e-19, 2.96014e+17]
```

The value for π is now accurate to 18 digits. About 18 digits may also have been lost to cancellation.

Here is an example with a geometric series, where the convergence is immediate (the sum is

exactly 1):

```
>>> mp.dps = 15
>>> for row in shanks([0.5, 0.75, 0.875, 0.9375, 0.96875]):
...     nprint(row)
[4.0]
[8.0, 1.0]
```

22.4.3 Levin's algorithm

Function **levin**(*Keywords* As String) As Object

The function **levin** returns an object with `sn update` method. Only for use within Python

Parameter:

Keywords: `method='levin'`, `variant='u'`.

This interface implements Levin's (nonlinear) sequence transformation for convergence acceleration and summation of divergent series. It performs better than the Shanks/Wynn-epsilon algorithm for logarithmic convergent or alternating divergent series.

Let A be the series we want to sum:

$$A = \sum_{k=0}^{\infty} a_k. \quad (22.4.2)$$

Attention: all a_k must be non-zero!

Let s_n be the partial sums of this series:

$$s_n = \sum_{k=0}^n a_k. \quad (22.4.3)$$

Methods

Calling **levin** returns an object with the following methods.

`update(...)` works with the list of individual terms a_k of A , and `update_psum(...)` works with the list of partial sums s_k of A :

```
v, e = ...update([a_0, a_1, ..., a_k])
v, e = ...update_psum([s_0, s_1, ..., s_k])
```

`step(...)` works with the individual terms and `step_psum(...)` works with the partial sums s_k :

```
v, e = ...step(a_k)
v, e = ...step_psum(s_k)
```

v is the current estimate for A , and e is an error estimate which is simply the difference between the current estimate and the last estimate. One should not mix `update`, `update_psum`, `step` and `step_psum`.

A word of caution

One can only hope for good results (i.e. convergence acceleration or resummation) if the s_n have some well defined asymptotic behavior for large n and are not erratic or random. Furthermore one usually needs very high working precision because of the numerical cancellation.

If the working precision is insufficient, levin may produce silently numerical garbage. Furthermore even if the Levin-transformation converges, in the general case there is no proof that the result is mathematically sound. Only for very special classes of problems one can prove that the Levin-transformation converges to the expected result (for example Stieltjes-type integrals).

Furthermore the Levintransform is quite expensive (i.e. slow) in comparison to Shanks/Wynn-epsilon, Richardson & co. In summary one can say that the Levin-transformation is powerful but unreliable and that it may need a copious amount of working precision.

The Levin transform has several variants differing in the choice of weights. Some variants are better suited for the possible flavours of convergence behaviour of A than other variants:

```
convergence behaviour levin-u levin-t levin-v shanks/wynn-epsilon
```

	+	-	+	-
logarithmic	+	-	+	-
linear	+	+	+	+
alternating divergent	+	+	+	+

"+" means the variant **is** suitable, "-" means the variant **is not** suitable;
for comparison the Shanks/Wynn-epsilon transform **is** listed, too.

The variant is controlled through the variant keyword (i.e. `variant="u"`, `variant="t"` or `variant="v"`). Overall "u" is probably the best choice. Finally it is possible to use the Sidi-S transform instead of the Levin transform by using the keyword `method='sidi'`. The Sidi-S transform works better than the Levin transformation for some divergent series (see the examples).

Parameters:

<code>method</code>	<code>"levin"</code> or <code>"sidi"</code> chooses either the Levin or the Sidi-S transformation
<code>variant</code>	<code>"u"</code> , <code>"t"</code> or <code>"v"</code> chooses the weight variant.

The Levin transform is also accessible through the nsum interface. `method="l"` or `method="levin"` select the normal Levin transform while `method="sidi"` selects the Sidi-S transform. The variant is in both cases selected through the `levin_variant` keyword. The stepsize in `nsum()` must not be chosen too large, otherwise it will miss the point where the Levin transform converges resulting in numerical overflow/garbage. For highly divergent series a copious amount of working precision must be chosen.

Examples

First we sum the zeta function:

```
>>> from mpFormulaPy import mp
>>> mp.prec = 53
>>> eps = mp.mpf(mp.eps)
>>> with mp.extrarec(2 * mp.prec): # levin needs a high working precision
...     L = mp.levin(method = "levin", variant = "u")
...     S, s, n = [], 0, 1
...     while 1:
...         s += mp.one / (n * n)
...         n += 1
...         S.append(s)
...         v, e = L.update_psum(S)
...         if e < eps:
...             break
```

```

...         if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - mp.pi ** 2 / 6))
0.0
>>> w = mp.nsum(lambda n: 1 / (n*n), [1, mp.inf], method = "levin", levin_variant =
    "u")
>>> print(mp.chop(v - w))
0.0

```

Now we sum the zeta function outside its range of convergence (attention: This does not work at the negative integers!):

```

>>> eps = mp.mpf(mp.eps)
>>> with mp.extrarec(2 * mp.prec): # levin needs a high working precision
...     L = mp.levin(method = "levin", variant = "v")
...     A, n = [], 1
...     while 1:
...         s = mp.mpf(n) ** (2 + 3j)
...         n += 1
...         A.append(s)
...         v, e = L.update(A)
...         if e < eps:
...             break
...     if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - mp.zeta(-2-3j)))
0.0
>>> w = mp.nsum(lambda n: n ** (2 + 3j), [1, mp.inf], method = "levin", levin_variant =
    "v")
>>> print(mp.chop(v - w))
0.0

```

Now we sum the divergent asymptotic expansion of an integral related to the exponential integral (see also [2] p.373). The Sidi-S transform works best here:

```

>>> z = mp.mpf(10)
>>> exact = mp.quad(lambda x: mp.exp(-x)/(1+x/z), [0,mp.inf])
>>> # exact = z * mp.exp(z) * mp.expint(1,z) # this is the symbolic expression for
    the integral
>>> eps = mp.mpf(mp.eps)
>>> with mp.extrarec(2 * mp.prec): # high working precisions are mandatory for
    divergent resummation
...     L = mp.levin(method = "sidi", variant = "t")
...     n = 0
...     while 1:
...         s = (-1)**n * mp.fac(n) * z ** (-n)
...         v, e = L.step(s)
...         n += 1
...         if e < eps:
...             break
...     if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - exact))
0.0
>>> w = mp.nsum(lambda n: (-1) ** n * mp.fac(n) * z ** (-n), [0, mp.inf], method =

```

```

"sidi", levin_variant = "t")
>>> print(mp.chop(v - w))
0.0

```

Another highly divergent integral is also summable:

```

>>> z = mp.mpf(2)
>>> eps = mp.mpf(mp.eps)
>>> exact = mp.quad(lambda x: mp.exp(-x * x / 2 - z * x ** 4), [0,mp.inf]) * 2 /
    mp.sqrt(2 * mp.pi)
>>> # exact = mp.exp(mp.one / (32 * z)) * mp.besselk(mp.one / 4, mp.one / (32 * z)) /
    (4 * mp.sqrt(z * mp.pi)) # this is the symbolic expression for the integral
>>> with mp.extrarec(7 * mp.prec): # we need copious amount of precision to sum this
    highly divergent series
...     L = mp.levin(method = "levin", variant = "t")
...     n, s = 0, 0
...     while 1:
...         s += (-z)**n * mp.fac(4 * n) / (mp.fac(n) * mp.fac(2 * n) * (4 ** n))
...         n += 1
...         v, e = L.step_psum(s)
...         if e < eps:
...             break
...         if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - exact))
0.0
>>> w = mp.nsum(lambda n: (-z)**n * mp.fac(4 * n) / (mp.fac(n) * mp.fac(2 * n) * (4
    ** n)),
...     [0, mp.inf], method = "levin", levin_variant = "t", workprec = 8*mp.prec, steps
    = [2] + [1 for x in xrange(1000)])
>>> print(mp.chop(v - w))
0.0

```

These examples run with 15-20 decimal digits precision. For higher precision the working precision must be raised.

Examples for nsum

Here we calculate Euler's constant as the constant term in the Laurent expansion of $\zeta(s)$ at $s = 1$. This sum converges extremely slowly because of the logarithmic convergence behaviour of the Dirichlet series for zeta:

```

>>> mp.dps = 30
>>> z = mp.mpf(10) ** (-10)
>>> a = mp.nsum(lambda n: n**(-(1+z)), [1, mp.inf], method = "l") - 1 / z
>>> print(mp.chop(a - mp.euler, tol = 1e-10))
0.0

```

The Sidi-S transform performs excellently for the alternating series of $\log(2)$:

```

>>> a = mp.nsum(lambda n: (-1)**(n-1) / n, [1, mp.inf], method = "sidi")
>>> print(mp.chop(a - mp.log(2)))
0.0

```

Hypergeometric series can also be summed outside their range of convergence. The stepsize in

`nsum()` must not be chosen too large, otherwise it will miss the point where the Levin transform converges resulting in numerical overflow/garbage:

```
>>> z = 2 + 1j
>>> exact = mp.hyp2f1(2 / mp.mpf(3), 4 / mp.mpf(3), 1 / mp.mpf(3), z)
>>> f = lambda n: mp.rf(2 / mp.mpf(3), n) * mp.rf(4 / mp.mpf(3), n) * z**n / (mp.rf(1
    / mp.mpf(3), n) * mp.fac(n))
>>> v = mp.nsum(f, [0, mp.inf], method = "levin", steps = [10 for x in xrange(1000)])
>>> print(mp.chop(exact-v))
0.0
```

22.4.4 Cohen's algorithm

Function `cohen_alt()` As Object

The function `cohen_alt` returns an object with an `update` method. Only for use within Python

This interface implements the convergence acceleration of alternating series as described in H. Cohen, F.R. Villegas, D. Zagier - "Convergence Acceleration of Alternating Series". This series transformation works only well if the individual terms of the series have an alternating sign. It belongs to the class of linear series transformations (in contrast to the Shanks/Wynn-epsilon or Levin transform). This series transformation is also able to sum some types of divergent series. See the paper under which conditions this resummation is mathematical sound.

Let A be the series we want to sum:

$$A = \sum_{k=0}^{\infty} a_k \quad (22.4.4)$$

Let s_n be the partial sums of this series:

$$s_n = \sum_{k=0}^n a_k. \quad (22.4.5)$$

Interface

Calling `cohen_alt` returns an object with the following methods.

Then `update(...)` works with the list of individual terms a_k and `update_psum(...)` works with the list of partial sums s_k :

```
v, e = ...update([a_0, a_1, ..., a_k])
v, e = ...update_psum([s_0, s_1, ..., s_k])
```

v is the current estimate for A , and e is an error estimate which is simply the difference between the current estimate and the last estimate.

Examples

Here we compute the alternating zeta function using `update_psum`:

```
>>> from mpFormulaPy import mp
>>> AC = mp.cohen_alt()
>>> S, s, n = [], 0, 1
>>> while 1:
...     s += -((-1) ** n) * mp.one / (n * n)
...     n += 1
```

```

... S.append(s)
... v, e = AC.update_psum(S)
... if e < mp.eps:
...     break
... if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> print(mp.chop(v - mp.pi ** 2 / 12))
0.0

```

Here we compute the product $\prod_{n=1}^{\infty} \Gamma(1 + 1/(2n - 1))/\Gamma(1 + 1/(2n))$:

```

>>> A = []
>>> AC = mp.cohen_alt()
>>> n = 1
>>> while 1:
...     A.append(mp.loggamma(1 + mp.one / (2 * n - 1)))
...     A.append(-mp.loggamma(1 + mp.one / (2 * n)))
...     n += 1
...     v, e = AC.update(A)
...     if e < mp.eps:
...         break
...     if n > 1000: raise RuntimeError("iteration limit exceeded")
>>> v = mp.exp(v)
>>> print(mp.chop(v - 1.06215090557106, tol = 1e-12))
0.0

```

cohen_alt is also accessible through the nsum() interface:

```

>>> v = mp.nsum(lambda n: (-1)**(n-1) / n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.log(2)))
0.0
>>> v = mp.nsum(lambda n: (-1)**n / (2 * n + 1), [0, mp.inf], method = "a")
>>> print(mp.chop(v - mp.pi / 4))
0.0
>>> v = mp.nsum(lambda n: (-1)**n * mp.log(n) * n, [1, mp.inf], method = "a")
>>> print(mp.chop(v - mp.diff(lambda s: mp.altzeta(s), -1)))
0.0

```

Chapter 23

Differentiation

23.1 Numerical derivatives

23.1.1 One-dimensional Differentiation

Function **diff**(*f* As *mpNum*, *x* As *mpNum*, *n* As *mpNum*, **Keywords** As String) As *mpNum*

The function *diff* returns the *n*-th derivative $f^{(n)}(x)$.

Parameters:

f: A one dimensional function

x: A real number.

n: An integer, indicating the *n*th derivative.

Keywords: method=step, tol=eps, direction=0. Many more, see below.

Numerically computes the derivative of *f*, $f'(x)$, or generally for an integer $n > 0$, the *n*-th derivative $f^{(n)}(x)$. A few basic examples are:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> diff(lambda x: x**2 + x, 1.0)
3.0
>>> diff(lambda x: x**2 + x, 1.0, 2)
2.0
>>> diff(lambda x: x**2 + x, 1.0, 3)
0.0
>>> nprint([diff(exp, 3, n) for n in range(5)]) # exp'(x) = exp(x)
[20.0855, 20.0855, 20.0855, 20.0855, 20.0855]
```

Even more generally, given a tuple of arguments (x_1, \dots, x_k) and order (n_1, \dots, n_k) , the partial derivative $f^{(n_1, \dots, n_k)}(x_1, \dots, x_k)$ is evaluated. For example:

```
>>> diff(lambda x,y: 3*x*y + 2*y - x, (0.25, 0.5), (0,1))
2.75
>>> diff(lambda x,y: 3*x*y + 2*y - x, (0.25, 0.5), (1,1))
3.0
```

Options

The following optional keyword arguments are recognized:

method: Supported methods are 'step' or 'quad': derivatives may be computed using either a finite difference with a small step size h (default), or numerical quadrature.

direction: Direction of finite difference: can be -1 for a left difference, 0 for a central difference (default), or +1 for a right difference; more generally can be any complex number.

addprec: Extra precision for h used to account for the function's sensitivity to perturbations (default = 10).

relative: Choose h relative to the magnitude of x , rather than an absolute value; useful for large or tiny x (default = False).

h: As an alternative to addprec and relative, manually select the step size h .

singular: If True, evaluation exactly at the point x is avoided; this is useful for differentiating functions with removable singularities. Default = False.

radius: Radius of integration contour (with method = 'quad'). Default = 0.25. A larger radius typically is faster and more accurate, but it must be chosen so that f has no singularities within the radius from the evaluation point.

A finite difference requires $n + 1$ function evaluations and must be performed at $n + 1$ times the target precision. Accordingly, f must support fast evaluation at high precision.

With integration, a larger number of function evaluations is required, but not much extra precision is required. For high order derivatives, this method may thus be faster if f is very expensive to evaluate at high precision.

Further examples

The direction option is useful for computing left- or right-sided derivatives of nonsmooth functions:

```
>>> diff(abs, 0, direction=0)
0.0
>>> diff(abs, 0, direction=1)
1.0
>>> diff(abs, 0, direction=-1)
-1.0
```

More generally, if the direction is nonzero, a right difference is computed where the step size is multiplied by $\text{sign}(\text{direction})$. For example, with $\text{direction}=+j$, the derivative from the positive imaginary direction will be computed:

```
>>> diff(abs, 0, direction=j)
(0.0 - 1.0j)
```

With integration, the result may have a small imaginary part even if the result is purely real:

```
>>> diff(sqrt, 1, method="quad")
(0.5 - 4.59...e-26j)
>>> chop(_)
0.5
```

Adding precision to obtain an accurate value:

```
>>> diff(cos, 1e-30)
```

```
0.0
>>> diff(cos, 1e-30, h=0.0001)
-9.9999998328279e-31
>>> diff(cos, 1e-30, addprec=100)
-1.0e-30
```

23.1.2 Sequence of derivatives

Function **diffs**(*f* As *mpNum*, *x* As *mpNum*, *n* As *mpNum*, **Keywords** As *String*) As *mpNum*

The function **diffs** returns the *n*-th derivative $f^{(n)}(x)$.

Parameters:

f: A one dimensional function

x: A real number.

n: An integer, indicating the *n*th derivative.

Keywords: method=step, tol=eps, direction=0. Many more, see below.

Returns a generator that yields the sequence of derivatives

$$f(x), f'(x), f''(x), \dots, f^{(k)}(x), \dots \quad (23.1.1)$$

With *method*='step', **diffs()** uses only $O(k)$ function evaluations to generate the first *k* derivatives, rather than the roughly $O(k^2)$ evaluations required if one calls **diff()** *k* separate times.

With $n < \infty$, the generator stops as soon as the *n*-th derivative has been generated. If the exact number of needed derivatives is known in advance, this is further slightly more efficient.

Options are the same as for **diff()**.

Examples

```
>>> from mpFormulaPy import *
>>> mp.dps = 15
>>> nprint(list(diffs(cos, 1, 5)))
[0.540302, -0.841471, -0.540302, 0.841471, 0.540302, -0.841471]
>>> for i, d in zip(range(6), diffs(cos, 1)):
... print("%s %s" % (i, d))
...
0 0.54030230586814
1 -0.841470984807897
2 -0.54030230586814
3 0.841470984807897
4 0.54030230586814
5 -0.841470984807897
```

23.2 Composition of derivatives

23.2.1 Differentiation of products of functions

Function **diffsprod**(*factors* As Object) As mpNum

The function `diffsprod` returns the result of the differentiation of products of functions.

Parameter:

factors: a list of N iterables or generators.

Given a list of N iterables or generators yielding $f_k(x), f'_k(x), f''_k(x), \dots$ for $k = 1, \dots, N$, generate $g_k(x), g'_k(x), g''_k(x), \dots$ where $g(x) = f_1(x)f_2(x) \cdots f_N(x)$.

At high precision and for large orders, this is typically more efficient than numerical differentiation if the derivatives of each $f_k(x)$ admit direct computation.

Note: This function does not increase the working precision internally, so guard digits may have to be added externally for full accuracy.

Examples

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> f = lambda x: exp(x)*cos(x)*sin(x)
>>> u = diffs(f, 1)
>>> v = mp.diffprod([diffs(exp,1), diffs(cos,1), diffs(sin,1)])
>>> next(u); next(v)
1.23586333600241
1.23586333600241
>>> next(u); next(v)
0.104658952245596
0.104658952245596
>>> next(u); next(v)
-5.96999877552086
-5.96999877552086
>>> next(u); next(v)
-12.4632923122697
-12.4632923122697
```

23.2.2 Differentiation of the exponential of functions

Function **diffsexp**(*fdiffs* As Object) As mpNum

The function `diffsexp` returns the result of the differentiation of the exponential of functions.

Parameter:

fdiffs: a list of N iterables or generators.

Given an iterable or generator yielding $f_k(x), f'_k(x), f''_k(x), \dots$ generate $g_k(x), g'_k(x), g''_k(x), \dots$ where $g(x) = \exp(f(x))$.

At high precision and for large orders, this is typically more efficient than numerical differentiation if the derivatives of $f(x)$ admit direct computation.

Note: This function does not increase the working precision internally, so guard digits may have to be added externally for full accuracy.

Examples

The derivatives of the gamma function can be computed using logarithmic differentiation:

```

>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>>
>>> def diffs_loggamma(x):
...     yield loggamma(x)
...     i = 0
...     while 1:
...         yield psi(i,x)
...         i += 1
...
>>> u = diffs_exp(diffs_loggamma(3))
>>> v = diffs(gamma, 3)
>>> next(u); next(v)
2.0
2.0
>>> next(u); next(v)
1.84556867019693
1.84556867019693
>>> next(u); next(v)
2.49292999190269
2.49292999190269
>>> next(u); next(v)
3.44996501352367
3.44996501352367

```

23.3 Fractional derivatives

Function **different**(*f* As *mpNum*, *x* As *mpNum*, *n* As *mpNum*, *x0* As *mpNum*) As *mpNum*

The function **different** returns the Riemann-Liouville differintegral.

Parameters:

f: A one dimensional function

x: A real interval.

n: A real interval.

x0: A real interval.

Calculates the Riemann-Liouville differintegral, or fractional derivative, defined by

$${}_{x_0} \mathbb{D}_x^n f(x) = \frac{1}{\Gamma(m-n)} \frac{d^m}{dx^m} \int_{x_0}^x (x-t)^{m-n-1} f(t) dt \quad (23.3.1)$$

where *f* is a given (presumably well-behaved) function, *x* is the evaluation point, *n* is the order, and *x0* is the reference point of integration (*m* is an arbitrary parameter selected automatically).

With $n = 1$, this is just the standard derivative $f'(x)$; with $n = 2$, the second derivative $f''(x)$, etc. With $n = -1$, it gives $\int_{x_0}^x f(t)dt$, with $n = -1$ it gives $\int_{x_0}^x \left(\int_{x_0}^x f(u)du \right) dt$, etc.

As n is permitted to be any number, this operator generalizes iterated differentiation and iterated integration to a single operator with a continuous order parameter.

Examples

There is an exact formula for the fractional derivative of a monomial x^p , which may be used as a reference. For example, the following gives a half-derivative (order 0.5):

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> x = mpf(3); p = 2; n = 0.5
>>> differint(lambda t: t**p, x, n)
7.81764019044672
>>> gamma(p+1)/gamma(p-n+1) * x**(p-n)
7.81764019044672
```

Another useful test function is the exponential function, whose integration / differentiation formula easy generalizes to arbitrary order. Here we first compute a third derivative, and then a triply nested integral. (The reference point x_0 is set to $-\infty$ to avoid nonzero endpoint terms.):

```
>>> differint(lambda x: exp(pi*x), -1.5, 3)
0.278538406900792
>>> exp(pi*-1.5) * pi**3
0.278538406900792
>>> differint(lambda x: exp(pi*x), 3.5, -3, -inf)
1922.50563031149
>>> exp(pi*3.5) / pi**3
1922.50563031149
```

However, for noninteger n , the differentiation formula for the exponential function must be modified to give the same result as the Riemann-Liouville differintegral:

```
>>> x = mpf(3.5)
>>> c = pi
>>> n = 1+2*j
>>> differint(lambda x: exp(c*x), x, n)
(-123295.005390743 + 140955.117867654j)
>>> x**(-n) * exp(c)**x * (x*c)**n * gammaint(-n, 0, x*c) / gamma(-n)
(-123295.005390743 + 140955.117867654j)
```

Chapter 24

Numerical integration (quadrature)

24.1 Standard quadrature

24.1.1 One-dimensional Integration

Function **quad**(*f* As *mpNum*, *interval* As *mpNum*, **Keywords** As *String*) As *mpNum*

The function **quad** returns a one dimensional integral.

Parameters:

f: A one dimensional function

interval: A real interval.

Keywords: method=TanhSinh, error=False, verbose=False, maxdegree. Many more, see below.

Computes a single, double or triple integral over a given 1D interval, 2D rectangle, or 3D cuboid. A basic example:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> quad(sin, [0, pi])
2.0
```

A basic 2D integral:

```
>>> f = lambda x, y: cos(x+y/2)
>>> quad2d(f, [-pi/2, pi/2], [0, pi])
4.0
```

Interval format

The integration range for each dimension may be specified using a list or tuple. Arguments are interpreted as follows:

quad(*f*, [*x*₁, *x*₂]) calculates $\int_{x_1}^{x_2} f(x) dx$

quad2d(*f*, [*x*₁, *x*₂], [*y*₁, *y*₂]) calculates $\int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x, y) dy dx$

quad3d(*f*, [*x*₁, *x*₂], [*y*₁, *y*₂], [*z*₁, *z*₂]) calculates $\int_{x_1}^{x_2} \int_{y_1}^{y_2} \int_{z_1}^{z_2} f(x, y, z) dz dy dx$

Endpoints may be finite or infinite. An interval descriptor may also contain more than two points. In this case, the integration is split into subintervals, between each pair of consecutive points. This

is useful for dealing with mid-interval discontinuities, or integrating over large intervals where the function is irregular or oscillates.

Options

`quad()` recognizes the following keyword arguments:

method: Chooses integration algorithm (described below).

error: If set to true, `quad()` returns (v, e) where v is the integral and e is the estimated error.

maxdegree: Maximum degree of the quadrature rule to try before quitting.

verbose: Print details about progress.

Algorithms

Mpmath presently implements two integration algorithms: tanh-sinh quadrature and Gauss-Legendre quadrature. These can be selected using `method='tanh-sinh'` or `method='gauss-legendre'`, or by passing the classes `method=TanhSinh`, `method=GaussLegendre`. The functions `quadts()` and `quadgl()` are also available as shortcuts.

Both algorithms have the property that doubling the number of evaluation points roughly doubles the accuracy, so both are ideal for high precision quadrature (hundreds or thousands of digits).

At high precision, computing the nodes and weights for the integration can be expensive (more expensive than computing the function values). To make repeated integrations fast, nodes are automatically cached.

The advantages of the tanh-sinh algorithm are that it tends to handle endpoint singularities well, and that the nodes are cheap to compute on the first run. For these reasons, it is used by `quad()` as the default algorithm.

Gauss-Legendre quadrature often requires fewer function evaluations, and is therefore often faster for repeated use, but the algorithm does not handle endpoint singularities as well and the nodes are more expensive to compute. Gauss-Legendre quadrature can be a better choice if the integrand is smooth and repeated integrations are required (e.g. for multiple integrals).

See the documentation for `TanhSinh` and `GaussLegendre` for additional details.

Examples of 1D integrals

Intervals may be infinite or half-infinite. The following two examples evaluate the limits of the inverse tangent function ($\int 1/(1+x^2) = \tan^{-1} x$), and the Gaussian integral $\int_{-\infty}^{\infty} \exp(-x^2) dx = \sqrt{\pi}$:

```
>>> mp.dps = 15
>>> quad(lambda x: 2/(x**2+1), [0, inf])
3.14159265358979
>>> quad(lambda x: exp(-x**2), [-inf, inf])**2
3.14159265358979
```

Integrals can typically be resolved to high precision. The following computes 50 digits of π by integrating the area of the half-circle defined by $x^2 + y^2 \leq 1, -1 < x < 1, y \geq 0$:

```
>>> mp.dps = 50
>>> 2*quad(lambda x: sqrt(1-x**2), [-1, 1])
3.1415926535897932384626433832795028841971693993751
```

One can just as well compute 1000 digits (output truncated):

```
>>> mp.dps = 1000
>>> 2*quad(lambda x: sqrt(1-x**2), [-1, 1])
3.141592653589793238462643383279502884...216420198
```

Complex integrals are supported. The following computes a residue at $z = 0$ by integrating counterclockwise along the diamond-shaped path from 1 to $+i$ to -1 to $-i$ to 1:

```
>>> mp.dps = 15
>>> chop(quad(lambda z: 1/z, [1,j,-1,-j,1]))
(0.0 + 6.28318530717959j)
```

24.1.1.1 Singularities

Both tanh-sinh and Gauss-Legendre quadrature are designed to integrate smooth (infinitely differentiable) functions. Neither algorithm copes well with mid-interval singularities (such as mid-interval discontinuities in $f(x)$ or $f'(x)$). The best solution is to split the integral into parts:

```
>>> mp.dps = 15
>>> quad(lambda x: abs(sin(x)), [0, 2*pi]) # Bad
3.99900894176779
>>> quad(lambda x: abs(sin(x)), [0, pi, 2*pi]) # Good
4.0
```

The tanh-sinh rule often works well for integrands having a singularity at one or both endpoints:

```
>>> mp.dps = 15
>>> quad(log, [0, 1], method="tanh-sinh") # Good
-1.0
>>> quad(log, [0, 1], method="gauss-legendre") # Bad
-0.999932197413801
```

However, the result may still be inaccurate for some functions:

```
>>> quad(lambda x: 1/sqrt(x), [0, 1], method="tanh-sinh")
1.99999999946942
```

This problem is not due to the quadrature rule per se, but to numerical amplification of errors in the nodes. The problem can be circumvented by temporarily increasing the precision:

```
>>> mp.dps = 30
>>> a = quad(lambda x: 1/sqrt(x), [0, 1], method="tanh-sinh")
>>> mp.dps = 15
>>> +a
2.0
```

24.1.1.2 Highly variable functions

For functions that are smooth (in the sense of being infinitely differentiable) but contain sharp mid-interval peaks or many 'bumps', quad() may fail to provide full accuracy.

For example, with default settings, quad() is able to integrate $\sin(x)$ accurately over an interval of length 100 but not over length 1000:

```
>>> quad(sin, [0, 100]); 1-cos(100) # Good
0.137681127712316
0.137681127712316
>>> quad(sin, [0, 1000]); 1-cos(1000) # Bad
-37.8587612408485
0.437620923709297
```

One solution is to break the integration into 10 intervals of length 100:

```
>>> quad(sin, linspace(0, 1000, 10)) # Good
0.437620923709297
```

Another is to increase the degree of the quadrature:

```
>>> quad(sin, [0, 1000], maxdegree=10) # Also good
0.437620923709297
```

Whether splitting the interval or increasing the degree is more efficient differs from case to case. Another example is the function $1/(1+x^2)$, which has a sharp peak centered around $x = 0$:

```
>>> f = lambda x: 1/(1+x**2)
>>> quad(f, [-100, 100]) # Bad
3.64804647105268
>>> quad(f, [-100, 100], maxdegree=10) # Good
3.12159332021646
>>> quad(f, [-100, 0, 100]) # Also good
3.12159332021646
```

24.1.2 Two-dimensional Integration

Function **quad2d**(*f* As *mpNum*, *interval1* As *mpNum*, *interval2* As *mpNum*, **Keywords** As *String*) As *mpNum*

The function quad2d returns a two dimensional integral.

Parameters:

f: A one dimensional function

interval1: A real interval.

interval2: A real interval.

Keywords: method=TanhSinh, error=False, verbose=False, maxdegree. Many more, see below.

Here are several nice examples of analytically solvable 2D integrals (taken from MathWorld [1]) that can be evaluated to high precision fairly rapidly by quad():

```
>>> mp.dps = 30
>>> f = lambda x, y: (x-1)/((1-x*y)*log(x*y))
>>> quad2d(f, [0, 1], [0, 1])
0.577215664901532860606512090082
>>> +euler
```

```

0.577215664901532860606512090082
>>> f = lambda x, y: 1/sqrt(1+x**2+y**2)
>>> quad2d(f, [-1, 1], [-1, 1])
3.17343648530607134219175646705
>>> 4*log(2+sqrt(3))-2*pi/3
3.17343648530607134219175646705
>>> f = lambda x, y: 1/(1-x**2 * y**2)
>>> quad2d(f, [0, 1], [0, 1])
1.23370055013616982735431137498
>>> pi**2 / 8
1.23370055013616982735431137498
>>> quad2d(lambda x, y: 1/(1-x*y), [0, 1], [0, 1])
1.64493406684822643647241516665
>>> pi**2 / 6
1.64493406684822643647241516665

```

Multiple integrals may be done over infinite ranges:

```

>>> mp.dps = 15
>>> print(quad2d(lambda x,y: exp(-x-y), [0, inf], [1, inf]))
0.367879441171442
>>> print(1/e)
0.367879441171442

```

For nonrectangular areas, one can call quad() recursively. For example, we can replicate the earlier example of calculating π by integrating over the unit-circle, and use double quadrature to actually measure the area circle:

```

>>> f = lambda x: quad(lambda y: 1, [-sqrt(1-x**2), sqrt(1-x**2)])
>>> quad(f, [-1, 1])
3.14159265358979

```

24.1.3 Three-dimensional Integration

Function **quad3d(f As mpNum, interval1 As mpNum, interval2 As mpNum, interval3 As mpNum, Keywords As String)** As mpNum

The function quad3d returns a three dimensional integral.

Parameters:

f: A one dimensional function

interval1: A real interval.

interval2: A real interval.

interval3: A real interval.

Keywords: method=TanhSinh, error=False, verbose=False, maxdegree. Many more, see below.

Here is a simple triple integral:

```

>>> mp.dps = 15
>>> f = lambda x,y,z: x*y/(1+z)
>>> quad3d(f, [0,1], [0,1], [1,2], method="gauss-legendre")
0.101366277027041

```

```
>>> (log(3)-log(2))/4
0.101366277027041
```

24.1.4 Oscillatory integrals

Function **quadosc**(*f* As *mpNum*, *interval* As *mpNum*, *Keywords* As *String*) As *mpNum*

The function `quadosc` returns a one dimensional oscillatory integral.

Parameters:

f: A one dimensional function

interval: A real interval.

Keywords: `omega`=None, `period`=None, `zeros`=None

Calculates

$$I = \int_a^b f(x) dx \quad (24.1.1)$$

where at least one of *a* and *b* is infinite and where $f(x) = g(x) \cos(\omega x + \phi)$ for some slowly decreasing function *g*(*x*). With proper input, `quadosc()` can also handle oscillatory integrals where the oscillation rate is different from a pure sine or cosine wave.

In the standard case when $|a| < \infty, b = \infty$, `quadosc()` works by evaluating the infinite series

$$I = \int_a^{x_1} f(x) dx + \sum_{k=1}^{\infty} \int_{x_k}^{x_{k+1}} f(x) dx \quad (24.1.2)$$

where x_k are consecutive zeros (alternatively some other periodic reference point) of *f*(*x*). Accordingly, `quadosc()` requires information about the zeros of *f*(*x*). For a periodic function, you can specify the zeros by either providing the angular frequency ω (`omega`) or the period $2\pi/\omega$. In general, you can specify the *n*-th zero by providing the `zeros` arguments. Below is an example of each:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> f = lambda x: sin(3*x)/(x**2+1)
>>> quadosc(f, [0,inf], omega=3)
0.37833007080198
>>> quadosc(f, [0,inf], period=2*pi/3)
0.37833007080198
>>> quadosc(f, [0,inf], zeros=lambda n: pi*n/3)
0.37833007080198
>>> (ei(3)*exp(-3)-exp(3)*ei(-3))/2 # Computed by Mathematica
0.37833007080198
```

Note that `zeros` was specified to multiply *n* by the half-period, not the full period. In theory, it does not matter whether each partial integral is done over a half period or a full period. However, if done over half-periods, the infinite series passed to `nsum()` becomes an alternating series and this typically makes the extrapolation much more efficient.

Here is an example of an integration over the entire real line, and a half-infinite integration starting at $-\infty$:

```
>>> quadosc(lambda x: cos(x)/(1+x**2), [-inf, inf], omega=1)
1.15572734979092
>>> pi/e
1.15572734979092
>>> quadosc(lambda x: cos(x)/x**2, [-inf, -1], period=2*pi)
-0.0844109505595739
>>> cos(1)+sin(1)-pi/2
-0.0844109505595738
```

Of course, the integrand may contain a complex exponential just as well as a real sine or cosine:

```
>>> quadosc(lambda x: exp(3*j*x)/(1+x**2), [-inf, inf], omega=3)
(0.156410688228254 + 0.0j)
>>> pi/e**3
0.156410688228254
>>> quadosc(lambda x: exp(3*j*x)/(2+x+x**2), [-inf, inf], omega=3)
(0.0031748698463794 - 0.0447701735209082j)
>>> 2*pi/sqrt(7)/exp(3*(j+sqrt(7))/2)
(0.0031748698463794 - 0.0447701735209082j)
```

24.1.4.1 Non-periodic functions

If $f(x) = g(x)h(x)$ for some function $h(x)$ that is not strictly periodic, omega or period might not work, and it might be necessary to use zeros.

A notable exception can be made for Bessel functions which, though not periodic, are 'asymptotically periodic' in a sufficiently strong sense that the sum extrapolation will work out:

```
>>> quadosc(j0, [0, inf], period=2*pi)
1.0
>>> quadosc(j1, [0, inf], period=2*pi)
1.0
```

More properly, one should provide the exact Bessel function zeros:

```
>>> j0zero = lambda n: findroot(j0, pi*(n-0.25))
>>> quadosc(j0, [0, inf], zeros=j0zero)
1.0
```

For an example where zeros becomes necessary, consider the complete Fresnel integrals

$$\int_0^\infty \cos x^2 dx = \int_0^\infty \sin x^2 dx = \sqrt{\frac{\pi}{8}} \quad (24.1.3)$$

Although the integrands do not decrease in magnitude as $x \rightarrow \infty$, the integrals are convergent since the oscillation rate increases (causing consecutive periods to asymptotically cancel out). These integrals are virtually impossible to calculate to any kind of accuracy using standard quadrature rules. However, if one provides the correct asymptotic distribution of zeros ($x_n \sim \sqrt{n}$), quadosc() works:

```
>>> mp.dps = 30
>>> f = lambda x: cos(x**2)
>>> quadosc(f, [0, inf], zeros=lambda n: sqrt(pi*n))
```

```
0.626657068657750125603941321203
>>> f = lambda x: sin(x**2)
>>> quadosc(f, [0,inf], zeros=lambda n:sqrt(pi*n))
0.626657068657750125603941321203
>>> sqrt(pi/8)
0.626657068657750125603941321203
```

(Interestingly, these integrals can still be evaluated if one places some other constant than π in the square root sign.)

In general, if $f(x) \sim g(x) \cos(h(x))$, the zeros follow the inverse-function distribution $h^{-1}(x)$:

```
>>> mp.dps = 15
>>> f = lambda x: sin(exp(x))
>>> quadosc(f, [1,inf], zeros=lambda n: log(n))
-0.25024394235267
>>> pi/2-si(e)
-0.250243942352671
```

24.1.4.2 Non-alternating functions

If the integrand oscillates around a positive value, without alternating signs, the extrapolation might fail. A simple trick that sometimes works is to multiply or divide the frequency by 2:

```
>>> f = lambda x: 1/x**2+sin(x)/x**4
>>> quadosc(f, [1,inf], omega=1) # Bad
1.28642190869861
>>> quadosc(f, [1,inf], omega=0.5) # Perfect
1.28652953559617
>>> 1+(cos(1)+ci(1)+sin(1))/6
1.28652953559617
```

Fast decay

`quadosc()` is primarily useful for slowly decaying integrands. If the integrand decreases exponentially or faster, `quad()` will likely handle it without trouble (and generally be much faster than `quadosc()`):

```
>>> quadosc(lambda x: cos(x)/exp(x), [0, inf], omega=1)
0.5
>>> quad(lambda x: cos(x)/exp(x), [0, inf])
0.5
```

24.2 Main Quadrature rules

For a finite interval, a simple linear change of variables is used. Otherwise, the following transformations are used:

$$[a, \infty] : t = \frac{1}{x} + (a - 1) \quad (24.2.1)$$

$$[-\infty, b] : t = (b + 1) - \frac{1}{x} \quad (24.2.2)$$

$$[-\infty, \infty] : t = \frac{x}{\sqrt{1 - x^2}} \quad (24.2.3)$$

24.2.1 TanhSinh

Tanh-sinh quadrature is a method for numerical integration introduced by Hidetosi Takahasi and Masatake Mori in 1974. It uses the change of variables

$$x = \tanh\left(\frac{1}{2}\pi \sinh t\right) \quad (24.2.4)$$

to transform an integral on the interval $x \in (-1, +1)$ to an integral on the entire real line $t \in (-\infty, \infty)$. After this transformation, the integrand decays with a double exponential rate, and thus, this method is also known as the double exponential (DE) formula.

For a given step size h , the integral is approximated by the sum

$$\int_{-1}^1 f(x) dx \approx \sum_{k=-\infty}^{\infty} w_k f(x_k), \quad (24.2.5)$$

with the abscissas

$$x_k = \tanh\left(\frac{1}{2}\pi \sinh t_k\right) \quad (24.2.6)$$

and the weights

$$w_k = \frac{\tanh \frac{1}{2}\pi \cosh t_k}{\cosh^2(\frac{1}{2}\pi \sinh t_k)} \quad (24.2.7)$$

where $t_k = t_0 + hk$ for a step length $h \sim 2^{-m}$.

Like Gaussian quadrature, Tanh-Sinh quadrature is well suited for arbitrary-precision integration, where an accuracy of hundreds or even thousands of digits is desired. The convergence is exponential (in the discretization sense) for sufficiently well-behaved integrands: doubling the number of evaluation points roughly doubles the number of correct digits.

Tanh-Sinh quadrature is less efficient than Gaussian quadrature for smooth integrands, but unlike Gaussian quadrature tends to work equally well with integrands having singularities or infinite derivatives at one or both endpoints of the integration interval. A further advantage is that the abscissas and weights are relatively easy to compute. The cost of calculating abscissa-weight pairs for n -digit accuracy is roughly $n^2 \log^2 n$ compared to $n^3 \log n$ for Gaussian quadrature.

This class implements 'tanh-sinh' or 'doubly exponential' quadrature. This quadrature rule is based on the Euler-Maclaurin integral formula. By performing a change of variables involving nested exponentials / hyperbolic functions (hence the name), the derivatives at the endpoints vanish rapidly. Since the error term in the Euler-Maclaurin formula depends on the derivatives at the endpoints, a simple step sum becomes extremely accurate. In practice, this means that doubling the number of evaluation points roughly doubles the number of accurate digits.

The implementation of the tanh-sinh algorithm is based on the description given in Borwein, Bailey & Girgensohn, "Experimentation in Mathematics - Computational Paths to Discovery", A K Peters, 2003, pages 312-313. In the present implementation, a few improvements have been made:

A more efficient scheme is used to compute nodes (exploiting recurrence for the exponential function). The nodes are computed successively instead of all at once.

We exploit the fact that half of the abscissas at degree are precisely the abscissas from degree . Thus reusing the result from the previous level allows a 2x speedup.

A summary is given by [Bailey *et al.* \(2004\)](#); [Bailey \(2006\)](#).

See also the external links at [Wikipedia](#).

24.2.2 Gauss-Legendre

This class implements Gauss-Legendre quadrature, which is exceptionally efficient for polynomials and polynomial-like (i.e. very smooth) integrands.

The abscissas and weights are given by roots and values of Legendre polynomials, which are the orthogonal polynomials on $[-1, 1]$ with respect to the unit weight (see `legendre()`).

In this implementation, we take the 'degree' of the quadrature to denote a Gauss-Legendre rule of degree $3 \cdot 2^m$ (following Borwein, Bailey & Girgensohn). This way we get quadratic, rather than linear, convergence as the degree is incremented.

Comparison to tanh-sinh quadrature:

Is faster for smooth integrands once nodes have been computed

Initial computation of nodes is usually slower

Handles endpoint singularities worse

Handles infinite integration intervals worse

24.3 Additional Quadrature rules

24.3.1 Gauss-Legendre

$$\int_a^b f(x)dx = \frac{b-a}{2} \sum_{i=1}^n w_i f\left(\frac{1}{2}(b-a)(x_i+1)\right) + R_n, \quad \text{where} \quad (24.3.1)$$

$$w_i = \frac{2}{((1-x^2)P'_n(x_i))^2}, \quad R_n = \frac{f^{(2n)}(x)(b-a)^{2n+1}(n!)^4}{((2n+1)(2n)!)^3} \text{ for } a < x < b, \quad (24.3.2)$$

P_n are the Legendre polynomials, and x_i is the i th zero of P_n .

24.3.2 Gauss-Hermite

$$\int_{-\infty}^{\infty} e^{-ax^2} f(x)dx = \frac{1}{\sqrt{a}} \sum_{i=1}^n w_i f\left(\frac{x_i}{\sqrt{a}}\right) + R_n, \quad \text{where} \quad (24.3.3)$$

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{(n H_{n-1}(x_i))^2}, \quad R_n = \frac{f^{(2n)}(x) n! \sqrt{\pi}}{2^n (2n)!} \text{ for } -\infty < x < \infty, \quad (24.3.4)$$

H_n are the Hermite polynomials, and x_i is the i th zero of H_n .

24.3.3 Gauss-Laguerre

$$\int_0^{\infty} e^{-ax} f(x)dx = \frac{1}{a} \sum_{i=1}^n w_i f\left(\frac{x_i}{a}\right) + R_n, \quad \text{where} \quad (24.3.5)$$

$$w_i = \frac{x_i}{((n+1)L_{n+1}(x_i))^2}, \quad R_n = \frac{f^{(2n)}(x)(n!)^2}{(2n)!} \text{ for } 0 < x < \infty, \quad (24.3.6)$$

L_n are the Laguerre polynomials, and x_i is the i th zero of L_n .

Chapter 25

Ordinary differential equations

25.1 Solving the ODE initial value problem

Function **odefun**(*f* As mpNum, *x0* As mpNum, *y0* As mpNum, *Keywords* As String) As mpNum

The function **odefun** returns a function $y(x) = [y_0(x), y_1(x), \dots, y_n(x)]$ that is a numerical solution of a $n + 1$ -dimensional first-order ordinary differential equation (ODE) system

Parameters:

f: A one dimensional function

x0: A real number.

y0: A real number.

Keywords: tol=None, degree=None, method='taylor', verbose=False

The (ODE) system has the form

$$y'_0(x) = F_0(x, [y_0(x), y_1(x), \dots, y_n(x)]) \quad (25.1.1)$$

$$y'_1(x) = F_1(x, [y_0(x), y_1(x), \dots, y_n(x)]) \quad (25.1.2)$$

$$y'_n(x) = F_n(x, [y_0(x), y_1(x), \dots, y_n(x)]) \quad (25.1.3)$$

The derivatives are specified by the vector-valued function F that evaluates

$$[y'_0, \dots, y'_n] = F(x, [y_0, \dots, y_n]).$$

The initial point x_0 is specified by the scalar argument *x0*, and the initial value

$$y(x_0) = [y_0(x_0), \dots, y_n(x_0)]$$

is specified by the vector argument *y0*.

For convenience, if the system is one-dimensional, you may optionally provide just a scalar value for *y0*. In this case, F should accept a scalar *y* argument and return a scalar. The solution function *y* will return scalar values instead of length-1 vectors.

Evaluation of the solution function $y(x)$ is permitted for any $x > x_0$.

A high-order ODE can be solved by transforming it into first-order vector form. This transformation is described in standard texts on ODEs. Examples will also be given below.

Options, speed and accuracy

By default, **odefun()** uses a high-order Taylor series method. For reasonably wellbehaved problems, the solution will be fully accurate to within the working precision. Note that F must be possible to evaluate to very high precision for the generation of Taylor series to work.

To get a faster but less accurate solution, you can set a large value for tol (which defaults roughly to eps). If you just want to plot the solution or perform a basic simulation, $\text{tol} = 0.01$ is likely sufficient.

The degree argument controls the degree of the solver (with $\text{method}=\text{taylor}$, this is the degree of the Taylor series expansion). A higher degree means that a longer step can be taken before a new local solution must be generated from F , meaning that fewer steps are required to get from x_0 to a given x_1 . On the other hand, a higher degree also means that each local solution becomes more expensive (i.e., more evaluations of F are required per step, and at higher precision).

The optimal setting therefore involves a tradeoff. Generally, decreasing the degree for Taylor series is likely to give faster solution at low precision, while increasing is likely to be better at higher precision.

The function object returned by `odefun()` caches the solutions at all step points and uses polynomial interpolation between step points. Therefore, once $y(x_1)$ has been evaluated for some x_1 , $y(x)$ can be evaluated very quickly for any $x_0 \leq x \leq x_1$, and continuing the evaluation up to $x_2 > x_1$ is also fast.

Examples of first-order ODEs

We will solve the standard test problem $y'(x) = y(x), y(0) = 1$ which has explicit solution $y(x) = \exp(x)$:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> f = odefun(lambda x, y: y, 0, 1)
>>> for x in [0, 1, 2.5]:
...     print((f(x), exp(x)))
...
(1.0, 1.0)
(2.71828182845905, 2.71828182845905)
(12.1824939607035, 12.1824939607035)
```

The solution with high precision:

```
>>> mp.dps = 50
>>> f = odefun(lambda x, y: y, 0, 1)
>>> f(1)
2.7182818284590452353602874713526624977572470937
>>> exp(1)
2.7182818284590452353602874713526624977572470937
```

Using the more general vectorized form, the test problem can be input as (note that f returns a 1-element vector):

```
>>> mp.dps = 15
>>> f = odefun(lambda x, y: [y[0]], 0, [1])
>>> f(1)
[2.71828182845905]
```

`odefun()` can solve nonlinear ODEs, which are generally impossible (and at best difficult) to solve analytically. As an example of a nonlinear ODE, we will solve $y'(x) = x \sin(y(x))$ for

$y(0) = \pi/2$. An exact solution happens to be known for this problem, and is given by $y(x) = 2 \tan^{-1}(\exp(x^2/2))$:

```
>>> f = odefun(lambda x, y: x*sin(y), 0, pi/2)
>>> for x in [2, 5, 10]:
... print((f(x), 2*atan(exp(mpf(x)**2/2))))
...
(2.87255666284091, 2.87255666284091)
(3.14158520028345, 3.14158520028345)
(3.14159265358979, 3.14159265358979)
```

If F is independent of y , an ODE can be solved using direct integration. We can therefore obtain a reference solution with quad():

```
>>> f = lambda x: (1+x**2)/(1+x**3)
>>> g = odefun(lambda x, y: f(x), pi, 0)
>>> g(2*pi)
0.72128263801696
>>> quad(f, [pi, 2*pi])
0.72128263801696
```

Examples of second-order ODEs

We will solve the harmonic oscillator equation $y''(x) + y(x) = 0$. To do this, we introduce the helper functions $y_0 = y, y_1 = y'_0$ whereby the original equation can be written as $y'_1 + y'_0 = 0$. Put together, we get the first-order, two-dimensional vector ODE

$$y'_0 = y_1; \quad y'_1 = -y_0. \quad (25.1.4)$$

To get a well-defined IVP, we need two initial values. With $y(0) = y_0(0) = 1$ and $-y'(0) = y_1(0) = 0$, the problem will of course be solved by $y(x) = y_0(x) = \cos(x)$ and $y(x) = y_1(x) = \sin(x)$. We check this:

```
>>> f = odefun(lambda x, y: [-y[1], y[0]], 0, [1, 0])
>>> for x in [0, 1, 2.5, 10]:
... nprint(f(x), 15)
... nprint([cos(x), sin(x)], 15)
... print("----")
...
[1.0, 0.0]
[1.0, 0.0]
----
[0.54030230586814, 0.841470984807897]
[0.54030230586814, 0.841470984807897]
----
[-0.801143615546934, 0.598472144103957]
[-0.801143615546934, 0.598472144103957]
----
[-0.839071529076452, -0.54402111088937]
[-0.839071529076452, -0.54402111088937]
----
```

Note that we get both the sine and the cosine solutions simultaneously.

Chapter 26

Function approximation

26.1 Taylor series

Function **taylor**(*f* As *mpNum*, *x* As *mpNum*, *n* As *mpNum*, **Keywords** As String) As *mpNum*

The function **taylor** returns a list of coefficients of a degree-*n* Taylor polynomial around the point *x* of the given function *f*.

Parameters:

f: A one dimensional function

x: A real number.

n: A real number.

Keywords: method=step, tol=eps, direction=0. Many more, see **diff()**

Examples:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> nprint(chop(taylor(sin, 0, 5)))
[0.0, 1.0, 0.0, -0.166667, 0.0, 0.00833333]
```

The coefficients are computed using high-order numerical differentiation. The function must be possible to evaluate to arbitrary precision. See **diff()** for additional details and supported keyword options.

Note that to evaluate the Taylor polynomial as an approximation of *f*, e.g. with **polyval()**, the coefficients must be reversed, and the point of the Taylor expansion must be subtracted from the argument:

```
>>> p = taylor(exp, 2.0, 10)
>>> polyval(p[::-1], 2.5 - 2.0)
12.1824939606092
>>> exp(2.5)
12.1824939607035
```

26.2 Pade approximation

Function **pade(*a* As mpNum, *L* As mpNum, *M* As mpNum) As mpNum**

The function `pade` returns coefficients of a Pade approximation of degree (L, M) to a function

Parameters:

a: A list of at least $L + M + 1$ Taylor coefficients approximating a function $A(x)$

L: An integer, specifying the degree of polynomials P .

M: An integer, specifying the degree of polynomials Q .

Computes a Pade approximation of degree (L, M) to a function. Given at least $L + M + 1$ Taylor coefficients approximating a function $A(x)$, `pade()` returns coefficients of polynomials P, Q satisfying

$$P = \sum_{k=0}^L p_k x^k; \quad Q = \sum_{k=0}^M q_k x^k; \quad Q_0 = 1; \quad A(x)Q(x) = P(x) + O(x^{L+M+1}) \quad (26.2.1)$$

$P(x)/Q(x)$ can provide a good approximation to an analytic function beyond the radius of convergence of its Taylor series (example from G.A. Baker 'Essentials of Pade Approximants' Academic Press, Ch.1A):

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> one = mpf(1)
>>> def f(x):
...     return sqrt((one + 2*x)/(one + x))
...
>>> a = taylor(f, 0, 6)
>>> p, q = pade(a, 3, 3)
>>> x = 10
>>> polyval(p[::-1], x)/polyval(q[::-1], x)
1.38169105566806
>>> f(x)
1.38169855941551
```

26.3 Chebyshev approximation

Function **chebyfit**(*f* As *mpNum*, *interval* As *mpNum*, *N* As *mpNum*, **Keywords** As *String*) As *mpNum*

The function **chebyfit** returns coefficients of a polynomial of degree $N - 1$ that approximates the given function *f* on the interval $[a, b]$

Parameters:

f: A one dimensional function

interval: A real interval.

N: An integer.

Keywords: error=False

Computes a polynomial of degree $N - 1$ that approximates the given function *f* on the interval $[a, b]$. With error=True, **chebyfit()** also returns an accurate estimate of the maximum absolute error; that is, the maximum value of $|f(x) - P(x)|$ for $x \in [a, b]$.

chebyfit() uses the Chebyshev approximation formula, which gives a nearly optimal solution: that is, the maximum error of the approximating polynomial is very close to the smallest possible for any polynomial of the same degree.

Chebyshev approximation is very useful if one needs repeated evaluation of an expensive function, such as function defined implicitly by an integral or a differential equation. (For example, it could be used to turn a slow **mpFormulaPy** function into a fast machine-precision version of the same.)

Examples

Here we use **chebyfit()** to generate a low-degree approximation of $f(x) = \cos(x)$, valid on the interval $[1, 2]$:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> poly, err = chebyfit(cos, [1, 2], 5, error=True)
>>> nprint(poly)
[0.00291682, 0.146166, -0.732491, 0.174141, 0.949553]
>>> nprint(err, 12)
1.61351758081e-5
```

The polynomial can be evaluated using **polyval**:

```
>>> nprint(polyval(poly, 1.6), 12)
-0.0291858904138
>>> nprint(cos(1.6), 12)
-0.0291995223013
```

Sampling the true error at 1000 points shows that the error estimate generated by **chebyfit** is remarkably good:

```
>>> error = lambda x: abs(cos(x) - polyval(poly, x))
>>> nprint(max([error(1+n/1000.) for n in range(1000)]), 12)
1.61349954245e-5
```

Choice of degree

The degree N can be set arbitrarily high, to obtain an arbitrarily good approximation. As a rule of thumb, an N -term Chebyshev approximation is good to $N/(b - a)$ decimal places on a unit interval (although this depends on how well-behaved f is). The cost grows accordingly: `chebyfit` evaluates the function $(N^2)/2$ times to compute the coefficients and an additional N times to estimate the error.

Possible issues

One should be careful to use a sufficiently high working precision both when calling `chebyfit` and when evaluating the resulting polynomial, as the polynomial is sometimes ill-conditioned. It is for example difficult to reach 15-digit accuracy when evaluating the polynomial using machine precision floats, no matter the theoretical accuracy of the polynomial. (The option to return the coefficients in Chebyshev form should be made available in the future.)

It is important to note the Chebyshev approximation works poorly if f is not smooth. A function containing singularities, rapid oscillation, etc can be approximated more effectively by multiplying it by a weight function that cancels out the nonsmooth features, or by dividing the interval into several segments.

26.4 Fourier series

Function **fourier(*f* As mpNum, *interval* As mpNum, *N* As mpNum) As mpNum**

The function **fourier** returns two lists of coefficients of the Fourier series of degree N of the given function on the interval $[a, b]$.

Parameters:

f: A one dimensional function

interval: A real interval.

N: An integer.

Computes the Fourier series of degree N of the given function on the interval $[a, b]$. More precisely, **fourier()** returns two lists (c, s) of coefficients (the cosine series and sine series, respectively), such that

$$f(x) \sim \sum_{k=0}^N c_k \cos(kmx) + s_k \sin(kmx) \quad (26.4.1)$$

where $m = 2\pi/(b - a)$.

Note that many texts define the first coefficient as $2c_0$ instead of c_0 . The easiest way to evaluate the computed series correctly is to pass it to **fourierval()**.

Examples

The function $f(x) = x$ has a simple Fourier series on the standard interval $[-\pi, \pi]$. The cosine coefficients are all zero (because the function has odd symmetry), and the sine coefficients are rational numbers:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> c, s = fourier(lambda x: x, [-pi, pi], 5)
>>> nprint(c)
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> nprint(s)
[0.0, 2.0, -1.0, 0.666667, -0.5, 0.4]
```

This computes a Fourier series of a nonsymmetric function on a nonstandard interval:

```
>>> I = [-1, 1.5]
>>> f = lambda x: x**2 - 4*x + 1
>>> cs = fourier(f, I, 4)
>>> nprint(cs[0])
[0.583333, 1.12479, -1.27552, 0.904708, -0.441296]
>>> nprint(cs[1])
[0.0, -2.6255, 0.580905, 0.219974, -0.540057]
```

It is instructive to plot a function along with its truncated Fourier series:

```
>>> plot([f, lambda x: fourierval(cs, I, x)], I)
```

Fourier series generally converge slowly (and may not converge pointwise). For example, if $f(x) = \cosh(x)$, a 10-term Fourier series gives an L^2 error corresponding to 2-digit accuracy:

```
>>> I = [-1, 1]
```

```
>>> cs = fourier(cosh, I, 9)
>>> g = lambda x: (cosh(x) - fourierval(cs, I, x))**2
>>> nprint(sqrt(quad(g, I)))
0.00467963
```

`fourier()` uses numerical quadrature. For nonsmooth functions, the accuracy (and speed) can be improved by including all singular points in the interval specification:

```
>>> nprint(fourier(abs, [-1, 1], 0), 10)
([0.5000441648], [0.0])
>>> nprint(fourier(abs, [-1, 0, 1], 0), 10)
([0.5], [0.0])
```

Function **fouriereval**(*series* As *mpNum*, *interval* As *mpNum*, *x* As *mpNum*) As *mpNum*

The function `fouriereval` returns the result of the evaluation of a Fourier series (in the format computed by `fourier()` for the given interval) at the point *x*.

Parameters:

series: a pair (c, s) where *c* is the cosine series and *s* is the sine series

interval: A real interval.

x: A real number.

The series should be a pair (c, s) where *c* is the cosine series and *s* is the sine series. The two lists need not have the same length.

Chapter 27

Number identification

Most functions in mpFormulaPy are concerned with producing approximations from exact mathematical formulas. It is also useful to consider the inverse problem: given only a decimal approximation for a number, such as 0.7320508075688772935274463, is it possible to find an exact formula?

Subject to certain restrictions, such 'reverse engineering' is indeed possible thanks to the existence of integer relation algorithms. Mpmath implements the PSLQ algorithm (developed by H. Ferguson), which is one such algorithm.

Automated number recognition based on PSLQ is not a silver bullet. Any occurring transcendental constants (π , e , etc) must be guessed by the user, and the relation between those constants in the formula must be linear (such as $x = 3\pi + 4e$). More complex formulas can be found by combining PSLQ with functional transformations; however, this is only feasible to a limited extent since the computation time grows exponentially with the number of operations that need to be combined.

The number identification facilities in mpFormulaPy are inspired by the Inverse Symbolic Calculator (ISC). The ISC is more powerful than mpFormulaPy, as it uses a lookup table of millions of precomputed constants (thereby mitigating the problem with exponential complexity).

27.1 Constant recognition

Function **identify**(*x* As *mpNum*, **constants** As String, **Keywords** As String) As String

The function `identify` returns the result of an attempt to find an exact formula for a given real number *x*

Parameters:

x: A real number

constants: A list of known constants.

Keywords: `tol=None`, `maxcoeff=1000`, `full=False`, `verbose=False`

Given a real number *x*, `identify(x)` attempts to find an exact formula for *x*. This formula is returned as a string. If no match is found, `None` is returned. With `full=True`, a list of matching formulas is returned.

As a simple example, `identify()` will find an algebraic formula for the golden ratio:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
```

```
>>> identify(phi)
'((1+sqrt(5))/2)'
```

identify() can identify simple algebraic numbers and simple combinations of given base constants, as well as certain basic transformations thereof. More specifically, identify() looks for the following:

1. Fractions
2. Quadratic algebraic numbers
3. Rational linear combinations of the base constants
4. Any of the above after first transforming x into $f(x)$ where $f(x)$ is $1/x$, \sqrt{x} , x^2 , or $\exp(x)$, either directly or with x or $f(x)$ multiplied or divided by one of the base constants
5. Products of fractional powers of the base constants and small integers

Base constants can be given as a list of strings representing mpFormulaPy expressions (identify() will eval the strings to numerical values and use the original strings for the output), or as a dict of formula:value pairs.

In order not to produce spurious results, identify() should be used with high precision; preferably 50 digits or more.

27.1.1 Examples

Simple identifications can be performed safely at standard precision. Here the default recognition of rational, algebraic, and exp/log of algebraic numbers is demonstrated:

```
>>> mp.dps = 15
>>> identify(0.2222222222222222)
'(2/9)'
>>> identify(1.9662210973805663)
'sqrt(((24+sqrt(48))/8))'
>>> identify(4.1132503787829275)
'exp(sqrt(8)/2)'
>>> identify(0.881373587019543)
'log((2+sqrt(8))/2))'
```

By default, identify() does not recognize π . At standard precision it finds a not too useful approximation. At slightly increased precision, this approximation is no longer accurate enough and identify() more correctly returns None:

```
>>> identify(pi)
'(2**((176/117)*3**((20/117)*5**((35/39)))/(7**((92/117)))'
>>> mp.dps = 30
>>> identify(pi)
>>>
```

Numbers such as π , and simple combinations of user-defined constants, can be identified if they are provided explicitly:

```
>>> identify(3*pi-2*e, ['pi', 'e'])
'(3*pi + (-2)*e)'
```

Here is an example using a dict of constants. Note that the constants need not be 'atomic'; identify() can just as well express the given number in terms of expressions given by formulas:

```
>>> identify(pi+e, {'a':pi+2, 'b':2*e})
'((-2) + 1*a + (1/2)*b)',
```

Next, we attempt some identifications with a set of base constants. It is necessary to increase the precision a bit.

```
>>> mp.dps = 50
>>> base = ['sqrt(2)', 'pi', 'log(2)']
>>> identify(0.25, base)
'(1/4)'
>>> identify(3*pi + 2*sqrt(2) + 5*log(2)/7, base)
'(2*sqrt(2) + 3*pi + (5/7)*log(2))'
>>> identify(exp(pi+2), base)
'exp((2 + 1*pi))'
>>> identify(1/(3+sqrt(2)), base)
'((3/7) + (-1/7)*sqrt(2))'
>>> identify(sqrt(2)/(3*pi+4), base)
'sqrt(2)/(4 + 3*pi)'
>>> identify(5**mpf(1)/3*pi*log(2)**2, base)
'5** (1/3)*pi*log(2)**2'
```

An example of an erroneous solution being found when too low precision is used:

```
>>> mp.dps = 15
>>> identify(1/(3*pi-4*e+sqrt(8)), ['pi', 'e', 'sqrt(2)'])
'((11/25) + (-158/75)*pi + (76/75)*e + (44/15)*sqrt(2))'
>>> mp.dps = 50
>>> identify(1/(3*pi-4*e+sqrt(8)), ['pi', 'e', 'sqrt(2)'])
'1/(3*pi + (-4)*e + 2*sqrt(2))'
```

27.1.2 Finding approximate solutions

The tolerance tol defaults to 3/4 of the working precision. Lowering the tolerance is useful for finding approximate matches. We can for example try to generate approximations for pi:

```
>>> mp.dps = 15
>>> identify(pi, tol=1e-2)
'(22/7)'
>>> identify(pi, tol=1e-3)
'(355/113)'
>>> identify(pi, tol=1e-10)
'(5** (339/269))/(2** (64/269)*3** (13/269)*7** (92/269))'
```

With full=True, and by supplying a few base constants, identify can generate almost endless lists of approximations for any number (the output below has been truncated to show only the first few):

```
>>> for p in identify(pi, ['e', 'catalan'], tol=1e-5, full=True):
...     print(p)
```

```

...
e/log((6 + (-4/3)*e))
(3**3*5*e*catalan**2)/(2*7**2)
sqrt((( -13) + 1*e + 22*catalan))
log((( -6) + 24*e + 4*catalan)/e)
exp(catalan*(( -1/5) + (8/15)*e))
catalan*(6 + (-6)*e + 15*catalan)
sqrt((5 + 26*e + (-3)*catalan))/e
e*sqrt((( -27) + 2*e + 25*catalan))
log((( -1) + (-11)*e + 59*catalan))
((3/20) + (21/20)*e + (3/20)*catalan)
...

```

The numerical values are roughly as close to π as permitted by the specified tolerance:

```

>>> e/log(6-4*e/3)
3.14157719846001
>>> 135*e*catalan**2/98
3.14166950419369
>>> sqrt(e-13+22*catalan)
3.14158000062992
>>> log(24*e-6+4*catalan)-1
3.14158791577159

```

27.1.3 Symbolic processing

The output formula can be evaluated as a Python expression. Note however that if fractions (like '2/3') are present in the formula, Python's eval() may erroneously perform integer division. Note also that the output is not necessarily in the algebraically simplest form:

```

>>> identify(sqrt(2))
'(sqrt(8)/2)'

```

As a solution to both problems, consider using SymPy's sympify() to convert the formula into a symbolic expression. SymPy can be used to pretty-print or further simplify the formula symbolically:

```

>>> from sympy import sympify
>>> sympify(identify(sqrt(2)))
2**(1/2)

```

Sometimes identify() can simplify an expression further than a symbolic algorithm:

```

>>> from sympy import simplify
>>> x = sympify(' -1/(-3/2+(1/2)*5** (1/2))*(3/2-1/2*5** (1/2))** (1/2)')
>>> x
(3/2 - 5** (1/2)/2)** (-1/2)
>>> x = simplify(x)
>>> x
2/(6 - 2*5** (1/2))** (1/2)
>>> mp.dps = 30
>>> x = sympify(identify(x.evalf(30)))

```

```
>>> x
1/2 + 5**(1/2)/2
```

(In fact, this functionality is available directly in SymPy as the function `nsimplify()`, which is essentially a wrapper for `identify()`.)

27.1.4 Miscellaneous issues and limitations

The input x must be a real number. All base constants must be positive real numbers and must not be rationals or rational linear combinations of each other.

The worst-case computation time grows quickly with the number of base constants. Already with 3 or 4 base constants, `identify()` may require several seconds to finish. To search for relations among a large number of constants, you should consider using `pslq()` directly.

The extended transformations are applied to x , not the constants separately. As a result, `identify` will for example be able to recognize $\exp(2\pi i + 3)$ with π given as a base constant, but not $2\exp(\pi) + 3$. It will be able to recognize the latter if $\exp(\pi)$ is given explicitly as a base constant.

27.2 Algebraic identification

Function **findpoly**(*x* As *mpNum*, *n* As Integer, **Keywords** As String) As String

The function `findpoly` returns the coefficients of an integer polynomial P of degree at most n such that $P(x) \sim 0$

Parameters:

x: A real number

n: max degree of polynomial.

Keywords: tol=None, maxcoeff=1000, maxsteps=100, verbose=False

`findpoly(x, n)` returns the coefficients of an integer polynomial P of degree at most n such that $P(x) \sim 0$. If no polynomial having x as a root can be found, `findpoly()` returns `None`.

`findpoly()` works by successively calling `pslq()` with the vectors $[1, x]$, $[1, x, x^2]$, $[1, x, x^2, x^3]$, ..., as input. Keyword arguments given to `findpoly()` are forwarded verbatim to `pslq()`. In particular, you can specify a tolerance for $P(x)$ with `tol` and a maximum permitted coefficient size with `maxcoeff`.

For large values of n , it is recommended to run `findpoly()` at high precision; preferably 50 digits or more.

27.2.1 Examples

By default (degree $n = 1$), `findpoly()` simply finds a linear polynomial with a rational root:

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> findpoly(0.7)
[-10, 7]
```

The generated coefficient list is valid input to `polyval` and `polyroots`:

```
>>> nprint(polyval(findpoly(phi, 2), phi), 1)
-2.0e-16
>>> for r in polyroots(findpoly(phi, 2)):
...     print(r)
...
-0.618033988749895
1.61803398874989
```

Numbers of the form $m + n\sqrt{p}$ for integers (m, n, p) are solutions to quadratic equations. As we find here, $1 + \sqrt{2}$ is a root of the polynomial $x^2 - 2x - 1$:

```
>>> findpoly(1+sqrt(2), 2)
[1, -2, -1]
>>> findroot(lambda x: x**2 - 2*x - 1, 1)
2.4142135623731
```

Despite only containing square roots, the following number results in a polynomial of degree 4:

```
>>> findpoly(sqrt(2)+sqrt(3), 4)
[1, 0, -10, 0, 1]
```

In fact, $x^4 - 10x^2 + 1$ is the minimal polynomial of $r = \sqrt{2} + \sqrt{3}$, meaning that a rational polynomial of lower degree having r as a root does not exist. Given sufficient precision, `findpoly()` will usually find the correct minimal polynomial of a given algebraic number.

27.2.2 Non-algebraic numbers

If `findpoly()` fails to find a polynomial with given coefficient size and tolerance constraints, that means no such polynomial exists.

We can verify that π is not an algebraic number of degree 3 with coefficients less than 1000:

```
>>> mp.dps = 15
>>> findpoly(pi, 3)
>>>
```

It is always possible to find an algebraic approximation of a number using one (or several) of the following methods:

1. Increasing the permitted degree
2. Allowing larger coefficients
3. Reducing the tolerance

One example of each method is shown below:

```
>>> mp.dps = 15
>>> findpoly(pi, 4)
[95, -545, 863, -183, -298]
>>> findpoly(pi, 3, maxcoeff=10000)
[836, -1734, -2658, -457]
>>> findpoly(pi, 3, tol=1e-7)
[-4, 22, -29, -2]
```

It is unknown whether Euler's constant is transcendental (or even irrational). We can use `findpoly()` to check that if it is an algebraic number, its minimal polynomial must have degree at least 7 and a coefficient of magnitude at least 1000000:

```
>>> mp.dps = 200
>>> findpoly(euler, 6, maxcoeff=10**6, tol=1e-100, maxsteps=1000)
>>>
```

Note that the high precision and strict tolerance is necessary for such high-degree runs, since otherwise unwanted low-accuracy approximations will be detected. It may also be necessary to set `maxsteps` high to prevent a premature exit (before the coefficient bound has been reached). Running with `verbose=True` to get an idea what is happening can be useful.

27.3 Integer relations (PSLQ)

Function **pslq**(*x* As *mpNum*, **Keywords** As *String*) As *mpNum*[]

The function **pslq** returns list of integers to approximate a function.

Parameters:

x: A vector of real numbers $x = [x_0, x_1, \dots, x_n]$

Keywords: tol=None, maxcoeff=1000, full=False, verbose=False

Given a vector of real numbers $x = [x_0, x_1, \dots, x_n]$, **pslq**(*x*) uses the PSLQ algorithm to find a list of integers $[c_0, c_1, \dots, c_n]$ such that

$$|c_1x_1 + c_2x_2 + \dots + c_nx_n| < \text{tol} \quad (27.3.1)$$

and such that $\max|c_k| < \text{maxcoeff}$. If no such vector exists, **pslq()** returns None. The tolerance defaults to 3/4 of the working precision.

27.3.1 Examples

Find rational approximations for π :

```
>>> from mpFormulaPy import *
>>> mp.dps = 15; mp.pretty = True
>>> pslq([-1, pi], tol=0.01)
[22, 7]
>>> pslq([-1, pi], tol=0.001)
[355, 113]
>>> mpf(22)/7; mpf(355)/113; +pi
3.14285714285714
3.14159292035398
3.14159265358979
```

Pi is not a rational number with denominator less than 1000:

```
>>> pslq([-1, pi])
>>>
```

To within the standard precision, it can however be approximated by at least one rational number with denominator less than 10^{12} :

```
>>> p, q = pslq([-1, pi], maxcoeff=10**12)
>>> print(p); print(q)
238410049439
75888275702
>>> mpf(p)/q
3.14159265358979
```

The PSLQ algorithm can be applied to long vectors. For example, we can investigate the rational (in)dependence of integer square roots:

```
>>> mp.dps = 30
>>> pslq([sqrt(n) for n in range(2, 5+1)])
>>>
```

```
>>> pslq([sqrt(n) for n in range(2, 6+1)])
>>>
>>> pslq([sqrt(n) for n in range(2, 8+1)])
[2, 0, 0, 0, 0, 0, -1]
```

27.3.2 Machin formulas

A famous formula for π is Machin's,

$$\frac{\pi}{4} = 4 \operatorname{acot} 5 - \operatorname{acot} 239 \quad (27.3.2)$$

There are actually infinitely many formulas of this type. Two others are

$$\frac{\pi}{4} = \operatorname{acot} 1 \quad (27.3.3)$$

$$\frac{\pi}{4} = 12 \operatorname{acot} 49 + 32 \operatorname{acot} 57 + 5 \operatorname{acot} 239 + 12 \operatorname{acot} 110443 \quad (27.3.4)$$

We can easily verify the formulas using the PSLQ algorithm:

```
>>> mp.dps = 30
>>> pslq([pi/4, acot(1)])
[1, -1]
>>> pslq([pi/4, acot(5), acot(239)])
[1, -4, 1]
>>> pslq([pi/4, acot(49), acot(57), acot(239), acot(110443)])
[1, -12, -32, 5, -12]
```

We could try to generate a custom Machin-like formula by running the PSLQ algorithm with a few inverse cotangent values, for example $\operatorname{acot}(2)$, $\operatorname{acot}(3)$... $\operatorname{acot}(10)$. Unfortunately, there is a linear dependence among these values, resulting in only that dependence being detected, with a zero coefficient for π :

```
>>> pslq([pi] + [acot(n) for n in range(2,11)])
[0, 1, -1, 0, 0, 0, -1, 0, 0, 0]
```

We get better luck by removing linearly dependent terms:

```
>>> pslq([pi] + [acot(n) for n in range(2,11) if n not in (3, 5)])
[1, -8, 0, 0, 4, 0, 0, 0]
```

In other words, we found the following formula:

```
>>> 8*acot(2) - 4*acot(7)
3.14159265358979323846264338328
>>> +pi
3.14159265358979323846264338328
```

Part V

Application Examples

Chapter 28

Date, Time and Financial Functions

Reference text [Benninga \(2008\)](#)

Reference text [Benninga \(2010\)](#)

Reference text [Day \(2010\)](#)

<http://stackoverflow.com/questions/16262007/datetime-toodate-time-only>.

<https://github.com/fsprojects/ExcelFinancialFunctions>.

<http://fsprojects.github.io/ExcelFinancialFunctions/>.

<http://ricardocovo.com/2013/01/14/financial-functions-in-net-c/>.

<https://msdn.microsoft.com/en-us/library/microsoft.visualbasic.financial%28v=vs.100%29.aspx>.

<http://luajalla.azurewebsites.net/fscheck-testing-excel-financial-functions/>.

<http://type-nat.ch/>.

<http://www.randombitsofcode.com/implementing-the-excel-yield-function-in-c-net/>.

<http://www.c-sharpcorner.com/uploadfile/shivprasadk/financial-calculation-using-net-part-i/>.

<http://www.c-sharpcorner.com/uploadfile/shivprasadk/financial-calculation-using-net-part-ii/>.

<https://code.msdn.microsoft.com/office/Excel-Financial-functions-6afc7d42>.

28.1 Date and Time: Conversions from Serial Number

Microsoft Excel stores dates as sequential serial numbers so they can be used in calculations. By default, January 1, 1900 is serial number 1, and January 1, 2008 is serial number 39448 because it is 39,448 days after January 1, 1900.

28.1.1 Serial Number to Second

WorksheetFunction.**SECOND**(*Timevalue* As Variant) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.SECOND returns the seconds of a time value. The second is given as an integer in the range 0 (zero) to 59.

Parameter:

Timevalue: The time that contains the seconds you want to find.

Note: Times may be entered as text strings within quotation marks (for example, "6:45 PM"), as decimal numbers (for example, 0.78125, which represents 6:45 PM), or as results of other formulas or functions (for example, TIMEVALUE("6:45 PM")).

28.1.2 Serial Number to Minute

WorksheetFunction.**MINUTE**(*Timevalue* As Variant) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.MINUTE returns the minutes of a time value. The minute is given as an integer, ranging from 0 to 59.

Parameter:

Timevalue: The time that contains the minute you want to find.

Note: Times may be entered as text strings within quotation marks (for example, "6:45 PM"), as decimal numbers (for example, 0.78125, which represents 6:45 PM), or as results of other formulas or functions (for example, TIMEVALUE("6:45 PM")).

28.1.3 Serial Number to Hour

WorksheetFunction.**HOUR**(*Timevalue* As Variant) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.HOUR returns the hour of a time value. The hour is given as an integer, ranging from 0 (12:00 A.M.) to 23 (11:00 P.M.).

Parameter:

Timevalue: The time that contains the hour you want to find.

Note: Times may be entered as text strings within quotation marks (for example, "6:45 PM"), as decimal numbers (for example, 0.78125, which represents 6:45 PM), or as results of other formulas or functions (for example, TIMEVALUE("6:45 PM")).

28.1.4 Serial Number to Day of the Month

WorksheetFunction.**DAY**(*Datevalue* As Date) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.DAY returns the day of a date, represented by a serial number. The day is given as an integer ranging from 1 to 31.

Parameter:

Datevalue: The date of the day you are trying to find.

Note: Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.1.5 Number of days between two dates

WorksheetFunction.**DAYs**(*EndDatevalue* As Date, *StartDatevalue* As Date) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.DAYs returns the number of days between two dates. EndDateValue and StartDateValue are the two dates between which you want to know the number of days.

Parameters:

EndDatevalue: The end of the time interval.

StartDatevalue: The start of the time interval.

28.1.6 Serial Number to Month

WorksheetFunction.**MONTH**(*Datevalue* As Date) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.MONTH returns the month of a date represented by a serial number. The month is given as an integer, ranging from 1 (January) to 12 (December).

Parameter:

Datevalue: The date of the month you are trying to find.

Note: Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.1.7 Serial Number to Year

WorksheetFunction.**YEAR**(*Datevalue* As Date) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.YEAR returns the year corresponding to a date. The year is returned as an integer in the range 1900-9999.

Parameter:

Datevalue: The date of the year you want to find.

Note: Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.1.8 Serial Number to a Day of the Week

WorksheetFunction.**WEEKDAY**(*Datevalue* As Date, *ReturnType* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.WEEKDAY returns the day of the week corresponding to a date. The day is given as an integer, ranging from 1 (Sunday) to 7 (Saturday), by default.

Parameters:

Datevalue: A sequential number that represents the date of the day you are trying to find.

ReturnType: A number that determines the type of return value.

Note: Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.1.9 Serial Number to Calendar Week

WorksheetFunction.**WEEKNUM**(*Datevalue* As Date, *ReturnType* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.WEEKNUM returns the week number of a specific date.

Parameters:

Datevalue: A date within the week.

ReturnType: A number that determines on which day the week begins. The default is 1.

WorksheetFunction.**WEEKNUM-ADD**(*Datevalue* As Date, *ReturnType* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.WEEKNUM-ADD returns the week number of a specific date.

Parameters:

Datevalue: A date within the week.

ReturnType: A number that determines on which day the week begins. The default is 1.

WorksheetFunction.**ISOWEEKNUM**(*Datevalue* As Date, *ReturnType* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.ISOWEEKNUM returns the week number of a specific date.

Parameters:

Datevalue: A date within the week.

ReturnType: A number that determines on which day the week begins. The default is 1.

Returns the week number of a specific date. For example, the week containing January 1 is the first week of the year, and is numbered week 1.

There are two systems used for these functions:

System 1 The week containing January 1 is the first week of the year, and is numbered week 1.

System 2 The week containing the first Thursday of the year is the first week of the year, and is numbered as week 1.

Note: Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.2 Date and Time: Conversions to Serial Number

Microsoft Excel stores dates as sequential serial numbers so they can be used in calculations. By default, January 1, 1900 is serial number 1, and January 1, 2008 is serial number 39448 because it is 39,448 days after January 1, 1900.

28.2.1 Serial Number of a particular Date

WorksheetFunction.**DATE**(*Year* As *mpReal*, *Month* As *mpReal*, *Day* As *mpReal*) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.DATE returns the sequential serial number that represents a particular date.

Parameters:

Year: A number that determines the Year (1900-9999).

Month: A number that determines the Month (1-12).

Day: A number that determines the day of the month (1-31).

The DATE function returns the sequential serial number that represents a particular date. For example, the formula DATE(2008,7,8) returns 39637, the serial number that represents 8th of July, 2008.

The DATE function is most useful in situations where the year, month, and day are supplied by formulas or cell references. For example, you might have a worksheet that contains dates in a format that Excel does not recognize, such as YYYYMMDD. You can use the DATE function in conjunction with other functions to convert the dates to a serial number that Excel recognizes.

28.2.2 Serial Number of Easter Sunday

WorksheetFunction.**EASTERSUNDAY**(*Year* As *mpReal*) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.EASTERSUNDAY returns the date of Easter Sunday in a given year.

Parameter:

Year: an integer between 1583 and 9956 or between 0 and 99, specifying the year.

Example: EASTERSUNDAY(2008) returns the date 23rd March 2008, which is the date of Easter Sunday in 2008.

28.2.3 Date as Text to Serial Number

WorksheetFunction.**DATEVALUE**(*DateText* As *String*) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.DATEVALUE returns a serial number that Excel recognizes as a date

Parameter:

DateText: Text that represents a date in an Excel date format

The DATEVALUE function converts a date that is stored as text to a serial number that Excel recognizes as a date. For example, the formula =DATEVALUE("1/1/2008") returns 39448, the serial number of the date 1/1/2008.

For example, "1/30/2008" or "30-Jan-2008" are text strings within quotation marks that represent dates. Using the default date system in Microsoft Excel for Windows, the DateText argument must represent a date between January 1, 1900 and December 31, 9999.

28.2.4 Serial Number of Months before or after Start Date

WorksheetFunction.EDATE(*StartDate* As Date, *Months* As Integer) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.EDATE returns a serial number that Excel recognizes as a date

Parameters:

StartDate: A date that represents the start date.

Months: The number of months before or after StartDate. A positive value for months yields a future date; a negative value yields a past date.

Returns the serial number that represents the date that is the indicated number of months before or after a specified date (the StartDate). Use EDATE to calculate maturity dates or due dates that fall on the same day of the month as the date of issue.

Note: Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008. Problems can occur if dates are entered as text.

28.2.5 Serial Number of the last day of the months

WorksheetFunction.EOMONTH(*StartDate* As Date, *Months* As Integer) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.EOMONTH returns a serial number that Excel recognizes as a date

Parameters:

StartDate: A date that represents the start date.

Months: The number of months before or after StartDate. A positive value for months yields a future date; a negative value yields a past date.

Returns the serial number for the last day of the month that is the indicated number of months before or after StartDate. Use EOMONTH to calculate maturity dates or due dates that fall on the last day of the month.

Note: Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008. Problems can occur if dates are entered as text.

28.2.6 Serial Number of the current date and time

WorksheetFunction.**NOW()** As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.NOW returns the serial number of the current date and time.

28.2.7 Serial Number of a particular Time

WorksheetFunction.**TIME(*Hour* As mpReal, *Minute* As mpReal, *Second* As mpReal)** As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.TIME returns the decimal number for a particular time. If the cell format was General before the function was entered, the result is formatted as a date.

Parameters:

Hour: A number from 0 (zero) to 32767 representing the hour. Any value greater than 23 will be divided by 24 and the remainder will be treated as the hour value. For example, TIME(27,0,0) = TIME(3,0,0) = .125 or 3:00 AM.

Minute: A number from 0 to 32767 representing the minute. Any value greater than 59 will be converted to hours and minutes. For example, TIME(0,750,0) = TIME(12,30,0) = .520833 or 12:30 PM.

Second: A number from 0 to 32767 representing the second. Any value greater than 59 will be converted to hours, minutes, and seconds. For example, TIME(0,0,2000) = TIME(0,33,22) = .023148 or 12:33:20 AM.

The decimal number returned by TIME is a value ranging from 0 (zero) to 0.99999999, representing the times from 0:00:00 (12:00:00 AM) to 23:59:59 (11:59:59 P.M.).

28.2.8 Time as Text to Serial Number

WorksheetFunction.**TIMEVALUE(*TimeText* As String)** As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.TIMEVALUE returns the decimal number of the time represented by a text string. The decimal number is a value ranging from 0 (zero) to 0.99999999, representing the times from 0:00:00 (12:00:00 AM) to 23:59:59 (11:59:59 P.M.).

Parameter:

TimeText: A text string that represents a time in any one of the Microsoft Excel time formats.

For example, "6:45 PM" and "18:45" text strings within quotation marks that represent time.

28.2.9 Serial Number of today's Date

WorksheetFunction.**TODAY()** As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.TODAY returns the serial number of the current date.

28.2.10 Serial Number of Date +/- n Workdays

WorksheetFunction.**WORKDAY**(*StartDate* As Date, *Days* As Integer, *Holidays* As DateList) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.WORKDAY returns a number that represents a date that is the indicated number of working days before or after a date (the starting date).

Parameters:

StartDate: A date that represents the start date.

Days: The number of nonweekend and nonholiday days before or after StartDate. A positive value for days yields a future date; a negative value yields a past date.

Holidays: An optional list of one or more dates to exclude from the working calendar

Note: Working days exclude weekends and any dates identified as holidays. Use WORKDAY to exclude weekends or holidays when you calculate invoice due dates, expected delivery times, or the number of days of work performed.

To calculate the serial number of the date before or after a specified number of workdays by using parameters to indicate which and how many days are weekend days, use the WORKDAY.INTL function.

Holidays: Optional. An optional list of one or more dates to exclude from the working calendar, such as state and federal holidays and floating holidays. The list can be either a range of cells that contain the dates or an array constant (array: Used to build single formulas that produce multiple results or that operate on a group of arguments that are arranged in rows and columns. An array range shares a common formula; an array constant is a group of constants used as an argument.) of the serial numbers that represent the dates.

28.2.11 Serial Number of Date +/- n Workdays, international

WorksheetFunction.**WORKDAY.INTL**(*StartDate* As Date, *Days* As Integer, *Weekend* As Integer, *Holidays* As DateList) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.WORKDAY.INTL returns a number that represents a date that is the indicated number of working days before or after a date (the starting date).

Parameters:

StartDate: The start date, truncated to integer.

Days: The number of nonweekend and nonholiday days before or after StartDate. A positive value for days yields a future date; a negative value yields a past date. Day-offset is truncated to an integer.

Weekend: Indicates the days of the week that are weekend days and are not considered working days.

Holidays: An optional list of one or more dates to exclude from the working calendar

Note: Weekend parameters indicate which and how many days are weekend days. Weekend days and any days that are specified as holidays are not considered as workdays

Weekend: Optional. Indicates the days of the week that are weekend days and are not considered working days. Weekend is a weekend number or string that specifies when weekends occur.

Weekend string values are seven characters long and each character in the string represents a day of the week, starting with Monday. 1 represents a non-workday and 0 represents a workday. Only the characters 1 and 0 are permitted in the string. 1111111 is an invalid string.

For example, 0000011 would result in a weekend that is Saturday and Sunday.

Holidays shall be a range of cells that contain the dates, or an array constant of the serial values that represent those dates. The ordering of dates or serial values in holidays can be arbitrary.

28.3 Date and Time: Calculations

WorksheetFunction.**DAY360**(*StartDate* As Date, *EndDate* As Date, *Method* As Boolean) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.DAY360 returns the number of days between two dates based on a 360-day year (twelve 30-day months), which is used in some accounting calculations

Parameters:

StartDate: A date that represents the start date.

EndDate: A date that represents the end date

Method: A logical value that specifies whether to use the U.S. or European method in the calculation

Use this function to help compute payments if your accounting system is based on twelve 30-day months

StartDate, *EndDate*: The two dates between which you want to know the number of days. If *StartDate* occurs after *EndDate*, the DAY360 function returns a negative number. Dates should be entered by using the DATE function, or derived from the results of other formulas or functions. For example, use DATE(2008,5,23) to return the 23rd day of May, 2008.

Method:

FALSE or omitted: U.S. (NASD) method. If the starting date is the last day of a month, it becomes equal to the 30th day of the same month. If the ending date is the last day of a month and the starting date is earlier than the 30th day of a month, the ending date becomes equal to the 1st day of the next month; otherwise the ending date becomes equal to the 30th day of the same month.

TRUE: European method. Starting dates and ending dates that occur on the 31st day of a month become equal to the 30th day of the same month.

28.3.1 Number of whole workdays between two dates

WorksheetFunction.**NETWORKDAYS**(*StartDate* As Date, *EndDate* As Date, *Holidays* As DateList) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NETWORKDAYS returns the number of whole working days between *StartDate* and *EndDate*.

Parameters:

StartDate: A date that represents the start date.

EndDate: A date that represents the end date

Holidays: An optional range of one or more dates to exclude from the working calendar, such as state and federal holidays and floating holidays

Returns the number of whole working days between *StartDate* and *EndDate*. Working days exclude weekends and any dates identified in *holidays*. Use NETWORKDAYS to calculate employee benefits that accrue based on the number of days worked during a specific term.

To calculate whole workdays between two dates by using parameters to indicate which and how many days are weekend days, use the NETWORKDAYS.INTL function.

Holidays: Optional. An optional range of one or more dates to exclude from the working calendar, such as state and federal holidays and floating holidays. The list can be either a range of cells that contains the dates or an array constant (array: Used to build single formulas that produce multiple results or that operate on a group of arguments that are arranged in rows and columns. An array range shares a common formula; an array constant is a group of constants used as an argument.) of the serial numbers that represent the dates.

Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.3.2 Number of whole workdays between two dates, international

WorksheetFunction.NETWORKDAYS.INTL(*StartDate* As Date, *EndDate* As Date, *Weekend* As String) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.NETWORKDAYS.INTL returns the number of whole working days between StartDate and EndDate.

Parameters:

StartDate: A date that represents the start date.

EndDate: A date that represents the end date

Weekend: Indicates the days of the week that are weekend days and are not included in the number of whole working days between StartDate and EndDate. Holidays? DateList? An optional range of one or more dates to exclude from the working calendar, such as state and federal holidays and floating holidays

Working days exclude weekends and any dates identified in holidays. Use NETWORKDAYS to calculate employee benefits that accrue based on the number of days worked during a specific term.

Weekend: Optional. Indicates the days of the week that are weekend days and are not included in the number of whole working days between StartDate and EndDate. Weekend is a weekend number or string that specifies when weekends occur. Weekend string values are seven characters long and each character in the string represents a day of the week, starting with Monday. 1 represents a non-workday and 0 represents a workday. Only the characters 1 and 0 are permitted in the string. Using 1111111 will always return 0.

For example, 0000011 would result in a weekend that is Saturday and Sunday.

Holidays: Optional. An optional set of one or more dates that are to be excluded from the working day calendar. holidays shall be a range of cells that contain the dates, or an array constant of the serial values that represent those dates. The ordering of dates or serial values in holidays can be arbitrary.

Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.3.3 Year Fraction representing whole days between 2 Dates

WorksheetFunction.**YEARFRAC**(*StartDate* As Date, *EndDate* As Date, *Basis* As DateList) As
mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.YEARFRAC returns the fraction of the year represented by the number of whole days between two dates.

Parameters:

StartDate: A date that represents the start date.

EndDate: A date that represents the end date

Basis: The type of day count basis to use. Basis Day count basis

Use the YEARFRAC worksheet function to identify the proportion of a whole year's benefits or obligations to assign to a specific term.

StartDate: A date that represents the start date.

EndDate: A date that represents the end date.

Basis:

0 or omitted: US (NASD) 30/360

1: Actual/actual

2: Actual/360

3: Actual/365

4: European 30/360

Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.4 Coupons

The Coupons functions share the following arguments and terminology:

SettlementDate: The security's settlement date. The security settlement date is the date after the issue date when the security is traded to the buyer.

MaturityDate: The security's maturity date. The maturity date is the date when the security expires.

Frequency: The number of coupon payments per year. For annual payments, frequency = 1; for semiannual, frequency = 2; for quarterly, frequency = 4.

Yield: The security's annual yield.

Coupon: Coupon payments per year.

Basis: Optional. The type of day count basis to use:

0 or omitted: US (NASD) 30/360

1: Actual/actual

2: Actual/360

3: Actual/365

4: European 30/360

For example, suppose a 30-year bond is issued on January 1, 2008, and is purchased by a buyer six months later. For example, suppose a 30-year bond is issued on January 1, 2008, and is purchased by a buyer six months later. The issue date would be January 1, 2008, the settlement date would be July 1, 2008, and the maturity date is January 1, 2038, 30 years after the January 1, 2008 issue date.

Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.4.1 Days from Beginning to Settlement Date

WorksheetFunction.COUPDAYBS(*SettlementDate* As Date, *MaturityDate* As Date, *Frequency* As mpReal, *Basis* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.COUPDAYBS returns the number of days from the beginning of the coupon period to the settlement date.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

28.4.2 Days in Coupon Period containing the Settlement Date

WorksheetFunction.**COUPDAYS**(*SettlementDate* As Date, *MaturityDate* As Date, *Frequency* As mpReal, *Basis* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.COUPDAYS returns the number of days in the coupon period that contains the settlement date.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

28.4.3 Days from Settlement Date to next Coupon Date

WorksheetFunction.**COUPDAYSNC**(*SettlementDate* As Date, *MaturityDate* As Date, *Frequency* As mpReal, *Basis* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.COUPDAYSNC returns the number of days from the settlement date to the next coupon date.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

28.4.4 Next Coupon Date after the Settlement Date

WorksheetFunction.**COUPNCD**(*SettlementDate* As Date, *MaturityDate* As Date, *Frequency* As mpReal, *Basis* As Integer) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.COUPNCD returns a number that represents the next coupon date after the settlement date.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

28.4.5 Coupons payable between Settlement and Maturity Date

WorksheetFunction.COUPNUM(**SettlementDate** As Date, **MaturityDate** As Date, **Frequency** As mpReal, **Basis** As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.COUPNUM returns the number of coupons payable between the settlement date and maturity date, rounded up to the nearest whole coupon.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

28.4.6 Previous Coupon Date before the Settlement Date

WorksheetFunction.COUPPCD(**SettlementDate** As Date, **MaturityDate** As Date, **Frequency** As mpReal, **Basis** As Integer) As Date

NOT YET IMPLEMENTED

The function WorksheetFunction.COUPPCD returns a number that represents the previous coupon date before the settlement date.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

28.4.7 Macauley Duration for an assumed par Value of 100

WorksheetFunction.DURATION(**SettlementDate** As Date, **MaturityDate** As Date, **Coupon** As Integer, **Yield** As Integer, **Frequency** As mpReal, **Basis** As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.DURATION returns the Macauley duration for an assumed par value of \$100.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Coupon: Coupon payments per year

Yield: The security's annual yield.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

Duration is defined as the weighted average of the present value of the cash flows and is used as a measure of a bond price's response to changes in yield.

28.4.8 Modified Macauley Duration

WorksheetFunction.**MDURATION**(*SettlementDate* As Date, *MaturityDate* As Date, *Coupon* As Integer, *Yield* As Integer, *Frequency* As mpReal, *Basis* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.MDURATION returns the modified Macauley duration for a security with an assumed par value of \$100.

Parameters:

SettlementDate: The security's settlement date.

MaturityDate: The security's maturity date.

Coupon: Coupon payments per year

Yield: The security's annual yield.

Frequency: The number of coupon payments per year.

Basis: The type of day count basis to use

The Modified Duration is defined as follows:

$$\text{MDURATION} = \text{DURATION} \div \left(1 + \frac{\text{Market yield}}{\text{Coupon payments per year}} \right) \quad (28.4.1)$$

28.5 Securities

The Securities functions share the following arguments and terminology:

IssueDate: The security's issue date.

FirstInterestDate: The security's first interest date.

SettlementDate: The security's settlement date. The security settlement date is the date after the issue date when the security is traded to the buyer.

Rate: The security's annual coupon rate.

Par: The security's par value. If you omit par, ACCRINT uses \$1,000.

Frequency: The number of coupon payments per year. For annual payments, frequency = 1; for semiannual, frequency = 2; for quarterly, frequency = 4.

Basis: Optional. The type of day count basis to use:

0 or omitted: US (NASD) 30/360

1: Actual/actual

2: Actual/360

3: Actual/365

4: European 30/360

CalcMethod: Optional. A logical value that specifies the way to calculate the total accrued interest when the date of settlement is later than the date of *FirstInterest*.

A value of TRUE (1) returns the total accrued interest from issue to settlement.

A value of FALSE (0) returns the accrued interest from *FirstInterest* to settlement. If you do not enter the argument, it defaults to TRUE.

Dates should be entered by using the DATE function, or as results of other formulas or functions. For example, use DATE(2008,5,23) for the 23rd day of May, 2008.

28.5.1 Accrued Interest

WorksheetFunction.**ACCRINT**(*Issue As Date*, *First_interest As Date*, *Settlement As Date*, *Rate As Integer*, *Par As mpReal*, *Frequency As Integer*, *Basis As Integer*, *Calc_method As Boolean*) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.ACCRINT returns the accrued interest for a security that pays periodic interest.

Parameters:

Issue: The security's issue date.

First_interest: The security's first interest date.

Settlement: The security's settlement date.

Rate: The security's annual coupon rate.

Par: The security's par value

Frequency: The number of coupon payments per year

Basis: The type of day count basis to use

Calc_method: The type calculation to use

ACCRINT is calculated as follows:

$$\text{ACCRINT} = \text{par} \times \frac{\text{rate}}{\text{frequency}} \times \sum_{i=1}^{NC} \frac{A_i}{NL_i}, \text{ where} \quad (28.5.1)$$

A_i is the number of accrued days for the i th quasi-coupon period within odd period.

NC is the number of quasi-coupon periods that fit in odd period. If this number contains a fraction, raise it to the next whole number.

NL_i is the normal length in days of the i th quasi-coupon period within odd period.

28.5.2 Accrued Interest at Maturity

WorksheetFunction.**ACCRINTM**(*Issue As Date*, **Settlement As Date**, **Rate As Integer**, **Par As mpReal**, **Basis As Integer**) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.ACCHRINTM returns the accrued interest for a security that pays interest at maturity.

Parameters:

Issue: The security's issue date.

Settlement: The security's settlement date.

Rate: The security's annual coupon rate.

Par: The security's par value

Basis: The type of day count basis to use.

ACCHRINTM is calculated as follows:

$$\text{ACCHRINTM} = \text{par} \times \text{rate} \times \frac{A}{D}, \text{ where} \quad (28.5.2)$$

A is the number of accrued days counted according to a monthly basis. For interest at maturity items, the number of days from the issue date to the maturity date is used.

D is the Annual Year Basis.

28.5.3 Discount Rate for a Security

WorksheetFunction.**DISC**(**Settlement As Date**, **Maturity As Date**, **Pr As Integer**, **Redemption As mpReal**, **Basis As Integer**) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.DISC returns the discount rate for a security.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Pr: The security's price per \$100 face value.

Redemption: The security's redemption value per \$100 face value.

Basis: The type of day count basis to use.

DISC is calculated as follows:

$$\text{DISC} = \frac{\text{redemption} - \text{par}}{\text{par}} \times \frac{B}{DSM}, \text{ where} \quad (28.5.3)$$

B is the number of days in a year, depending on the year basis, and DSM is the number of days between settlement and maturity.

28.5.4 Interest Rate for a fully invested Security

WorksheetFunction.**INTRATE**(*Settlement* As Date, *Maturity* As Date, *Investment* As mpReal, *Redemption* As mpReal, *Basis* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.INTRATE returns the interest rate for a fully invested security.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Investment: The amount invested in the security.

Redemption: The amount to be received at maturity.

Basis: The type of day count basis to use.

INTRATE is calculated as follows:

$$\text{INTRATE} = \frac{\text{redemption} - \text{investment}}{\text{investment}} \times \frac{B}{DSM}, \text{ where} \quad (28.5.4)$$

B is the number of days in a year, depending on the year basis, and DSM is the number of days between settlement and maturity.

28.5.5 Price of a Security having an odd first Period

WorksheetFunction.**ODDFPRICE**(*Settlement* As Date, *Maturity* As Date, *Issue* As Date, *First_Coupon* As Date, *Rate* As mpReal, *Yld* As mpReal, *Redemption* As mpReal, *Frequency* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.ODDFPRICE returns the price per \$100 face value of a security having an odd (short or long) first period.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Issue: The security's maturity date.

First_Coupon: The security's first coupon date.

Rate: The security's interest rate.

Yld: The security's annual yield.

Redemption: The security's redemption value per \$100 face value.

Frequency: The number of coupon payments per year

NOTE: The parameter Basis is missing from this list (Latex issue).

ODDFPRICE is calculated as follows:

Odd short first coupon:

$$\begin{aligned} \text{ODDFPRICE} = & \frac{\text{Redemption}}{(1 + \text{YF})^{N-1+\text{DSC}/E}} + \frac{100 \times \text{RF} \times \text{DFC}/E}{(1 + \text{YF})^{\text{DSC}/E}} \\ & + \sum_{k=2}^N \frac{100 \times \text{RF}}{(1 + \text{YF})^{k-1+\text{DSC}/E}} - 100 \times \text{RF} \times \frac{A}{E} \end{aligned} \quad (28.5.5)$$

A = number of days from the beginning of the coupon period to the settlement date (accrued days).

DSC = number of days from the settlement to the next coupon date.

DFC = number of days from the beginning of the odd first coupon to the first coupon date.

E = number of days in the coupon period.

N = number of coupons payable between the settlement date and the redemption date. (If this number contains a fraction, it is raised to the next whole number.)

Odd long first coupon:

$$\begin{aligned} \text{ODDFPRICE} = & \frac{\text{Redemption}}{(1 + \text{YF})^{N-1+\text{DSC}/E}} + \frac{100 \times \text{RF} \times \text{DFC}/E}{(1 + \text{YF})^{\text{DSC}/E}} \\ & + \sum_{k=2}^N \frac{100 \times \text{RF}}{(1 + \text{YF})^{k-1+\text{DSC}/E}} - 100 \times \text{RF} \times \frac{A}{E} \end{aligned} \quad (28.5.6)$$

A_i = number of days from the beginning of the i th, or last, quasi-coupon period within odd period.

DC_i = number of days from dated date (or issue date) to first quasi-coupon ($i = 1$) or number of days in quasi-coupon ($i = 2, \dots, i = NC$).

DSC = number of days from settlement to next coupon date.

E = number of days in coupon period.

N = number of coupons payable between the first real coupon date and redemption date. (If this number contains a fraction, it is raised to the next whole number.)

NC = number of quasi-coupon periods that fit in odd period. (If this number contains a fraction, it is raised to the next whole number.)

NL_i = normal length in days of the full i th, or last, quasi-coupon period within odd period.

N_q = number of whole quasi-coupon periods between settlement date and first coupon.

28.5.6 Yield of a Security that has an odd first Period

WorksheetFunction.ODDFYIELD(**Settlement** As Date, **Maturity** As Date, **Issue** As Date, **First_Coupon** As Date, **Rate** As mpReal, **Pr** As mpReal, **Redemption** As mpReal, **Frequency** As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.ODDFYIELD returns the yield of a security that has an odd (short or long) first period.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Issue: The security's maturity date.

First_Coupon: The security's first coupon date.

Rate: The security's interest rate.

Pr: The security's price.

Redemption: The security's redemption value per \$100 face value.

Frequency: The number of coupon payments per year

NOTE: The parameter *Basis* is missing from this list (Latex issue).

Returns the yield of a security that has an odd (short or long) first period.

Excel uses an iterative technique to calculate **ODDFYIELD**. This function uses the Newton method based on the formula used for the function **ODDFPRICE**. The yield is changed through 100 iterations until the estimated price with the given yield is close to the price. See **ODDFPRICE** for the formula that **ODDFYIELD** uses.

28.5.7 Price of a Security having an odd last Coupon

WorksheetFunction.ODDLPRICE(*Settlement* As Date, *Maturity* As Date, *Issue* As Date, *Last_interest* As Date, *Rate* As mpReal, *Yld* As mpReal, *Redemption* As mpReal, *Frequency* As Integer) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.ODDLPRICE** returns the price per \$100 face value of a security having an odd (short or long) last coupon period.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Issue: The security's maturity date.

Last_interest: The security's last coupon date.

Rate: The security's interest rate.

Yld: The security's price.

Redemption: The security's redemption value per \$100 face value.

Frequency: The number of coupon payments per year

NOTE: The parameter *Basis* is missing from this list (Latex issue).

Returns the price per \$100 face value of a security having an odd (short or long) last coupon period.

28.5.8 Yield of a Security that has an odd last Period

WorksheetFunction.ODDLYIELD(*Settlement* As Date, *Maturity* As Date, *Last_interest* As Date, *Rate* As mpReal, *Pr* As mpReal, *Redemption* As mpReal, *Frequency* As Integer, *Basis* As Integer) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.ODDLYIELD** returns the yield of a security that has an odd (short or long) last coupon period.

Parameters:*Settlement*: The security's settlement date.*Maturity*: The security's maturity date.*Last_interest*: The security's last coupon date.*Rate*: The security's interest rate.*Pr*: The security's price.*Redemption*: The security's redemption value per \$100 face value.*Frequency*: The number of coupon payments per year*Basis*: The type of day count basis to use.

ODDLYIELD is calculated as follows:

$$\text{ODDLYIELD} = \frac{\text{Redemption} + SDC \times 100RF - \text{par} + SA \times 100RF}{\text{par} + SA \times 100RF} \times \frac{\text{Frequency}}{SDSC} \quad (28.5.7)$$

$$SDC = \sum_{j=1}^{NC} \frac{DC_i}{NL_i}; \quad SDSC = \sum_{j=1}^{NC} \frac{DSC_i}{NL_i}; \quad SA = \sum_{j=1}^{NC} \frac{A_i}{NL_i}; \quad (28.5.8)$$

where:

 A_i = number of accrued days for the ith, or last, quasi-coupon period within odd period counting forward from last interest date before redemption. DC_i = number of days counted in the ith, or last, quasi-coupon period as delimited by the length of the actual coupon period. NC = number of quasi-coupon periods that fit in odd period; if this number contains a fraction it will be raised to the next whole number. NL_i = normal length in days of the ith, or last, quasi-coupon period within odd coupon period.

28.5.9 Price of a Security that pays periodic Interest

WorksheetFunction.PRICE(*Settlement* As Date, *Maturity* As Date, *Rate* As mpReal, *Yld* As mpReal, *Redemption* As mpReal, *Frequency* As Integer, *Basis* As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.PRICE returns the price per \$100 face value of a security that pays periodic interest.

Parameters:*Settlement*: The security's settlement date.*Maturity*: The security's maturity date.*Rate*: The security's interest rate.*Yld*: The security's annual yield.*Redemption*: The security's redemption value per \$100 face value.*Frequency*: The number of coupon payments per year*Basis*: The type of day count basis to use.

Returns the price per \$100 face value of a security that pays periodic interest. PRICE is calculated as follows:

$$\begin{aligned} \text{PRICE} = & \frac{\text{Redemption}}{(1 + \text{YF})^{N-1+\text{DSC}/\text{E}}} \\ & + \sum_{k=2}^N \frac{100 \times \text{RF}}{(1 + \text{YF})^{k-1+\text{DSC}/\text{E}}} - 100 \times \text{RF} \times \frac{A}{E} \end{aligned} \quad (28.5.9)$$

where:

DSC = number of days from settlement to next coupon date.

E = number of days in coupon period in which the settlement date falls.

N = number of coupons payable between settlement date and redemption date.

A = number of days from beginning of coupon period to settlement date.

28.5.10 Price of a discounted Security

WorksheetFunction.**PRICEDISC**(*Settlement As Date*, *Maturity As Date*, *Discount As mpReal*, *Redemption As mpReal*, *Basis As Integer*) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.PRICEDISC returns Returns the price per \$100 face value of a discounted security.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Discount: The security's interest rate.

Redemption: The security's redemption value per \$100 face value.

Basis: The type of day count basis to use.

PRICEDISC is calculated as follows:

$$\text{PRICEDISC} = \text{redemption} - \text{discount} \times \text{redemption} \times \frac{DSM}{B} \quad (28.5.10)$$

where:

B = number of days in year, depending on year basis.

DSM = number of days from settlement to maturity.

28.5.11 Price of a Security that pays Interest at Maturity

WorksheetFunction.**PRICEMAT**(*Settlement As Date*, *Maturity As Date*, *Issue As Date*, *Rate As mpReal*, *Yld As mpReal*, *Basis As Integer*) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.PRICEMAT returns the price per \$100 face value of a security that pays interest at maturity.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Issue: The security's issue date.

Rate: The security's interest rate.

Yld: The security's annual yield.

Basis: The type of day count basis to use.

PRICEMAT is calculated as follows:

$$\text{PRICEMAT} = \frac{100 + (\text{DIM} \times \text{Rate} \times 100/B)}{1 + (\text{DSM} \times \text{yld}/B)} - \frac{A \times \text{Rate} \times 100}{B} \quad (28.5.11)$$

where:

B = number of days in year, depending on year basis.

DSM = number of days from settlement to maturity.

DIM = number of days from issue to maturity.

A = number of days from issue to settlement.

28.5.12 Amount received at Maturity for a fully invested Security

WorksheetFunction.RECEIVED(**Settlement** As Date, **Maturity** As Date, **Investment** As mpReal, **Discount** As mpReal, **Basis** As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.RECEIVED returns the amount received at maturity for a fully invested security.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Investment: The amount invested in the security.

Discount: The security's discount rate.

Basis: The type of day count basis to use.

RECEIVED is calculated as follows:

$$\text{RECEIVED} = \frac{\text{investment}}{1 - (\text{DIM} \times \text{discount}/B)} \quad (28.5.12)$$

where:

B = number of days in year, depending on year basis.

DIM = number of days from issue to maturity.

28.5.13 Yield on a Security that pays periodic Interest

WorksheetFunction.YIELD(**Settlement** As Date, **Maturity** As Date, **Rate** As mpReal, **Pr** As mpReal, **Redemption** As mpReal, **Frequency** As Integer, **Basis** As Integer) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.YIELD returns the yield on a security that pays periodic interest.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Rate: The security's interest rate.

Pr: The security's price per \$100 face value.

Redemption: The security's redemption value per \$100 face value.

Frequency: The number of coupon payments per year

Basis: The type of day count basis to use.

YIELD is calculated as follows:

$$\text{YIELD} = \frac{(\text{Redemption}/100) + \text{RF} - S_1}{S_1} \frac{\text{Frequency} \times E}{\text{DSR}}; \quad S_1 = \frac{\text{Par}}{100} + \frac{A}{E} \times \text{RF} \quad (28.5.13)$$

where:

A = number of days from the beginning of the coupon period to the settlement date (accrued days).

DSR = number of days from the settlement date to the redemption date.

E = number of days in the coupon period.

28.5.14 Annual Yield for a discounted Security

WorksheetFunction.**YIELDDISC**(*Settlement As Date*, *Maturity As Date*, *Pr As mpReal*, *Redemption As mpReal*, *Basis As Integer*) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.YIELDDISC returns the annual yield for a discounted security.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Pr: The security's price per \$100 face value.

Redemption: The security's redemption value per \$100 face value.

Basis: The type of day count basis to use.

Returns the annual yield for a discounted security.

28.5.15 Annual Yield of a Security that pays Interest at Maturity

WorksheetFunction.**YIELDMAT**(*Settlement As Date*, *Maturity As Date*, *Issue As Date*, *Rate As mpReal*, *Pr As mpReal*, *Basis As Integer*) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.YIELDMAT returns the price per \$100 face value of a security that pays interest at maturity.

Parameters:

Settlement: The security's settlement date.

Maturity: The security's maturity date.

Issue: The security's issue date.

Rate: The security's interest rate.

Pr: The security's price per \$100 face value.

Basis: The type of day count basis to use.

Returns the annual yield of a security that pays interest at maturity.

28.6 Treasury Bills

28.6.1 Bond-equivalent Yield for a Treasury bill

WorksheetFunction.**TBILLEQ**(*Settlement* As Date, *Maturity* As Date, *Discount* As mpReal)
As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.TBILLEQ returns the bond-equivalent yield for a Treasury bill.

Parameters:

Settlement: The Treasury bill's settlement date.

Maturity: The Treasury bill's maturity date.

Discount: The Treasury bill's discount rate.

TBILLEQ is calculated as

$$\text{TBILLEQ} = \frac{365 \times \text{Rate}}{360 - \text{Rate} \times \text{DSM}} \quad (28.6.1)$$

where:

DSM = number of days between settlement and maturity computed according to the 360 days per year basis.

28.6.2 Price for a Treasury bill

WorksheetFunction.**TBILLPRICE**(*Settlement* As Date, *Maturity* As Date, *Discount* As mpReal)
As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.TBILLPRICE returns the price per \$100 face value for a Treasury bill.

Parameters:

Settlement: The Treasury bill's settlement date.

Maturity: The Treasury bill's maturity date.

Discount: The Treasury bill's discount rate.

Returns the price per \$100 face value for a Treasury bill. TBILLPRICE is calculated as

$$\text{TBILLPRICE} = 100 \times \left(1 - \frac{\text{Discount} \times \text{DSM}}{360}\right) \quad (28.6.2)$$

where:

DSM = number of days from settlement to maturity, excluding any maturity date that is more than one calendar year after the settlement date.

28.6.3 Yield for a Treasury bill

WorksheetFunction.**TBILLYIELD**(*Settlement* As Date, *Maturity* As Date, *Pr* As mpReal)
As mpReal

NOT YET IMPLEMENTED

The function `WorksheetFunction.TBILLYIELD` returns the yield for a Treasury bill.

Parameters:

Settlement: The Treasury bill's settlement date.

Maturity: The Treasury bill's maturity date.

Pr: The Treasury bill's price per \$face value.

Returns the yield for a Treasury bill. `TBILLYIELD` is calculated as

$$\text{TBILLYIELD} = \frac{100 - \text{Price}}{\text{Price}} \frac{360}{\text{DSM}} \quad (28.6.3)$$

where:

DSM = number of days from settlement to maturity, excluding any maturity date that is more than one calendar year after the settlement date.

28.7 Depreciation Functions

The depreciation functions are used in accounting to calculate the amount of monetary value a fixed asset loses over a period of time. These functions share the following arguments and terminology:

Cost: Initial cost of asset

Salvage: Required. The value at the end of the depreciation (sometimes called the salvage value of the asset). This value can be 0.

Life: The number of periods over which the asset is being depreciated (sometimes called the useful life of the asset).

Period . The period for which you want to calculate the depreciation. Period must use the same units as life.

Factor: Optional. The rate at which the balance declines. If factor is omitted, it is assumed to be 2 (the double-declining balance method).

TD: Total depreciation from prior periods.

28.7.1 Depreciation of an Asset

WorksheetFunction.DDB(***Cost As mpReal, Salvage As mpReal, Life As mpReal, Period As mpReal, Factor As mpReal***) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.DDB returns the depreciation of an asset for a specified period using the double-declining balance method or some other method you specify.

Parameters:

Cost: The initial cost of the asset.

Salvage: The salvage value at the end of the life of the asset.

Life: The number of periods over which the asset is being depreciated.

Period: The period for which you want to calculate the depreciation.

Factor: The rate at which the balance declines.

The double-declining balance method computes depreciation at an accelerated rate. Depreciation is highest in the first period and decreases in successive periods. DDB uses the following formula to calculate depreciation for a period:

$$\text{DDB} = \text{Min}((\text{Cost} - \text{TD}) * (\text{Factor}/\text{Life}), (\text{Cost} - \text{Salvage} - \text{TD})) \quad (28.7.1)$$

Change factor if you do not want to use the double-declining balance method. Use the VDB function if you want to switch to the straight-line depreciation method when depreciation is greater than the declining balance calculation.

28.7.2 Straight-Line Depreciation of an Asset

WorksheetFunction.SLN(***Cost As mpReal, Salvage As mpReal, Life As mpReal***) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.SLN returns the straight-line depreciation of an asset for a single period

Parameters:*Cost*: The initial cost of the asset.*Salvage*: The salvage value at the end of the life of the asset.*Life*: The number of periods over which the asset is being depreciated.

Returns the straight-line depreciation of an asset for a single period

28.7.3 Sum-of-Years' Digits Depreciation of an Asset

WorksheetFunction.SYD(*Cost* As mpReal, *Salvage* As mpReal, *Life* As mpReal, *Period* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.SYD** returns the sum-of-years' digits depreciation of an asset for a specified period.

Parameters:*Cost*: The initial cost of the asset.*Salvage*: The salvage value at the end of the life of the asset.*Life*: The number of periods over which the asset is being depreciated.*Period*: The period for which you want to calculate the depreciation.

SYD is calculated as follows:

$$\text{SYD} = \frac{(\text{Cost} - \text{Salvage}) \times 2(\text{Life} - \text{Period} + 1)}{\text{Life}(\text{Life} + 1)} \quad (28.7.2)$$

28.7.4 Fixed Declining Balance Method

WorksheetFunction.DB(*Cost* As mpReal, *Salvage* As mpReal, *Life* As mpReal, *Period* As mpReal, *Month* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.DB** returns the depreciation of an asset for a specified period using the fixed-declining balance method.

Parameters:*Cost*: The initial cost of the asset.*Salvage*: The salvage value at the end of the life of the asset.*Life*: The number of periods over which the asset is being depreciated.*Period*: The period for which you want to calculate the depreciation.*Month*: The number of months in the first year.

The fixed-declining balance method computes depreciation at a fixed rate. DB uses the following formulas to calculate depreciation for a period:

$$\text{DB} = (\text{Cost} - \text{TD}) \times \text{Rate}, \quad (28.7.3)$$

where $\text{Rate} = 1 - (\text{Salvage}/\text{Cost})^{1/\text{Life}}$, rounded to three decimal places

Depreciation for the first and last periods is a special case.

For the first period, DB uses this formula:

$$DB = Cost \times Rate \times Month/12. \quad (28.7.4)$$

For the last period, DB uses this formula:

$$DB = ((Cost - TD) \times Rate \times (12 - Month))/12. \quad (28.7.5)$$

28.7.5 Variable Declining Balance

WorksheetFunction.VDB(**Cost** As mpReal, **Salvage** As mpReal, **Life** As mpReal, **Start_Period** As mpReal, **End_Period** As mpReal, **Factor** As mpReal, **No_switch** As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.VDB returns the depreciation of an asset for any period you specify, including partial periods, using the double-declining balance method or some other method you specify. VDB stands for variable declining balance.

Parameters:

Cost: The initial cost of the asset.

Salvage: The salvage value at the end of the life of the asset.

Life: The number of periods over which the asset is being depreciated.

Start_Period: The period for which you want to calculate the depreciation.

End_Period: The ending period for which you want to calculate the depreciation. EndPeriod must use the same units as life.

Factor: The rate at which the balance declines.

No_switch: A logical value specifying whether to switch to straight-line depreciation when depreciation is greater than the declining balance calculation.

If NoSwitch is TRUE, Microsoft Excel does not switch to straight-line depreciation even when the depreciation is greater than the declining balance calculation.

If NoSwitch is FALSE or omitted, Excel switches to straight-line depreciation when depreciation is greater than the declining balance calculation.

28.7.6 Depreciation for each accounting period

WorksheetFunction.AMORLINC(**Cost** As mpReal, **Date_Purchased** As Date, **First_Period** As mpReal, **Salvage** As mpReal, **Period** As mpReal, **Rate** As mpReal, **Basis** As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.AMORLINC returns the depreciation for each accounting period.

Parameters:

Cost: The initial cost of the asset.

Date_Purchased: The date the asset is purchased.

First_Period: The date of the end of the first period.

Salvage: The salvage value at the end of the life of the asset.

Period: The period.

Rate: The rate of depreciation.

Basis: Year Basis: 0 for 360 days, 1 for actual, 3 for 365 days.

This function is provided for the French accounting system. If an asset is purchased in the middle of the accounting period, the prorated depreciation is taken into account.

28.7.7 Depreciation using a depreciation coefficient

WorksheetFunction.**AMORDEGRC**(*Cost* As mpReal, *Date_Purchased* As Date, *First_Period* As mpReal, *Salvage* As mpReal, *Period* As mpReal, *Rate* As mpReal, *Basis* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.AMORDEGRC returns the prorated linear depreciation of an asset for each accounting period.

Parameters:

Cost: The initial cost of the asset.

Date_Purchased: The date the asset is purchased.

First_Period: The date of the end of the first period.

Salvage: The salvage value at the end of the life of the asset.

Period: The period.

Rate: The rate of depreciation.

Basis: Year Basis: 0 for 360 days, 1 for actual, 3 for 365 days.

This function is provided for the French accounting system. If an asset is purchased in the middle of the accounting period, the prorated depreciation is taken into account. The function is similar to AMORLINC, except that a depreciation coefficient is applied in the calculation depending on the life of the assets.

28.8 Annuity Functions

An annuity is a series of payments that represents either the return on an investment or the amortization of a loan. Negative numbers represent monies paid out, like contributions to savings or loan payments. Positive numbers represent monies received, like dividends. The Annuity Functions share the following arguments and terminology:

Rate: Interest rate per period, must use the same unit for Period as used for Nper.

Nper: Total number of payment periods in the annuity.

PMT: Payment to be made each period

PV: Present value (or lump sum) that a series of payments to be paid in the future is worth now.

FV: Optional. Value of the annuity after the final payment has been made (if omitted, 0 is assumed, which is the usual future value of a loan).

Type: Optional. Number indicating when payments are due: 0 if payments are due at the end of the payment period and 1 if payments are due at the beginning of the period, if omitted, 0 is assumed.

In general, the routines solve for one financial argument in terms of the others. If rate is not 0, then:

$$PV \times (1 + Rate)^{Nper} + PMT(1 + Rate \times Type) \times \left(\frac{(1 + Rate)^{Nper} - 1}{Rate} \right) + FV = 0. \quad (28.8.1)$$

If *Rate* is 0, then

$$(PMT \times Nper) + PV + FV = 0. \quad (28.8.2)$$

28.8.1 Future Value

WorksheetFunction.**FV**(*Rate* As mpReal, *Nper* As mpReal, *Pmt* As mpReal, *PV* As mpReal, *Type* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.FV returns the future value of an annuity based on periodic fixed payments and a fixed interest rate.

Parameters:

Rate: The the interest rate per period.

Nper: The total number of payment periods in the investment.

Pmt: The payment made each period.

PV: The present value.

Type: a value representing the timing of payment.

The future value is calculated as

$$FV = -PV(1 + r)^n + PMT \left(\frac{(1 + r)^n - 1}{r} \right) \quad (28.8.3)$$

28.8.2 Present Value

WorksheetFunction.**PV**(*Rate* As mpReal, *Nper* As mpReal, *Pmt* As mpReal, *FV* As mpReal, *Type* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function `WorksheetFunction.PV` returns the present value of an annuity based on periodic fixed payments to be paid in the future at a fixed interest rate.

Parameters:

Rate: The interest rate per period.

Nper: The total number of payment periods in the investment.

Pmt: The payment made each period.

FV: The future value.

Type: a value representing the timing of payment.

The present value is the total amount that a series of future payments is worth now. For example, when you borrow money, the loan amount is the present value to the lender.

The present value is calculated as

$$PV = - \left(FV + PMT \left(\frac{(1 + r)^n - 1}{r} \right) \right) (1 + r)^{-n} \quad (28.8.4)$$

28.8.3 Payment

`WorksheetFunction.PMT(Rate As mpReal, Nper As mpReal, PV As mpReal, FV As mpReal, Type As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.PMT` returns the payment for a loan based on constant payments and a constant interest rate.

Parameters:

Rate: The interest rate per period.

Nper: The total number of payment periods in the investment.

PV: The present value.

FV: The future value.

Type: a value representing the timing of payment.

The payment is calculated as

$$PMT = - (FV + PV(1 + r)^n) \times \left(\frac{r}{(1 + r)^n - 1} \right) \quad (28.8.5)$$

28.8.4 Number of periods

`WorksheetFunction.NPER(Rate As mpReal, Pmt As mpReal, PV As mpReal, FV As mpReal, Type As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.NPER` returns the number of periods for an investment based on periodic, constant payments and a constant interest rate.

Parameters:

Rate: The interest rate per period.

Pmt: The made each period.

PV: The present value.

FV: The future value.

Type: a value representing the timing of payment.

The number of periods is calculated as

$$n = \frac{1}{\ln(1+r)} \ln \left(\frac{(PMT/r) - FV}{(PMT/r) + PV} \right) \quad (28.8.6)$$

28.8.5 Number of periods required

WorksheetFunction.**PDURATION**(*Rate* As mpReal, *PV* As mpReal, *FV* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.PDURATION returns the number of periods required by an investment to reach a specified value.

Parameters:

Rate: The interest rate per period.

PV: The present value.

FV: The future value.

PDURATION is calculated as

$$PDURATION = \frac{\ln(FV) - \ln(PV)}{\ln(1+r)} \quad (28.8.7)$$

28.8.6 Interest Rate

WorksheetFunction.**RATE**(*Nper* As mpReal, *Pmt* As mpReal, *PV* As mpReal, *FV* As mpReal, *Type* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.RATE returns the interest rate per period of an annuity

Parameters:

Nper: The the interest rate per period.

Pmt: The made each period.

PV: The present value.

FV: The future value.

Type: a value representing the timing of payment.

RATE is calculated by iteration and can have zero or more solutions. If the successive results of RATE do not converge to within 0.0000001 after 20 iterations, RATE returns the #NUM! error value.

An iterative scheme is used to solve

$$f(r) = FV + PV(1+r)^n + PMT \left(\frac{(1+r)^n - 1}{r} \right) = 0 \quad (28.8.8)$$

28.8.7 Interest Payment

WorksheetFunction.**IPMT**(**Rate** As mpReal, **Per** As mpReal, **Nper** As mpReal, **Pmt** As mpReal, **FV** As mpReal, **Type** As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.IPMT returns the interest payment for a given period for an investment based on periodic, constant payments and a constant interest rate.

Parameters:

Rate: The interest rate per period.

Per: The period for which you want to find the interest and must be in the range 1 to NPer

Nper: The total number of payment periods in the investment.

Pmt: The payment made each period.

FV: The future value.

Type: a value representing the timing of payment.

The Interest Payment is calculated as

$$IPMT = - \left((1 + r)^{i-1} (PMT + PV \times r) \right) \quad (28.8.9)$$

28.8.8 Principal Payment

WorksheetFunction.**PPMT**(**Rate** As mpReal, **Per** As mpReal, **Nper** As mpReal, **PV** As mpReal, **FV** As mpReal, **Type** As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.PPMT returns the payment on the principal for a given period for an investment based on periodic, constant payments and a constant interest rate.

Parameters:

Rate: The the interest rate per period.

Per: The period for which you want to find the interest and must be in the range 1 to NPer

Nper: The total number of payment periods in the investment.

PV: The payment made each period.

FV: The future value.

Type: a value representing the timing of payment.

Returns the payment on the principal for a given period for an investment based on periodic, constant payments and a constant interest rate.

The Principal Payment is calculated as

$$PPMT = PMT - IPMT \quad (28.8.10)$$

28.8.9 Cumulative Interest Paid

WorksheetFunction.**CUMIPMT**(**Rate** As mpReal, **Nper** As mpReal, **PV** As mpReal, **StartPeriod** As mpReal, **EndPeriod** As mpReal, **Type** As mpReal) As mpReal

NOT YET IMPLEMENTED

The function `WorksheetFunction.CUMIPMT` returns the cumulative interest paid on a loan between `StartPeriod` and `EndPeriod`.

Parameters:

`Rate`: The interest rate per period.

`Nper`: The total number of payment periods in the investment.

`PV`: The payment made each period.

`StartPeriod`: The first period in the calculation.

`EndPeriod`: the last period in the calculation

`Type`: a value representing the timing of payment.

28.8.10 Cumulative Principal Paid

`WorksheetFunction.CUMPRINC(Rate As mpReal, Nper As mpReal, PV As mpReal, StartPeriod As mpReal, EndPeriod As mpReal, Type As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.CUMPRINC` returns the effective annual interest rate, given the nominal annual interest rate and the number of compounding periods per year.

Parameters:

`Rate`: The interest rate per period.

`Nper`: The total number of payment periods in the investment.

`PV`: The payment made each period.

`StartPeriod`: The first period in the calculation.

`EndPeriod`: the last period in the calculation

`Type`: a value representing the timing of payment.

28.8.11 Effective Annual Interest Rate

`WorksheetFunction.EFFECT(NominalRate As mpReal, Npery As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.EFFECT` returns the effective annual interest rate.

Parameters:

`NominalRate`: The nominal interest rate per period.

`Npery`: The number of compounding periods per year

28.8.12 Nominal Annual Interest Rate

`WorksheetFunction.NOMINAL(EffectiveRate As mpReal, Npery As mpReal) As mpReal`

NOT YET IMPLEMENTED

The function `WorksheetFunction.NOMINAL` returns the nominal annual interest rate, given the effective rate and the number of compounding periods per year.

Parameters:

`EffectiveRate`: The nominal interest rate per period.

`Npery`: The number of compounding periods per year

The relationship between NOMINAL and EFFECT is shown in the following equation:

$$EFFECT = \left(1 + \frac{NominalRate}{Nperry}\right)^{Nperry} - 1. \quad (28.8.11)$$

28.8.13 FV Schedule, variable Compound Interest Rates

WorksheetFunction.**FVSCHEDULE**(*Principal* As mpReal, *Schedule* As mpNum) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.FVSCHEDULE returns the future value of an initial principal after applying a series of compound interest rates. Use FVSCHEDULE to calculate the future value of an investment with a variable or adjustable rate.

Parameters:

Principal: The present value.

Schedule: An array of interest rates to apply

28.8.14 Interest paid during a specific Period of an Investment

WorksheetFunction.**ISPMT**(*Rate* As mpReal, *Per* As mpReal, *Nper* As mpReal, *PV* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.ISPMT returns the interest paid during a specific period of an investment.

Parameters:

Rate: The interest rate per period.

Per: The period for which you want to find the interest and must be in the range 1 to NPer

Nper: The total number of payment periods in the investment.

PV: The present value.

28.9 Cash-Flow Functions

The cash-flow functions perform financial calculations based on a series of periodic payments and receipts. As with the annuity functions, negative numbers represent payments and positive numbers represent receipts. However, unlike the annuity functions, the cash-flow functions allow to list varying amounts for the payments or receipts over the course of a loan or investment. Payments and receipts can even be mixed up within the cash-flow series. The cash-flow functions share the following arguments and terminology:

Values(): array of cash-flow values; the array must contain at least one negative value (a payment) and one positive value (a receipt).

Rate: Discount rate over the length of the period, expressed as a decimal.

FinanceRate: Interest rate paid as the cost of financing.

ReinvestRate: Interest rate received on gains from cash reinvestment.

[*Guess*]: Optional value as estimate of return; if omitted, Guess is 0.1 (10 percent).

28.9.1 Internal Rate of Return

WorksheetFunction.IRR(*Values* As mpNum, *Guess* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.IRR returns the effective annual interest rate.

Parameters:

Values: An array which contains numbers for which the internal rate of return is calculated.

Guess: An initial guess for the IRR, 0.1 if omitted

Returns the internal rate of return for a series of cash flows represented by the numbers in values. These cash flows do not have to be even, as they would be for an annuity. However, the cash flows must occur at regular intervals, such as monthly or annually. The internal rate of return is the interest rate received for an investment consisting of payments (negative values) and income (positive values) that occur at regular periods.

Microsoft Excel uses an iterative technique for calculating IRR. Starting with guess, IRR cycles through the calculation until the result is accurate within 0.00001 percent. If IRR can't find a result that works after 20 tries, the #NUM! error value is returned.

In most cases you do not need to provide guess for the IRR calculation. If guess is omitted, it is assumed to be 0.1 (10 percent).

IRR is closely related to NPV, the net present value function. The rate of return calculated by IRR is the interest rate corresponding to a 0 (zero) net present value.

28.9.2 Calc: Rate of Return

WorksheetFunction.RRI(*Nper* As mpReal, *PV* As mpReal, *FV* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.RRI returns an equivalent interest rate for the growth of an investment

Parameters:

Nper: The total number of periods for the investment.

PV: The present value for the investment.

PV: The future value for the investment.

RRI returns the interest rate given nper (the number of periods), pv (present value), and fv (future value), calculated by using the following equation:

$$RRI = \frac{FV^{1/N_{per}}}{PV} - 1. \quad (28.9.1)$$

28.9.3 Modified Internal Rate of Return

WorksheetFunction.MIRR(**Values** As *mpNum[]*, **FinanceRate** As *mpReal*, **ReinvestRate** As *mpReal*) As *mpReal*

NOT YET IMPLEMENTED

The function WorksheetFunction.MIRR returns the modified internal rate of return for a series of periodic cash flows. MIRR considers both the cost of the investment and the interest received on reinvestment of cash.

Parameters:

Values: An array that contains numbers that represent a series of payments (negative) and income (positive) at regular periods.

FinanceRate: The interest rate paid on the money used in the cash flows.

ReinvestRate: The interest rate received on the money used in the cash flows.

MIRR uses the order of values to interpret the order of cash flows. Be sure to enter your payment and income values in the sequence you want and with the correct signs (positive values for cash received, negative values for cash paid).

If n is the number of cash flows in values, frate is the FinanceRate, and rrate is the ReinvestRate, then the formula for MIRR is:

$$MIRR = \left(\frac{-NPV(rrate, values[positive]) \times (1 + rrate)^n}{NPV(frate, values[negative]) \times (1 + frate)} \right)^{1/(n-1)} - 1. \quad (28.9.2)$$

28.9.4 Net Present Value

WorksheetFunction.NPV(**Rate** As *mpReal*, **Values** As *mpNum[]*) As *mpReal*

NOT YET IMPLEMENTED

The function WorksheetFunction.NPV returns the net present value of an investment based on a series of periodic cash flows and a discount rate.

Parameters:

Rate: The total number of periods for the investment.

Values: An array that contains numbers that represent a series of payments (negative) and income (positive) at regular periods.

Calculates the net present value of an investment by using a discount rate and a series of future payments (negative values) and income (positive values).

The NPV investment begins one period before the date of the value1 cash flow and ends with the last cash flow in the list. The NPV calculation is based on future cash flows. If your first cash flow occurs at the beginning of the first period, the first value must be added to the NPV result, not included in the values arguments. For more information, see the examples below.

If n is the number of cash flows in the list of values, the formula for NPV is:

$$NPV = \sum_{i=1}^n \frac{values_i}{(1 + rate)^i} \quad (28.9.3)$$

NPV is similar to the PV function (present value). The primary difference between PV and NPV is that PV allows cash flows to begin either at the end or at the beginning of the period. Unlike the variable NPV cash flow values, PV cash flows must be constant throughout the investment. For information about annuities and financial functions, see PV.

NPV is also related to the IRR function (internal rate of return). IRR is the rate for which NPV equals zero: $NPV(IRR(...), ...) = 0$.

28.9.5 Internal Rate of Return, non-periodic Schedule

WorksheetFunction.XIRR(*Values* As mpNum[], *Dates* As mpReal, *Guess* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.XIRR** returns the modified internal rate of return for a series of periodic cash flows. MIRR considers both the cost of the investment and the interest received on reinvestment of cash.

Parameters:

Values: An array that contains cash flows that correspond to a schedule of payments in *Dates*.

Dates: A schedule of payment dates that correspond to the cash flow payments

Guess: An initial guess for XIRR.

Returns the internal rate of return for a schedule of cash flows that is not necessarily periodic. To calculate the internal rate of return for a series of periodic cash flows, use the IRR function. XIRR is closely related to XNPV, the net present value function. The rate of return calculated by XIRR is the interest rate corresponding to $XNPV = 0$.

Excel uses an iterative technique for calculating XIRR. Using a changing rate (starting with guess), XIRR cycles through the calculation until the result is accurate within 0.000001 percent. If XIRR can't find a result that works after 100 tries, the #NUM! error value is returned. The rate is changed until:

$$\sum_{i=1}^N \frac{P_i}{(1 + rate)^{(d_i - d_1)/365}} = 0, \quad (28.9.4)$$

where:

d_i = the i th, or last, payment date,

d_1 = the 0th payment date,

P_i = the i th, or last, payment.

28.9.6 Net Present Value, non-periodic Schedule

WorksheetFunction.XNPV(*Rate* As mpReal, *Values* As mpNum[], *Dates* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function **WorksheetFunction.XNPV** returns the modified internal rate of return for a series of periodic cash flows. MIRR considers both the cost of the investment and the interest received on reinvestment of cash.

Parameters:

Rate: The discount rate to apply to the cash flows.

Values: An array that contains cash flows that correspond to a schedule of payments in *Dates*.

Dates: A schedule of payment dates that correspond to the cash flow payments

Returns the net present value for a schedule of cash flows that is not necessarily periodic. To calculate the net present value for a series of cash flows that is periodic, use the NPV function. XNPV is calculated as follows:

$$XNPV = \sum_{i=1}^N \frac{P_i}{(1 + rate)^{(d_i - d_1)/365}}, \quad (28.9.5)$$

where:

d_i = the ith, or last, payment date,

d_1 = the 0th payment date,

P_i = the ith, or last, payment.

28.10 Conversion

28.10.1 Price as a fraction into a price as decimal

WorksheetFunction.**DOLLARDE**(*FractionalDollar* As mpReal, *Fraction* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.DOLLARDE returns a dollar price expressed as a decimal number, converted from a dollar price expressed as an integer part and a fraction part.

Parameters:

FractionalDollar: A number expressed as a fraction.

Fraction: The integer to use in the denominator of the fraction

Converts a dollar price expressed as an integer part and a fraction part, such as 1.02, into a dollar price expressed as a decimal number. Fractional dollar numbers are sometimes used for security prices.

The fraction part of the value is divided by an integer that you specify. For example, if you want your price to be expressed to a precision of 1/16 of a dollar, you divide the fraction part by 16. In this case, 1.02 represents \$1.125 ($\$1 + 2/16 = \1.125).

28.10.2 Price as a decimal into a price as fraction

WorksheetFunction.**DOLLARFR**(*DecimalDollar* As mpReal, *Fraction* As mpReal) As mpReal

NOT YET IMPLEMENTED

The function WorksheetFunction.DOLLARFR returns a dollar price expressed as a fraction, converted from a dollar price expressed as a decimal number.

Parameters:

DecimalDollar: A decimal number.

Fraction: The integer to use in the denominator of the fraction

Converts a dollar price expressed as a decimal number into a dollar price expressed as a fraction. Use DOLLARFR to convert decimal numbers to fractional dollar numbers, such as securities prices.

Part VI

Appendices

Appendix A

Python

A.1 Overview

Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C. The language provides constructs intended to enable clear programs on both a small and large scale.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.

Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts. Using third-party tools, Python code can be packaged into standalone executable programs. Python interpreters are available for many operating systems.

A.2 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.

A.2.1 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.

A.2.2 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.

A.2.2.1 Function curve plots

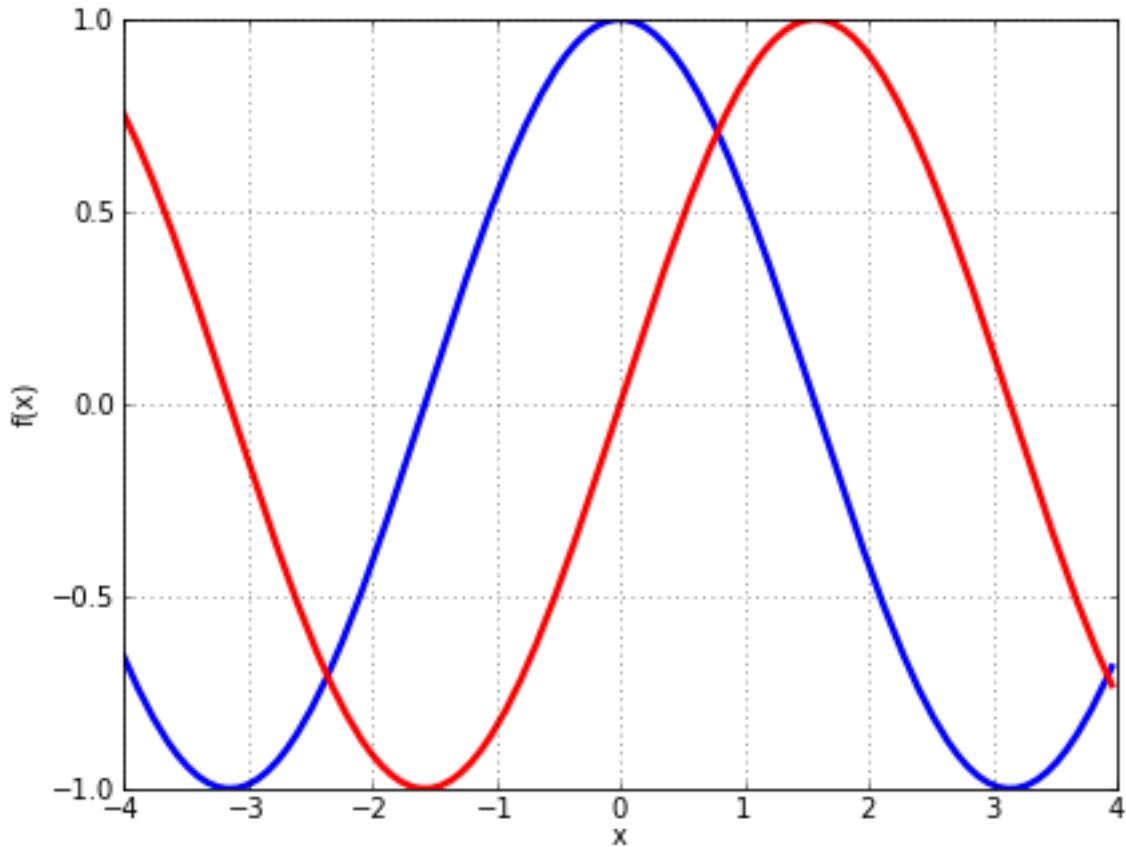


Figure A.1: MpMath 2Dplot

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).

A.2.2.2 Complex function plots

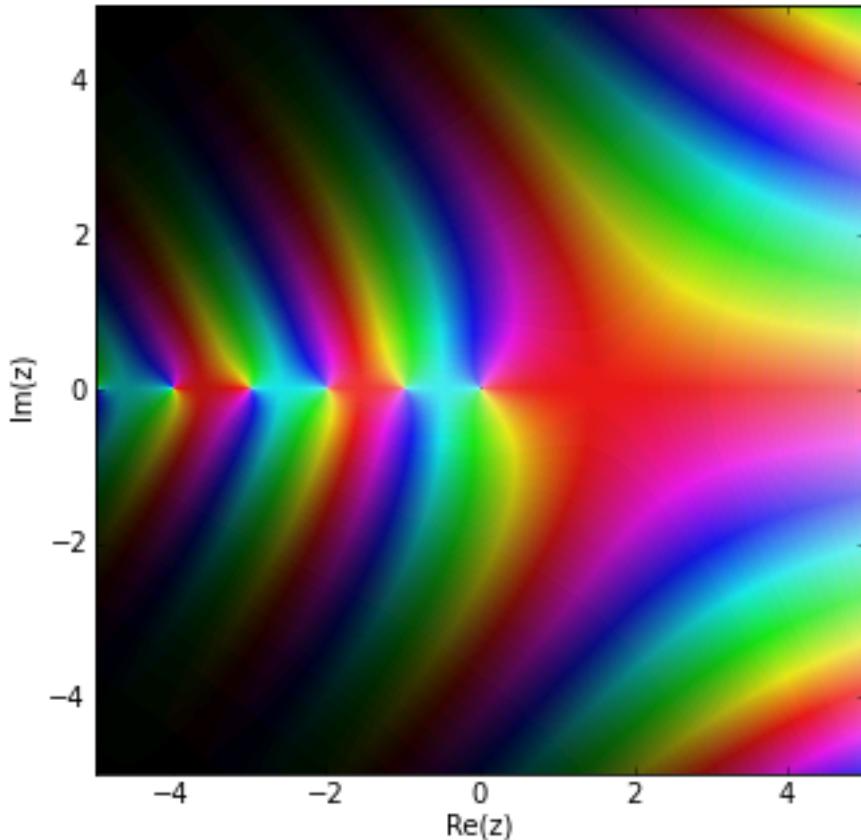


Figure A.2: MpMath cplot

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)`.

A.2.2.3 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)
```

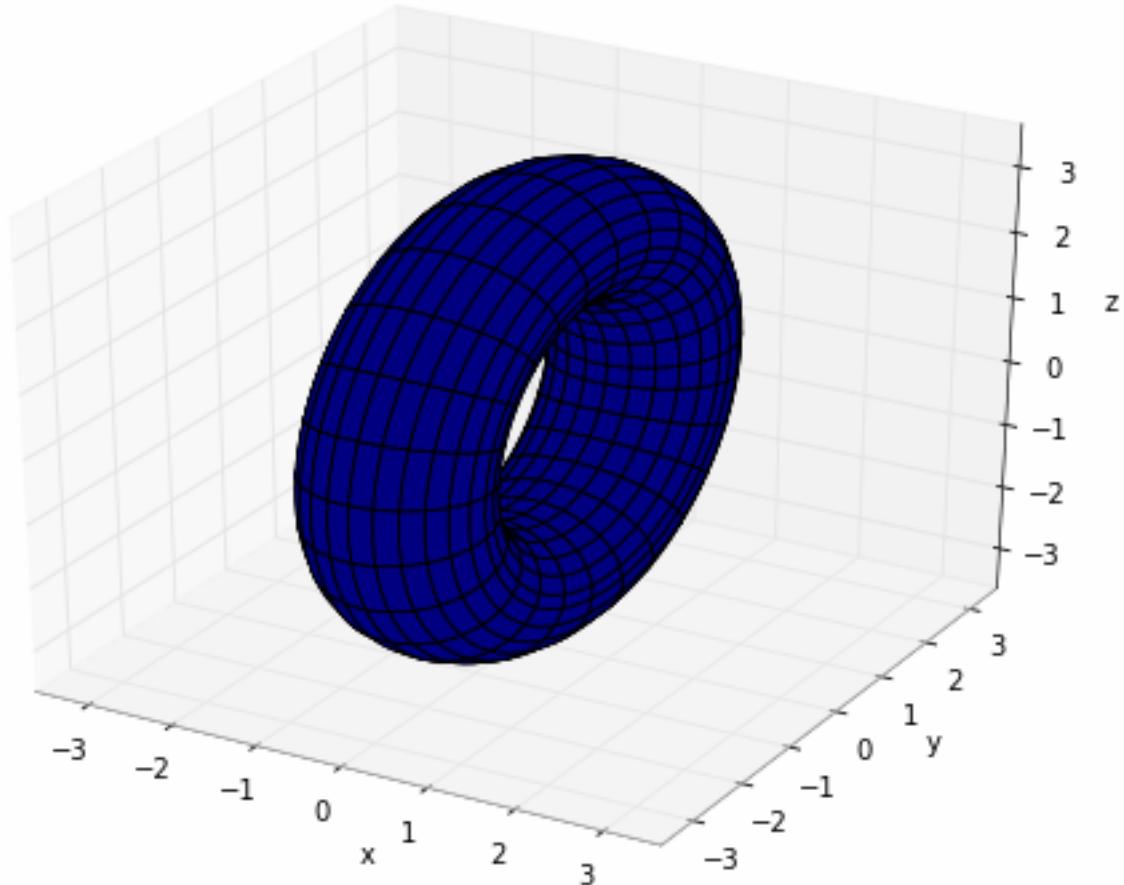


Figure A.3: MpMath Surface plot

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.

A.2.3 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 ??).

A.2.4 Interfaces to the C family of languages

A.2.4.1 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](#).

A.2.4.2 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();
}
```

A.2.5 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.

A.2.6 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")
```

A.3 PyPy: a fast alternative Python interpreter

PyPy is a fast, compliant alternative implementation of the Python language (2.7.8 and 3.2.5). Thanks to its Just-in-Time compiler, Python programs often run faster on PyPy than on CPython. PyPy's Python Interpreter is written in RPython and implements the full Python language. This interpreter very closely emulates the behavior of CPython. It contains the following key components: a bytecode compiler responsible for producing Python code objects from the source code of a user application; a bytecode evaluator responsible for interpreting Python code objects; a standard object space, responsible for creating and manipulating the Python objects seen by the application.

The bytecode compiler is the preprocessing phase that produces a compact bytecode format via a chain of flexible passes (tokenizer, lexer, parser, abstract syntax tree builder, bytecode generator). The bytecode evaluator interprets this bytecode. It does most of its work by delegating all actual manipulations of user objects to the object space. The latter can be thought of as the library of built-in types. It defines the implementation of the user objects, like integers and lists, as well as the operations between them, like addition or truth-value-testing.

This division between bytecode evaluator and object space gives a lot of flexibility. One can plug in different object spaces to get different or enriched behaviours of the Python objects.

A.3.1 Installation

A.3.1.1 Windows

A.3.1.2 Mac OSX

A.3.1.3 GNU/Linux

A.4 Jython: Python for the Java Virtual Machine

This list of JVM Languages comprises notable computer programming languages that are used to produce software that runs on the Java Virtual Machine (JVM). Some of these languages are interpreted by a Java program, and some are compiled to Java bytecode and JITcompiled during execution as regular Java programs to improve performance. The JVM was initially designed to support only the Java programming language. However, as time passed, ever more languages were adapted or designed to run on the Java platform.

Apart from the Java language itself, the most common or well-known JVM languages are:

- Clojure, a Lisp dialect
- Groovy, a programming and scripting language
- Scala, an object-oriented and functional programming language[1]
- JRuby, an implementation of Ruby
- Jython, an implementation of Python

Jython is an implementation of the Python language for the Java platform. Throughout this book, you will be learning how to use the Python language, and along the way we will show you where the Jython implementation differs from CPython, which is the canonical implementation of Python written in the C language. It is important to note that the Python language syntax remains consistent throughout the different implementations. At the time of this writing, there are three mainstream implementations of Python. These implementations are: CPython, Jython for the Java platform, and IronPython for the .NET platform. At the time of this writing, CPython is the most prevalent of the implementations. Therefore if you see the word Python somewhere, it could well be referring to that implementation.

This book will reference the Python language in sections regarding the language syntax or functionality that is inherent to the language itself. However, the book will reference the name Jython when discussing functionality and techniques that are specific to the Java platform implementation. No doubt about it, this book will go in-depth to cover the key features of Jython and you'll learn concepts that only adhere to the Jython implementation. Along the way, you will learn how to program in Python and advanced techniques.

Developers from all languages and backgrounds will benefit from this book. Whether you are interested in learning Python for the first time or discovering Jython techniques and advanced concepts, this book is a good fit. Java developers and those who are new to the Python language will find specific interest in reading through Part I of this book as it will teach the Python language from the basics to more advanced concepts. Seasoned Python developers will probably find more interest in Part II and Part III as they focus more on the Jython implementation specifics. Often in this reference, you will see Java code compared with Python code.

A.4.1 Installing Java

A.4.1.1 Windows

A.4.1.2 Mac OSX

A.4.1.3 GNU/Linux

A.4.2 Installing Eclipse with support for JPython

A.4.2.1 Windows

A.4.2.2 Mac OSX

A.4.2.3 GNU/Linux

A.5 IronPython: Python for .NET Framework and Mono

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 mpFormulaPy 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();
```

A.6 Pythonista: Python for iOS

Pythonista is an integrated development environment for writing Python scripts on iOS. You can create interactive experiments and prototypes using multi-touch, animations, and sound â€” or just use the interactive prompt as a powerful calculator.

With extensive support for URL schemes, access to the iOS clipboard and photo library, and all the powerful libraries that come with Python, it is also possible to use Pythonista as a flexible automation tool for text or image processing.

Pythonista is also an excellent companion for learning the Python language â€” You can get started easily with lots of ready-to-run examples, the interactive prompt helps you experiment, and you can read the entire documentation right within the app.

The code editor has everything you'd expect: Syntax highlighting, code completion and an extended keyboard specifically designed for writing Python. You can even extend the editor with your own scripts.

The Standard Library has tons of modules for doing math, processing text, working with data from the web, and much more. Additionally, you have access to the iOS clipboard, an in-app web browser and the powerful Python Imaging Library â€” including support for accessing photos in your camera roll.

Pythonista includes the NumPy and matplotlib packages for scientific computation and data visualization.

With the integrated UI Editor, you can create user interfaces for your scripts without writing any code.

A.7 QPython: Python for Android

QPython is a script engine that runs Python on android devices. It lets your android device run Python scripts and projects. It contains the Python interpreter, console, editor, and the SL4A Library for Android. It's Python on Android! It offers the development kit which lets you easily develop Python projects and scripts on your Android device.

[[Main Features]]

- * Supports Python programming on Android including web apps, games, and SL4A programming etc
- * Run Python scripts / projects on Android devices
- * Can execute Python code & files from QRCode
- * QEdit lets you create/edit Python scripts / projects easily
- * Includes many useful python libraries
- * Support pip

[[Programming Features]]

- * Supports Web App programming, which let you develop mobile apps with web development framework, this speeds up your mobile development greatly
- * Supports native UI programming, which let you develop games more easily by using scripts
- * Supports SL4A programming to access Android's features: network, Bluetooth, GPS, and more

A.8 Brython: a Python to JavaScript compiler

Brython is a Python to JavaScript compiler, but it does the whole job in the browser and so makes Python appear to be a client scripting language.

Brythonic is an ancient Celtic language, but in this case Brython stands for Browser Python, or I suppose it could be reference to the Life of Brian. Wherever the name derives, Brython is another of the growing examples of treating JavaScript as an assembly language. In this case, though, the approach is slightly different. The idea is that you can write Python in the browser as if it was a language with built-in browser support.

For example:

```
<script type="text/python">
def echo():
    alert("Hello %s !" %doc["zone"].value) </script>
```

You can see that this looks like a native script. You can also see that it is Python with a few extensions to enable it to work in the browser environment - doc represents the DOM and you can use it to select an element using indexing doc["zone"] is the element with id "zone".

Appendix B

LibreOffice Calc and Microsoft Excel

B.1 LibreOffice Calc

The mpFormulaPy library provides multiprecision arithmetic versions of functions which are equivalent to the numerically orientated functions of LibreOffice Calc.

The following references are useful for automating LibreOffice:

www.pitonyak.org/oo.php.

http://www.pitonyak.org/OOME_3.0.pdf.

http://www.wollmux.net/wiki/images/f/f9/Makro_Kochbuch.pdf.

https://wiki.documentfoundation.org/images/6/63/Makroprogrammierung_V41.pdf.

<https://code.google.com/p/pyscripter/>.

B.1.1 Windows

Installation on Windows, LibreOffice Portable:

The mpmath directory needs to be copied into the LibreOffice Python site-packages directory:

`\LibreOfficePortable\App\libreoffice\program\python-core-3.3.3\lib\site-packages`

If you want gmpy2 support, copy the files

`gmpy2.pyd`
`gmpy2-2.0.5-py3.3.egg-info`

into the same directory.

The `sheetFunction.py` file needs to be copied into the LibreOffice Python Scripts directory:

`\LibreOfficePortable\App\libreoffice\share\Scripts\python`

Example for calling a worksheet function from within StarBasic.

```
Function foo(x As Double) As Double
    Dim svc As Object
    svc = createUnoService("com.sun.star.sheet.FunctionAccess")
    foo = svc.callFunction("ERFC", Array(x))
End Function
```

Example for calling a worksheet function from within VB.NET using the Windows-specific COM Interface

Module Program

```

Sub CallLO()
    Dim objServiceManager As Object = CreateObject("com.sun.star.ServiceManager")
    Dim objDesktop As Object =
        objServiceManager.CreateInstance("com.sun.star.frame.Desktop")

    Dim objSvc As Object =
        objServiceManager.CreateInstance("com.sun.star.sheet.FunctionAccess")
    Dim x As Double = 3.0
    Dim arr1 As Double() = {x}
    Dim Result As Double = objSvc.callFunction("ERFC", arr1)

    Dim arrEmpty As Object() = {}
    Dim objCalcDoc As Object = objDesktop.loadComponentFromURL("private:factory/scalc",
        "_blank", 0, arrEmpty)
    'Set objCalcDoc = objDesktop.loadComponentFromURL("file:///C:/file.ods", "_blank",
    '0, Array())

    Dim objSheet As Object = objCalcDoc.getSheets().getByIndex(0)
    objSheet.getCellByPosition(0, 0).SetString("Result: " + Result.ToString())
    objSheet.getCellByPosition(0, 0).charWeight = 150
End Sub

Sub Main()
    CallLO()
End Sub

End Module

```

Additional examples for calling LibreOffice via COM can be found here:
<http://www.columbia.edu/~em36/wp-ood.vbs.html>.

B.1.2 Mac OSX

Installation on the Macintosh:

In Finder, navigate to:

Macintosh HD/Applications/LibreOffice

Right-Click on LibreOffice, and choose 'Show Package Contents'. This will open the LibreOffice 'Contents' subdirectory. The mpmath directory needs to be copied into the Python site-packages directory:

Contents/Frameworks/LibreOfficePython.framework/Versions/3.3/lib/python3.3/site-packages

The `sheetFunction.py` file needs to be copied into the LibreOffice Python Scripts directory:

Contents/Resources/Scripts/Python

B.1.3 GNU/Linux

Installation on Ubuntu 14.04

To install support for python scripting from within LibreOffice:

```
sudo apt-get install libreoffice-script-provider-python
```

To install mpmath for Python 3.x:

```
sudo apt-get install python3/mpmath
```

if nautilus is not already working, do the following:

```
sudo mkdir -p /root/.config/nautilus
gksu nautilus
```

if gksu was not already installed, you need to install it:

```
sudo apt-get install gksu
```

The `sheetFunction.py` file needs to be copied into the LibreOffice Python Scripts directory:

Computer/usr/lib/libreoffice/share/Scripts/python

B.1.4 Controlling LibreOffice using sockets or named pipes

It is possible to control LibreOffice from CPython using sockets or named pipes. This will work on Windows, Mac OSX and Linux, and requires access to `uno.py`, which is typically pre-installed with LibreOffice. The use of `uno` is much simplified by an `uno` wrapper like `pyoo`, which is available from here:

<https://pypi.python.org/pypi/pyoo/1.0>.

This site also contains detailed installation instructions.

When calling `uno` from a CPython installation other than the one provided by LibreOffice, it is important to make sure that the first two digits of the Python versions are the same, e.g. on Windows, the most current LibreOffice version includes CPython 3.3.5. This is compatible with CPython 3.3.4, but NOT with CPython 3.4.3.

It is also necessary to set a number of environment variables.

The following code assumes that `pyoo` is available in the CPython path, and that you have already started LibreOffice with sockets:

```
import os
import sys

## This is the hard-coded path of your LibreOffice installation
## Adapt as appropriate
LOPath = 'I:\\\\Extra\\\\PortableApps\\\\LibreOfficePortable\\\\App\\\\libreoffice\\\\'

os.environ['PATH'] += os.pathsep + LOPath + 'URE\\\\bin'
os.environ['PATH'] += os.pathsep + LOPath + 'program'
os.environ['URE_BOOTSTRAP'] = 'vnd.sun.star.pathname:' + LOPath +
    'program\\\\fundamental.ini'
os.environ['UNO_PATH'] = LOPath + 'program\\\\'
```

```
sys.path.append(LOPath + 'program')

import pyoo
desktop = pyoo.Desktop('localhost', 2002)
doc = desktop.create_spreadsheet()
doc.sheets[0]
sheet = doc.sheets[0]
str(sheet[0,0].address)
sheet[0,0].value = 1
sheet[0,1].value = 2
sheet[0,2].formula = '$A$1+$B$1'
sheet[0,2].formula = '=A$1+$B$1'
sheet[0,2].value
sheet[1:3,0:2].values = [[3,4],[5,6]]
sheet[3,0:2].formulas = ['=$A$1+$A$2+$A$3', '$B$1+$B$2+$B$3']
sheet[3,0:2].values
##doc.close()
```

This file is available as ..\mpFormulaPy\Win32_Python33\Lib\TestUno.py

Additional background information can be found at the following links:

<http://stackoverflow.com/questions/24965406/getting-python-to-import-uno-pyuno>.

<http://stackoverflow.com/questions/4270962/using-pyuno-with-my-existing-python-installation>.
An alternative uno wrapper can be found here:

<https://pypi.python.org/pypi/oosheet>.

B.2 Microsoft Excel

The mpFormulaPy library provides multiprecision arithmetic versions of functions which are equivalent to the numerically orientated functions of MS Excel.

B.2.1 Windows

The functions are enabled by selecting and activating the mpFormula32.xll or mpFormula64.xll Addin.

B.2.2 Mac OSX

The connection to MS Excel on OSX still needs to be worked out.

B.3 Spreadsheet functions implemented in mpFormulaPy

The mpFormulaPy library provides multiprecision arithmetic versions of functions which are equivalent to the numerically orientated functions of MS Excel and OpenOffice Calc.

B.3.1 Spreadsheet Mathematical Functions

The mathematical functions listed below are supported by MS Excel 2013, Apache Open Office 4.0, and LibreOffice 4.1. Earlier versions of these programs support a subset of these functions. All of these functions have multiprecision arithmetic equivalents in mpFormulaPy.

Table B.1: Spreadsheet Mathematical Functions

Function	Description	Section
ABS	Returns the absolute value of a number	4.6.7 (page 93)
ACOS	Returns the arccosine of a number	5.6.2 (page 186)
ACOSH	Returns the inverse hyperbolic cosine of a number	5.7.2 (page 194)
ACOT	Returns the inverse cotangent of a number	5.6.5 (page 191)
ACOTH	Returns the inverse hyperbolic cotangent of a number	5.7.4 (page 196)
ASIN	Returns the arcsine of a number	5.6.1 (page 184)
ARABIC	Returns a Roman number to Arabic, as a number	4.3.1 (page 77)
ASINH	Returns the inverse hyperbolic sine of a number	5.7.1 (page 193)
ATAN	Returns the arctangent of a number	5.6.3 (page 188)
ATAN2	Returns the arctangent from x- and y-coordinates	5.6.4 (page 190)
ATANH	Returns the inverse hyperbolic tangent of a number	5.7.3 (page 195)
BASE	Converts a number into a text representation with the given radix (base)	4.3.6 (page 80)
CEILING	Rounds a number to the nearest integer or to the nearest multiple of significance	4.5.2 (page 83)
CEILING.MATH	Rounds a number to the nearest integer or to the nearest multiple of significance	4.5.2 (page 84)
COMBIN	Returns the number of combinations for a given number of objects	5.9.3 (page 202)
COMBINA	Returns the number of combinations with repetitions for a given number of objects	5.9.3 (page 202)
COS	Returns the cosine of a number	5.4.5 (page 160)
COSH	Returns the hyperbolic cosine of a number	5.5.2 (page 174)
COT	Returns the cotangent of a number	5.4.9 (page 168)
COTH	Returns the hyperbolic cotangent of a number	5.5.6 (page 182)
CSC	Returns the cosecant of a number	5.4.8 (page 166)
CSCH	Returns the hyperbolic cosecant of a number	5.5.5 (page 180)
DECIMAL	Converts a text representation of a number in a given base into a decimal number	4.3.6 (page 81)
DEGREES	Converts radians to degrees	5.4.2 (page 156)
EVEN	Rounds a number up to the nearest even integer	4.5.5 (page 86)
EXP	Returns e raised to the power of a given number	5.2.1 (page 136)

Continued on next page

Table B.1 – *Continued from previous page*

Function	Description	Section
FACT	Returns the factorial of a number	5.9.1 (page 202)
FACTDOUBLE	Returns the double factorial of a number	5.9.2 (page 202)
FLOOR	Rounds a number down, toward zero	4.5.3 (page 84)
FLOOR.MATH	Rounds a number down, to the nearest integer or to the nearest multiple of significance	4.5.3 (page 85)
GCD	Returns the greatest common divisor	5.9.6 (page 203)
GCD_ADD	*Returns the greatest common divisor	5.9.6 (page 203)
INT	Rounds a number down to the nearest integer	4.5.7 (page 86)
LCM	Returns the least common multiple	5.9.7 (page 204)
LCM_ADD	*Returns the least common multiple	5.9.7 (page 204)
LN	Returns the natural logarithm of a number	5.2.4 (page 141)
LOG	Returns the logarithm of a number to a specified base	5.2.4.1 (page 142)
LOG10	Returns the base-10 logarithm of a number	5.2.5 (page 143)
MDETERM	Returns the matrix determinant of an array	6.1.1 (page 205)
MINVERSE	Returns the matrix inverse of an array	6.1.2 (page 205)
MMULT	Returns the matrix product of two arrays	4.12.7 (page 128)
MUNIT	Returns the unit matrix	4.12.1 (page 123)
MOD	Returns the remainder from division	4.7.10 (page 109)
MROUND	Returns a number rounded to the desired multiple	4.5.11 (page 88)
MULTINOMIAL	Returns the multinomial of a set of numbers	5.9.4 (page 203)
ODD	Rounds a number up to the nearest odd integer	4.5.6 (page 86)
PI	Returns the value of pi	5.1.1 (page 133)
POWER	Returns the result of a number raised to a power	5.3.2 (page 146)
PRODUCT	Multiplies its arguments	4.7.6 (page 104)
QUOTIENT	Returns the integer portion of a division	4.5.12 (page 89)
RADIANS	Converts degrees to radians	5.4.2 (page 156)
RAND	Returns a random number between 0 and 1	4.11.1 (page 120)
RANDBETWEEN	Returns a random number between the numbers you specify	4.11.1 (page 120)
ROMAN	Converts an arabic numeral to roman, as text	4.3.1 (page 77)
ROUND	Rounds a number to a specified number of digits	4.5.1 (page 83)
ROUNDDOWN	Rounds a number down, toward zero	4.5.9 (page 88)
ROUNDUP	Rounds a number up, away from zero	4.5.10 (page 88)
SERIESSUM	Returns the sum of a power series based on the formula	4.7.2 (page 100)
SIGN	Returns the sign of a number	4.6.9 (page 95)
SEC	Returns the secant of the given angle	5.4.7 (page 164)
SECH	Returns the hyperbolic secant of the given angle	5.5.4 (page 178)
SIN	Returns the sine of the given angle	5.4.4 (page 158)
SINH	Returns the hyperbolic sine of a number	5.5.1 (page 172)
SQRT	Returns a positive square root	5.3.3 (page 149)
SQRTPI	Returns the square root of (number * pi)	5.4.3 (page 157)
SUBTOTAL	Returns a subtotal in a list or database	8.5.1 (page 298)
SUM	Adds its arguments	8.3.1 (page 284)

Continued on next page

Table B.1 – *Continued from previous page*

Function	Description	Section
SUMIF	Adds the cells specified by a given criteria	8.3.1 (page 284)
SUMIFS	!!Adds the cells in a range that meet multiple criteria	8.3.1 (page 284)
SUMPRODUCT	Returns the sum of the products of corresponding array components	4.7.2 (page 100)
SUMSQ	Returns the sum of the squares of the arguments	4.7.2 (page 100)
SUMX2MY2	Returns the sum of the difference of squares of corresponding values in two arrays	4.7.2 (page 99)
SUMX2PY2	Returns the sum of the sum of squares of corresponding values in two arrays	4.7.2 (page 99)
SUMXMY2	Returns the sum of squares of differences of corresponding values in two arrays	4.7.2 (page 99)
TAN	Returns the tangent of a number	5.4.6 (page 162)
TANH	Returns the hyperbolic tangent of a number	5.5.3 (page 176)
TRUNC	Truncates a number to an integer	4.5.4 (page 86)

B.3.2 Spreadsheet Engineering Functions

The Engineering functions listed below are supported by MS Excel 2013, Apache Open Office 4.0, and LibreOffice 4.1. Earlier versions of these programs support a subset of these functions. All of these functions have multiprecision arithmetic equivalents in mpFormulaPy. The implementation is based on Apache Open Office. Functions marked with an !! are only available in MS Excel.

Table B.2: Spreadsheet Engineering Functions

Function	Description	Section
BESSELI	Returns the modified Bessel function $I_n(x)$	5.8.3 (page 198)
BESSELJ	Returns the Bessel function $J_n(x)$	5.8.1 (page 198)
BESSELK	Returns the modified Bessel function $K_n(x)$	5.8.4 (page 199)
BESSELY	Returns the Bessel function $Y_n(x)$	5.8.2 (page 198)
BIN2DEC	Converts a binary number to decimal	4.3.2 (page 77)
BIN2HEX	Converts a binary number to hexadecimal	4.3.2 (page 77)
BIN2OCT	Converts a binary number to octal	4.3.2 (page 77)
BITAND	Returns a value shifted left by shift_amounts bits	4.8.1 (page 110)
BITLSHIFT	Returns a value shifted left by shift_amounts bits	4.7.7 (page 106)
BITOR	Returns a value shifted left by shift_amounts bits	4.8.2 (page 110)
BITRSHIFT	Returns a value shifted right by shift_amounts bits	4.7.9 (page 108)
BITXOR	Returns a value shifted left by shift_amounts bits	4.8.3 (page 110)
COMPLEX	Converts real and imaginary coefficients into a complex number	4.6.3 (page 91)
CONVERT	Converts a number from one measurement system to another	8.2.1 (page 280)
DEC2BIN	Converts a decimal number to binary	4.3.3 (page 78)
DEC2HEX	Converts a decimal number to hexadecimal	4.3.3 (page 78)
DEC2OCT	Converts a decimal number to octal	4.3.3 (page 78)
DELTA	Tests whether two values are equal	4.9.7 (page 113)
ERF	Returns the error function	5.8.4 (page 199)
ERFC	Returns the complementary error function	5.8.5 (page 199)
GAMMA	Returns the Gamma function value	5.8.6 (page 200)
GESTEP	!!Tests whether a number is greater than a threshold value	4.9.7 (page 113)
HEX2BIN	Converts a hexadecimal number to binary	4.3.4 (page 79)
HEX2DEC	Converts a hexadecimal number to decimal	4.3.4 (page 79)
HEX2OCT	Converts a hexadecimal number to octal	4.3.4 (page 79)
IMABS	Returns the absolute value (modulus) of a complex number	4.6.7 (page 93)
IMAGINARY	Returns the imaginary coefficient of a complex number	4.6.6 (page 92)
IMARGUMENT	Returns the argument theta, an angle expressed in radians	4.6.8 (page 94)
IMCONJUGATE	Returns the complex conjugate of a complex number	4.6.10 (page 95)
IMCOS	Returns the cosine of a complex number	5.4.5 (page 160)
IMCOSH	Returns the hyperbolic cosine of a complex number	5.5.2 (page 174)
IMCOT	Returns the cotangent of a complex number	5.4.9 (page 168)
IMCSC	Returns the hyperbolic cosine of a complex number	5.4.8 (page 166)

Continued on next page

Table B.2 – *Continued from previous page*

Function	Description	Section
IMCSCH	Returns the cotangent of a complex number	5.5.5 (page 180)
IMDIV	Returns the quotient of two complex numbers	4.7.8 (page 107)
IMEXP	Returns the exponential of a complex number	5.2.1 (page 136)
IMLN	Returns the natural logarithm of a complex number	5.2.4 (page 141)
IMLOG10	Returns the base-10 logarithm of a complex number	5.2.5 (page 143)
IMLOG2	Returns the base-2 logarithm of a complex number	5.2.6 (page 144)
IMPOWER	Returns a complex number raised to an integer power	5.3.2 (page 146)
IMPRODUCT	Returns the product of complex numbers	4.7.6 (page 105)
IMREAL	Returns the real coefficient of a complex number	4.6.5 (page 92)
IMSEC	Returns the secant of a complex number	5.4.7 (page 164)
IMSECH	Returns the hyperbolic secant of a complex number	5.5.4 (page 178)
IMSIN	Returns the sine of a complex number	5.4.4 (page 158)
IMSINH	Returns the hyperbolic sine of a complex number	5.5.1 (page 172)
IMSQRT	Returns the square root of a complex number	5.3.3 (page 149)
IMSUB	Returns the difference between two complex numbers	4.7.3 (page 101)
IMSUM	Returns the sum of complex numbers	4.7.2 (page 98)
IMTAN	Returns the tangent of a complex number	5.4.6 (page 162)
OCT2BIN	Converts an octal number to binary	4.3.5 (page 80)
OCT2DEC	Converts an octal number to decimal	4.3.5 (page 80)
OCT2HEX	Converts an octal number to hexadecimal	4.3.5 (page 80)

B.3.3 Spreadsheet Statistical Functions

The Statistical functions listed below are supported by MS Excel 2013, Apache Open Office 4.0, and LibreOffice 4.1. Earlier versions of these programs support a subset of these functions. Functions marked with an !! are only available in MS Excel. Functions marked with an asterix are only available in Open Office. All of these functions have multiprecision arithmetic equivalents in mpFormulaPy.

Table B.3: Spreadsheet Statistical Functions

Function	Description	Section
AVEDEV	Returns the average of the absolute deviations of data points from their mean	8.3.7 (page 289)
AVERAGE	Returns the average of its arguments	8.3.2 (page 285)
AVERAGEA	Returns the average of its arguments, including numbers, text, and logical values	8.3.2 (page 285)
AVERAGEIF	!!Returns the average (arithmetic mean) of all the cells in a range that meet a given criteria	8.3.2 (page 285)
AVERAGEIFS	!!Returns the average (arithmetic mean) of all cells that meet multiple criteria.	8.3.2 (page 285)
BETADIST	Returns the beta cumulative distribution function	7.2.2 (page 223)
BETAINV	Returns the inverse of the cumulative distribution function for a specified beta distribution	7.2.3 (page 224)
BINOMDIST	Returns the individual term binomial distribution probability	7.3.1 (page 228)
CHIDIST	Returns the one-tailed probability of the chi-squared distribution	7.4.2 (page 232)
CHIINV	Returns the inverse of the one-tailed probability of the chi-squared distribution	7.4.3 (page 234)
CHITEST	Returns the test for independence	8.6.8 (page 303)
CONFIDENCE	Returns the confidence interval for a population mean	8.6.1 (page 300)
CORREL	Returns the correlation coefficient between two data sets	8.7.2 (page 305)
COUNT	Counts how many numbers are in the list of arguments	8.1.1 (page 277)
COUNTA	Counts how many values are in the list of arguments	8.1.1 (page 277)
COUNTBLANK	Counts the number of blank cells within a range	8.1.1 (page 277)
COUNTIF	Counts the number of cells within a range that meet the given criteria	8.1.1 (page 278)
COUNTIFS	!!Counts the number of cells within a range that meet multiple criteria	8.1.1 (page 278)
COVAR	Returns covariance, the average of the products of paired deviations	8.7.1 (page 305)
CRITBINOM	Returns the smallest value for which the cumulative binomial distribution is less than or equal to a criterion value	7.3.2 (page 229)

Continued on next page

Table B.3 – *Continued from previous page*

Function	Description	Section
DEVSQ	Returns the sum of squares of deviations	8.3.8 (page 289)
EXPONDIST	Returns the exponential distribution	7.5.1 (page 238)
FDIST	Returns the F probability distribution	7.6.2 (page 241)
FINV	Returns the inverse of the F probability distribution	7.6.3 (page 243)
FISHER	Returns the Fisher transformation	8.7.3 (page 306)
FISHERINV	Returns the inverse of the Fisher transformation	8.7.3 (page 307)
FORECAST	Returns a value along a linear trend	8.8.3 (page 308)
FREQUENCY	Returns a frequency distribution as a vertical array	8.1.2 (page 278)
FTEST	Returns the result of an F-test	8.6.5 (page 302)
GAMMADIST	Returns the gamma distribution	7.7.1 (page 247)
GAMMAINV	Returns the inverse of the gamma cumulative distribution	7.7.2 (page 248)
GAMMALN	Returns the natural logarithm of the gamma	5.8.6 (page 200)
GEOMEAN	Returns the geometric mean	8.3.3 (page 286)
GROWTH	Returns values along an exponential trend	6.2.2 (page 207)
HARMEAN	Returns the harmonic mean	8.3.4 (page 286)
HYPGEOMDIST	Returns the hypergeometric distribution	7.8.2 (page 251)
INTERCEPT	Returns the intercept of the linear regression line	8.8.1 (page 308)
KURT	Returns the kurtosis of a data set	8.3.10 (page 291)
LARGE	Returns the k-th largest value in a data set	8.4.5 (page 293)
LINEST	Returns the parameters of a linear trend	6.1.3 (page 206)
LOGEST	Returns the parameters of an exponential trend	6.2.1 (page 207)
LOGINV	Returns the inverse of the lognormal distribution	7.9.3 (page 256)
LOGNORMDIST	Returns the cumulative lognormal distribution	7.9.2 (page 254)
MAX	Returns the maximum value in a list of arguments	8.4.2 (page 292)
MAXA	Returns the maximum value in a list of arguments, including numbers, text, and logical values	8.4.2 (page 292)
MEDIAN	Returns the median of the given numbers	8.4.3 (page 292)
MIN	Returns the minimum value in a list of arguments	8.4.1 (page 292)
MINA	Returns the smallest value in a list of arguments, including numbers, text, and logical values	8.4.1 (page 292)
MODE	Returns the most common value in a data set	8.4.4 (page 293)
NEGBINOMDIST	Returns the negative binomial distribution	7.10.1 (page 258)
NORMDIST	Returns the normal cumulative distribution	7.11.2 (page 261)
NORMINV	Returns the inverse of the normal cumulative distribution	7.11.3 (page 263)
NORMSDIST	Returns the standard normal cumulative distribution	7.11.2 (page 262)
NORMSINV	Returns the inverse of the standard normal cumulative distribution	7.11.3 (page 264)
PEARSON	Returns the Pearson product moment correlation coefficient	8.7.2 (page 306)
PERCENTILE	Returns the k-th percentile of values in a range	8.4.7 (page 294)
PERCENTRANK	Returns the percentage rank of a value in a	8.4.8 (page 295)

Continued on next page

Table B.3 – *Continued from previous page*

Function	Description	Section
PERMUT	Returns the number of permutations for a given number of objects	5.9.5 (page 203)
POISSON	Returns the Poisson distribution	7.12.2 (page 266)
PROB	Returns the probability that values in a range are between two limits	8.4.11 (page 297)
QUARTILE	Returns the quartile of a data set	8.4.9 (page 295)
RANK	Returns the rank of a number in a list of numbers	8.4.10 (page 296)
RSQ	Returns the square of the Pearson product moment correlation coefficient	8.7.2 (page 306)
SKEW	Returns the skewness of a distribution	8.3.9 (page 290)
SLOPE	Returns the slope of the linear regression line	8.8.2 (page 308)
SMALL	Returns the k-th smallest value in a data set	8.4.6 (page 294)
STANDARDIZE	Returns a normalized value	8.2.2 (page 282)
STDEV	Estimates standard deviation based on a sample	8.3.6 (page 288)
STDEVA	Estimates standard deviation based on a sample, including numbers, text, and logical values	8.3.6 (page 288)
STDEVP	Calculates standard deviation based on the entire population	8.3.6 (page 288)
STDEVPA	Calculates standard deviation based on the entire population, including numbers, text, and logical values	8.3.6 (page 289)
STEYX	Returns the standard error of the predicted y-value for each x in the regression	8.8.4 (page 309)
TDIST	Returns the Student's t-distribution	7.13.2 (page 269)
TINV	Returns the inverse of the Student's t-distribution	7.13.3 (page 271)
TREND	Returns values along a linear trend	6.1.4 (page 206)
TRIMMEAN	Returns the mean of the interior of a data set	8.2.3 (page 282)
TTEST	Returns the probability associated with a Student's t-test	8.6.4 (page 301)
VAR	Estimates variance based on a sample	8.3.5 (page 286)
VARA	Estimates variance based on a sample, including numbers, text, and logical values	8.3.5 (page 287)
VARP	Calculates variance based on the entire population	8.3.5 (page 287)
VARPA	Calculates variance based on the entire population, including numbers, text, and logical values	8.3.5 (page 287)
WEIBULL	Returns the Weibull distribution	7.14.1 (page 274)
ZTEST	Returns the one-tailed probability-value of a z-test	8.6.3 (page 301)
PERMUTATIONA	Returns the number of permutations for a given number of objects (with repetitions) that can be selected from the total objects	5.9.5 (page 203)
PHI	Returns the value of the density function for a standard normal distribution	7.11.2 (page 262)
GAUSS	Returns 0.5 less than the standard normal cumulative distribution	7.11.2 (page 262)

Continued on next page

Table B.3 – *Continued from previous page*

Function	Description	Section
CHISQDIST	*Calc	7.4.2 (page 232)
CHISQINV	*Calc	7.4.3 (page 235)

B.3.4 Spreadsheet Revised Statistical Functions

The Statistical functions listed below are supported by MS Excel 2010, 2013, but not by earlier versions. They are not supported by Apache Open Office or LibreOffice. All of these functions have multiprecision arithmetic equivalents in mpFormulaPy.

Table B.4: Spreadsheet Revised Statistical Functions

Function	Description	Section
AGGREGATE	Returns an aggregate in a list or database	8.5.2 (page 298)
BETA.DIST	Returns the cumulative beta probability density function.	7.2.2 (page 223)
BETA.INV	Returns the inverse of the cumulative beta probability density function.	7.2.3 (page 225)
BINOM.DIST	Returns the individual term binomial distribution probability.	7.3.1 (page 228)
BINOM.DIST.RANGE	Returns the probability of a trial result using a binomial distribution.	7.3.1 (page 229)
BINOM.INV	Returns the smallest number which is the cumulative binomial distribution that is greater than a criterion value.	7.3.2 (page 230)
CEILING.PRECISE	Rounds a number the nearest integer or to the nearest multiple of significance. Regardless of the sign of the number, the number is rounded up.	4.5.2 (page 83)
CHISQ.DIST	Returns the cumulative beta probability density function	7.4.2 (page 232)
CHISQ.DIST.RT	Returns the one tailed probability of the chi-squared distribution.	7.4.2 (page 233)
CHISQ.INV	Returns the inverse of the chi-squared distribution.	7.4.3 (page 234)
CHISQ.INV.RT	Returns the inverse of the one tailed probability of the chi-squared distribution.	7.4.3 (page 235)
CHISQ.TEST	Returns the value from the chi-squared test	8.6.8 (page 303)
CONFIDENCE.NORM	Returns the confidence interval for a population mean.	8.6.1 (page 300)
CONFIDENCE.T	Returns the confidence interval for a population using a Student's t distribution	8.6.2 (page 300)
COVARIANCE.P	Returns the covariance of two lists of numbers.	8.7.1 (page 305)
COVARIANCE.S	Returns the sample covariance, the average of the products deviations for each data point pair in two data sets	8.7.1 (page 305)
ERF.PRECISE	Returns the error function	5.8.4 (page 199)
ERFC.PRECISE	Returns the complementary ERF function integrated between x and infinity	5.8.5 (page 199)
EXPON.DIST	Returns the exponential distribution.	7.5.1 (page 238)
F.DIST	Returns the F probability distribution	7.6.2 (page 241)

Continued on next page

Table B.4 – *Continued from previous page*

Function	Description	Section
F.DIST.RT	Returns the F probability distribution.	7.6.2 (page 241)
F.INV	Returns the inverse of the F probability distribution.	7.6.3 (page 243)
F.INV.RT	Returns the inverse of the F probability distribution.	7.6.3 (page 243)
F.TEST	Returns the result of an F-test.	8.6.5 (page 302)
FLOOR.PRECISE	Rounds a number the nearest integer or to the nearest multiple of significance. Regardless of the sign of the number, the number is rounded up.	4.5.3 (page 84)
GAMMA.DIST	Returns the gamma distribution.	7.7.1 (page 247)
GAMMA.INV	Returns the inverse of the gamma distribution.	7.7.2 (page 248)
GAMMALN.PRECISE	Returns the natural logarithm of the gamma function, $\Gamma(x)$	5.8.6 (page 200)
HYPGEOM.DIST	Returns the hyper geometric distribution for a finite population.	7.8.2 (page 252)
LOGNORM.DIST	Returns the cumulative lognormal distribution of x, where $\ln(x)$ is normally distributed with parameters mean and stdev.	7.9.2 (page 255)
LOGNORM.INV	Returns the inverse of the lognormal cumulative distribution function	7.9.3 (page 256)
MODE.MULT	Returns a vertical array of the most frequently occurring, or repetitive values in an array or range of data	8.4.4 (page 293)
MODE.SNGL	Returns the number that occurs most frequently in a range.	8.4.4 (page 293)
NEGBINOM.DIST	Returns the negative binomial distribution.	7.10.1 (page 258)
NETWORKDAYS.INTL	Returns the number of whole workdays between two dates using parameters to indicate which and how many days are weekend days	28.3.2 (page 577)
NORM.DIST	Returns the normal cumulative distribution.	7.11.2 (page 261)
NORM.INV	Returns the inverse of the normal cumulative distribution.	7.11.3 (page 264)
NORM.S.DIST	Returns the standard normal cumulative distribution.	7.11.2 (page 262)
NORM.S.INV	Returns the inverse of the standard normal cumulative distribution.	7.11.3 (page 264)
PERCENTILE.EXC	Returns the Kth percentile of values in a range, where k is in the range 0..1, exclusive	8.4.7 (page 294)
PERCENTILE.INC	The Kth percentile of values in an array of numbers.	8.4.7 (page 294)
PERCENTRANK.EXC	Returns the rank of a value in a data set as a	8.4.8 (page 295)

Continued on next page

Table B.4 – *Continued from previous page*

Function	Description	Section
PERCENTRANK.INC	percentage (0..1, exclusive) of the data set	
	Returns the rank of a value in a data set as a percentage of the data set.	8.4.8 (page 295)
POISSON.DIST	Returns the Poisson distribution.	7.12.2 (page 266)
QUARTILE.EXC	Returns the quartile of the data set, based on percentile values from 0..1, exclusive	8.4.9 (page 296)
QUARTILE.INC	Returns the quartile of a data set.	8.4.9 (page 296)
RANK.AVG	Returns the rank of a number in a list of numbers.	8.4.10 (page 297)
RANK.EQ	Returns the rank of a number in a list of numbers.	8.4.10 (page 297)
SKEW.P	Returns the skewness of a distribution based on the entire population.	8.3.9 (page 290)
STDEV.P	Returns the standard deviation based on the entire population.	8.3.6 (page 289)
STDEV.S	Returns the standard deviation based on a sample.	8.3.6 (page 288)
T.DIST	Returns the Percentage Points (probability) for the Student t-distribution	7.13.2 (page 269)
T.DIST.2T	Returns the two-tailed Student's t-distribution	7.13.2 (page 270)
T.DIST.RT	Returns the right-tailed Student's t-distribution	7.13.2 (page 269)
T.INV	Returns the T-value of the Student's t-distribution	7.13.3 (page 271)
T.INV.2T	Returns the t-value of the Student's t-distribution	7.13.3 (page 271)
T.TEST	Returns the probability associated with the F-Test.	8.6.4 (page 302)
VAR.P	Calculates variance based on the population.	8.3.5 (page 287)
VAR.S	Returns the variance based on a sample.	8.3.5 (page 286)
WEIBULL.DIST	Returns the Weibull distribution.	7.14.1 (page 274)
WORKDAY.INTL	Returns the serial number of the date before or after a specified number of workdays	28.2.11 (page 574)
Z.TEST	Returns the two-tailed P-value of a z-test	8.6.3 (page 301)

B.3.5 Spreadsheet Database Functions

The Database functions listed below are supported by MS Excel 2013, Apache Open Office 4.0, and LibreOffice 4.1. Earlier versions of these programs support a subset of these functions. All of these functions have multiprecision arithmetic equivalents in mpFormulaPy. The implementation is based on Apache Open Office.

Table B.5: Spreadsheet Database Functions

Function	Description	Section
DAVERAGE	Returns the average of selected database entries	8.9.5 (page 311)
DCOUNT	Counts the cells that contain numbers in a database	8.9.3 (page 310)
DCOUNTA	Counts nonblank cells in a database	8.9.3 (page 311)
DGET	Extracts from a database a single record that matches the specified criteria	8.9.1 (page 310)
DMAX	Returns the maximum value from selected database entries	8.9.9 (page 313)
DMIN	Returns the minimum value from selected database entries	8.9.8 (page 313)
DPRODUCT	Multiplies the values in a particular field of records that match the criteria in a database	8.9.2 (page 310)
DSTDEV	Estimates the standard deviation based on a sample of selected database entries	8.9.7 (page 312)
DSTDEVP	Calculates the standard deviation based on the entire population of selected database entries	8.9.7 (page 312)
DSUM	Adds the numbers in the field column of records in the database that match the criteria	8.9.4 (page 311)
DVAR	Estimates variance based on a sample from selected database entries	8.9.6 (page 312)
DVARP	Calculates variance based on the entire population	8.9.6 (page 312)

B.3.6 Spreadsheet Financial Functions

The financial functions listed below are supported by MS Excel 2013, Apache Open Office 4.0, and LibreOffice 4.1. Earlier versions of these programs support a subset of these functions. All of these functions have multiprecision arithmetic equivalents in mpFormulaPy. The implementation is based on Apache Open Office. Functions marked with an asterix are only available in Open Office.

Table B.6: Spreadsheet Financial Functions

Function	Description	Section
ACCRINT	Returns the accrued interest for a security that pays periodic interest	28.5.1 (page 583)
ACCRINTM	Returns the accrued interest for a security that pays interest at maturity	28.5.2 (page 584)
AMORDEGRC	Returns the depreciation for each accounting period by using a depreciation coefficient	28.7.7 (page 597)
AMORLINC	Returns the depreciation for each accounting period	28.7.6 (page 596)
COUPDAYBS	Returns the number of days from the beginning of the coupon period to the settlement date	28.4.1 (page 579)
COUPDAYS	Returns the number of days in the coupon period that contains the settlement date	28.4.2 (page 580)
COUPDAYSNC	Returns the number of days from the settlement date to the next coupon date	28.4.3 (page 580)
COUPNCD	Returns the next coupon date after the settlement date	28.4.4 (page 580)
COUPNUM	Returns the number of coupons payable between the settlement date and maturity date	28.4.5 (page 581)
COUPPCD	Returns the previous coupon date before the settlement date	28.4.6 (page 581)
CUMIPMT	Returns the cumulative interest paid between two periods	28.8.9 (page 601)
CUMIPMT_ADD	*Returns the cumulative interest paid between two periods	28.8.9 (page 601)
CUMPRINC	Returns the cumulative principal paid on a loan between two periods	28.8.10 (page 602)
CUMPRINC_ADD	*Returns the cumulative principal paid on a loan between two periods	28.8.10 (page 602)
DB	Returns the depreciation of an asset for a specified period by using the fixed-declining balance method	28.7.4 (page 595)
ddb	Returns the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify	28.7.1 (page 594)
DISC	Returns the discount rate for a security	28.5.3 (page 584)
DOLLARDE	Converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number	28.10.1 (page 608)
DOLLARFR	Converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fraction	28.10.2 (page 608)

Continued on next page

Table B.6 – *Continued from previous page*

Function	Description	Section
DURATION	Returns the annual duration of a security with periodic interest payments	28.4.7 (page 581)
DURATION_ADD	*Returns the annual duration of a security with periodic interest payments	28.4.7 (page 581)
EFFECT	Returns the effective annual interest rate	28.8.11 (page 602)
EFFECT_ADD	*Returns the effective annual interest rate	28.8.11 (page 602)
FV	Returns the future value of an investment	28.8.1 (page 598)
FVSCHEDULE	Returns the future value of an initial principal after applying a series of compound interest rates	28.8.13 (page 603)
INTRATE	Returns the interest rate for a fully invested given period	28.5.4 (page 585)
IPMT	Returns the interest payment for an investment for a security	28.8.7 (page 601)
IRR	Returns the internal rate of return for a series of cash flows	28.9.1 (page 604)
ISPMT	Calculates the interest paid during a specific period of an investment	28.8.14 (page 603)
MDURATION	Returns the Macauley modified duration for a security with an assumed par value of \$100	28.4.8 (page 582)
MIRR	Returns the internal rate of return where positive and negative cash flows are financed at different rates	28.9.3 (page 605)
NOMINAL	Returns the annual nominal interest rate	28.8.12 (page 602)
NOMINAL_ADD	*Returns the annual nominal interest rate	28.8.12 (page 602)
NPER	Returns the number of periods for an investment	28.8.4 (page 599)
NPV	Returns the net present value of an investment based on a series of periodic cash flows and a discount rate	28.9.4 (page 605)
ODDFPRICE	Returns the price per \$100 face value of a security with an odd first period	28.5.5 (page 585)
ODDFYIELD	Returns the yield of a security with an odd first period cash flows	28.5.6 (page 586)
ODDLPRICE	Returns the price per \$100 face value of a security with an odd last period	28.5.7 (page 587)
ODDLYIELD	Returns the yield of a security with an odd last period	28.5.8 (page 587)
PDURATION	Returns the number of periods required by an investment to reach a specified value	28.8.5 (page 600)
PMT	Returns the periodic payment for an annuity	28.8.3 (page 599)
PPMT	Returns the payment on the principal for an investment for a given period	28.8.8 (page 601)
PRICE	Returns the price per \$100 face value of a security that pays periodic interest	28.5.9 (page 588)
PRICEDISC	Returns the price per \$100 face value of a discounted security	28.5.10 (page 589)

Continued on next page

Table B.6 – *Continued from previous page*

Function	Description	Section
PRICEMAT	Returns the price per \$100 face value of a security that pays interest at maturity	28.5.11 (page 589)
PV	Returns the present value of an investment	28.8.2 (page 598)
RATE	Returns the interest rate per period of an annuity	28.8.6 (page 600)
RECEIVED	Returns the amount received at maturity for a fully invested security	28.5.12 (page 590)
RRI	Returns an equivalent interest rate for the growth of an investment	28.9.2 (page 604)
SLN	Returns the straight-line depreciation of an asset for one period	28.7.2 (page 594)
SYD	Returns the sum-of-years' digits depreciation of an asset for a specified period	28.7.3 (page 595)
TBILLEQ	Returns the bond-equivalent yield for a Treasury bill	28.6.1 (page 592)
TBILLPRICE	Returns the price per \$100 face value for a Treasury bill	28.6.2 (page 592)
TBILLYIELD	Returns the yield for a Treasury bill	28.6.3 (page 592)
VDB	Returns the depreciation of an asset for a specified or partial period by using a declining balance method	28.7.5 (page 596)
XIRR	Returns the internal rate of return for a schedule of cash flows that is not necessarily periodic	28.9.5 (page 606)
XNPV	Returns the net present value for a schedule of cash flows that is not necessarily periodic	28.9.6 (page 606)
YIELD	Returns the yield on a security that pays periodic interest	28.5.13 (page 590)
YIELDDISC	Returns the annual yield for a discounted security; for example, a Treasury bill	28.5.14 (page 591)
YIELDMAT	Returns the annual yield of a security that pays	28.5.15 (page 591)

B.3.7 Spreadsheet Date and Time Functions

The Date and Time functions listed below are supported by MS Excel 2013, Apache Open Office 4.0, and LibreOffice 4.1. All of these functions have equivalents in mpFormulaPy, but there are of course no multiprecision versions.

Table B.7: Spreadsheet Date and Time Functions

Function	Description	Section
DATE	Returns the serial number of a particular date	28.2.1 (page 571)
DATEVALUE	Converts a date in the form of text to a serial number	28.2.3 (page 571)
DAY	Converts a serial number to a day of the month	28.1.4 (page 568)
DAYS	Returns the number of days between two dates	28.1.5 (page 569)
DAYS360	Calculates the number of days between two dates based on a 360-day year	28.3 (page 576)
EDATE	Returns the serial number of the date that is the indicated number of months before or after the start date	28.2.4 (page 572)
EOMONTH	Returns the serial number of the last day of the month before or after a specified number of months	28.2.5 (page 572)
EASTERSUNDAY	*Returns the serial number of Easter Sunday	28.2.2 (page 571)
HOUR	Converts a serial number to an hour	28.1.3 (page 568)
ISOWEEKNUM	Returns the number of the ISO week number of the year for a given date	28.1.9 (page 570)
MINUTE	Converts a serial number to a minute	28.1.2 (page 568)
MONTH	Converts a serial number to a month	28.1.6 (page 569)
NETWORKDAYS	Returns the number of whole workdays between two dates	28.3.1 (page 576)
NOW	Returns the serial number of the current date and time	28.2.6 (page 573)
SECOND	Converts a serial number to a second	28.1.1 (page 567)
TIME	Returns the serial number of a particular time	28.2.7 (page 573)
TIMEVALUE	Converts a time in the form of text to a serial number	28.2.8 (page 573)
TODAY	Returns the serial number of today's date	28.2.9 (page 573)
WEEKDAY	Converts a serial number to a day of the week	28.1.8 (page 569)
WEEKNUM	Converts a serial number to a number representing where the week falls numerically with a year	28.1.9 (page 570)
WEEKNUM_ADD	*Converts a serial number to a number representing where the week falls numerically with a year	28.1.9 (page 570)
WORKDAY	Returns the serial number of the date before or after a specified number of workdays	28.2.10 (page 574)
YEAR	Converts a serial number to a year	28.1.7 (page 569)
YEARFRAC	Returns the year fraction representing the number of whole days between start_date and end_date	28.3.3 (page 578)

Appendix C

Languages with CLR Support

C.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. The most recent version is VB 2012, which was released on August 15, 2012.

C.1.1 Visual Basic 2005

Visual Basic 2005 was the name used to refer to Visual Basic .NET, as Microsoft decided to drop the .NET portion of the title.

For this release, Microsoft added many features, including:

Edit and Continue

Design-time expression evaluation.

The My pseudo-namespace (overview, details), which provides easy access to certain areas of the .NET Framework that otherwise require significant code to access dynamically generated classes (notably My.Forms)

The Using keyword, simplifying the use of objects that require the Dispose pattern to free resources

Just My Code, which when debugging hides (steps over) boilerplate code written by the Visual Studio .NET IDE and system library code

Data Source binding, easing database client/server development

Generics

Partial classes, a method of defining some parts of a class in one file and then adding more definitions later; particularly useful for integrating user code with auto-generated code

Operator overloading and nullable Types

Support for unsigned integer data types commonly used in other languages

C.1.2 Visual Basic 2008

Visual Basic 2008 was released together with the Microsoft .NET Framework 3.5 on 19 November 2007.

For this release, Microsoft added many features, including:

A true conditional operator, "If(condition as boolean, truepart, falsepart)", to replace the "IIf" function. Anonymous types

Support for LINQ

Lambda expressions

XML Literals

Type Inference

Extension methods

C.1.3 Visual Basic 2010

In April 2010, Microsoft released Visual Basic 2010. Microsoft had planned to use the Dynamic Language Runtime (DLR) for that release[8] but shifted to a co-evolution strategy between Visual Basic and sister language C# to bring both languages into closer parity with one another. Visual Basic's innate ability to interact dynamically with CLR and COM objects has been enhanced to work with dynamic languages built on the DLR such as IronPython and IronRuby.[9] The Visual Basic compiler was improved to infer line continuation in a set of common contexts, in many cases removing the need for the "_" line continuation character. Also, existing support of inline Functions was complemented with support for inline Subs as well as multi-line versions of both Sub and Function lambdas.[10]

C.1.4 Visual Basic 2012

The latest version of Visual Basic .NET, which uses .NET framework 4.5.

Async Feature,

Iterators,

Call Hierarchy,

Caller Information and

Global Keyword in Namespace Statements

are some of the major features introduced in this version of VB.

C.1.5 Relation to older versions of Visual Basic

Whether Visual Basic .NET should be considered as just another version of Visual Basic or a completely different language is a topic of debate. This is not obvious, as once the methods that have been moved around and that can be automatically converted are accounted for, the basic syntax of the language has not seen many "breaking" changes, just additions to support new features like structured exception handling and short-circuited expressions. Two important data type changes occurred with the move to VB.NET. Compared to VB6, the Integer data type has been doubled in length from 16 bits to 32 bits, and the Long data type has been doubled in length from 32 bits to 64 bits. This is true for all versions of VB.NET. A 16-bit integer in all versions of VB.NET is now known as a Short. Similarly, the Windows Forms GUI editor is very similar in style and function to the Visual Basic form editor.

The version numbers used for the new Visual Basic (7, 7.1, 8, 9, ...) clearly imply that it is viewed by Microsoft as still essentially the same product as the old Visual Basic.

The things that have changed significantly are the semantics from those of an object-based programming language running on a deterministic, reference-counted engine based on COM to a fully object-oriented language backed by the .NET Framework, which consists of a combination of the Common Language Runtime (a virtual machine using generational garbage collection and a

just-in-time compilation engine) and a far larger class library. The increased breadth of the latter is also a problem that VB developers have to deal with when coming to the language, although this is somewhat addressed by the My feature in Visual Studio 2005.

The changes have altered many underlying assumptions about the "right" thing to do with respect to performance and maintainability. Some functions and libraries no longer exist; others are available, but not as efficient as the "native" .NET alternatives. Even if they compile, most converted VB6 applications will require some level of refactoring to take full advantage of the new language. Documentation is available to cover changes in the syntax, debugging applications, deployment and terminology.[11]

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
```

C.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";
        }
    }
}
```

C.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();
```

C.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();
}
```

C.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.SetFloatingPointType(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
```

C.6 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 D

Building the library and toolbox

D.1 Building and installing the numerical library

D.1.1 Downloading and installing mpMath

mpmath is a free (BSD licensed) Python library for real and complex floating-point arithmetic with arbitrary precision. It has been developed by Fredrik Johansson since 2007, with help from many contributors. mpmath runs on Python 2.5 or higher (including Python 3.x), both on 32 and 64 bit, with no other required dependencies.

mpmath can be downloaded from

<http://mpmath.org/>

The latest version is 0.19, released 2014-06-10. Download: mpmath-0.19.tar.gz., extract it, open a Windows command prompt in the extracted directory, and run

```
python setup.py install
```

The license is the New BSD License can be found in appendix [F.2.2](#))

The contributors are listed in section [E.1.1](#)

D.1.2 Running tests

It is recommended that you run mpmath's full set of unit tests to make sure everything works. The tests are located in the tests subdirectory of the main mpmath directory. They can be run in the interactive interpreter using the runtests() function:

```
import mpmath
mpmath.runtests()
```

D.2 Building the documentation and standard interfaces

D.2.1 Documentation

The following software was used to build the documentation:

miktex 2.9.4813 : [miktex](#).

texniccenter 2.02: [texniccenter](#).

After installing these programs the following additional steps are needed:

Build → Select Output Profile: Latex ⇒ PDF

Build → Define Output Profile: Latex ⇒ PDF.

In this dialogue box, select the tab "(La)Tex". Find the item "Path to (La)Tex compiler". Depending on your system, it will have an entry like

```
C:\Program Files\MiKTeX 2.9\miktex\bin\pdflatex.exe
```

Change this to

```
C:\Program Files\MiKTeX 2.9\miktex\bin\pdflatex.exe --enable-write18
```

Still in the same 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.list "%tm.nlo" -o "%tm.nls"
```

D.2.2 C interface

The source code for C interface is automatically generated when building the documentation.

D.2.3 C++ interface

The source code for C++ interface is automatically generated when building the documentation.

D.2.4 COM interface

The source code for COM interface is automatically generated when building the documentation.

D.2.5 .NET interface

The source code for .NET interface is automatically generated when building the documentation.

D.3 Building the specific interfaces

D.3.1 Excel support via Excel-DNA

ExcelDNA: [ExcelDNA](#).

Copyright (c) 2005-2009 Govert van Drimmelen

The license can be found in appendix [F.2.4](#)

The contributors are listed in section [E.2.4](#)

D.3.2 OpenOffice.org/Apache OpenOffice/LibreOffice Calc support

Support for Calc is provided by an Add-in, which connects Calc with mpFormulaPy using the built-in Basic scripting language.

There is also an external program, which is used to install the locally available functions to the global library.

D.4 Building the Toolbox GUI

The GUI is written in CSharp, using the following libraries:

D.4.1 NetOffice

NetOffice: [NetOffice](#).

Copyright (c) 2011 Sebastian Lange.

The license is the MIT License (see appendix [F.2.3](#))

The contributors are listed in section [E.2.3](#)

D.5 Other Software

D.5.1 Downloads

The following references include suggested downloads:

Microsoft .NET Framework 4 (Standalone Installer): [Microsoft .NET Framework 4](#).

Adobe Reader: [Adobe Reader](#).

gnuplot: <http://www.gnuplot.info/>.

R: <http://www.r-project.org/>.

LibreOffice: [LibreOffice](#).

LibreOffice API: [LibreOffice API](#).

OpenOffice Basic Guide: [OpenOffice Basic Guide](#).

Microsoft Office Compatibility Pack : [Microsoft Office Compatibility Pack](#).

Microsoft Access Database Engine 2010 Redistributable: [Microsoft Access Database Engine](#).

Connection Strings: [Connection Strings](#).

Chart Controls 3.5: [Chart Controls 3.5](#).

Samples Environment for Microsoft Chart Controls 3.5 [Samples 3.5](#).

Samples Environment for Microsoft Chart Controls 4.0 [Samples 4.0](#).

High performance WPF 3D Chart: [High performance WPF 3D Chart](#).

Interactive 3D bar chart custom control in WPF: [Interactive 3D bar chart custom control in WPF](#).

WPF-Print-Engine-Part-I: [WPF-Print-Engine-Part-I](#).

Printing-in-wpf: [printing-in-wpf](#).

EMF-Printer-Driver: [EMF-Printer-Driver](#).

3-D Graphics Overview: [3-D Graphics Overview](#).

csharphelper: [csharphelper](#).

vbhelper: [vbhelper](#).

SVG Adobe: [SVG Adobe](#).

D.5.2 Other References

The following references also influenced the design:

Apophenia: <http://apophenia.info/>.

Burkhard: <http://people.sc.fsu.edu/~jburkardt/>.

Gladman: <http://www.gladman.me.uk/>.

Arndt: <http://www.jjj.de/fxt/fxtbook>.

Vogt: <http://www.axelvogt.de/axalom/index.html>.

L'Ecuyer: <http://www.iro.umontreal.ca/~lecuyer/>.

CVM: <http://www.cvmlib.com/>.

Alglib: <http://www.alglib.net/>.

GSL: <https://www.gnu.org/software/gsl/>.

FFTW: <http://www.fftw.org/>.

Function Parser: <http://warp.povusers.org/FunctionParser/fparser.html>.

Xnumbers v.6.0 for Excel 2010: <http://www.thetropicalevents.com/Xnumbers60.htm>.

NIST: <http://dlmf.nist.gov/>.

Mathjax: <http://docs.mathjax.org/en/latest/start.htmltex-and-latex-input>.

Vbsedit: <http://www.vbsedit.com/>.

STATA: [STATA](#).

gpower3: [gpower3](#).

TIOBE: [TIOBE Programming Community Index](#).

PYPL: [PYPL PopularitY of Programming Language index](#).

ilnumerics: [ilnumerics](#).

pdfBooks: <http://www.pdfbook.co.ke/>.

D.5.3 Previous version

SQLite: <http://www.sqlite.org/about.html>.

System.Data.SQLite: [System.Data.SQLite](#).

SQLite2009 Pro Enterprise Manager: [SQLite2009 Pro Enterprise Manager](#).

Process Caller: [Process Caller](#).

Interprocess-communication: [Interprocess-communication](#).

MDI Tabcontrol: [MDI Tabcontrol](#).

ScintillaNET: [ScintillaNET](#).

Fast-Colored-TextBox: [Fast-Colored-TextBox](#).

Spreadsheet-Control: [Spreadsheet-Control](#).

Spreadsheet-Control: [Spreadsheet-Control](#).

DataGridView Printer: [DataGridView Printer](#).

HTML Editor: [HTML Editor](#).

HTML Editor: [HTML Editor](#).

Formula Engine: [Formula Engine](#).

Grammatica: [Grammatica](#).

Excel xll add-in library: [Excel xll add-in library](#).

jni4net: [jni4net](#).

jedit: <http://www.jedit.org/>.

Setting up jedit for compiling: <http://courses.cs.washington.edu/courses/cse413/02au/jEdit.html>.

FSharp MSDN: [FSharp MSDN](#).

FSharp Wiki: [FSharp Wiki](#).

FSharp Home: [FSharp Home](#).

FSharp .NET: [FSharp .NET](#).

Tsunami: [Tsunami](#).

Gnuplot CSharp: [Gnuplot CSharp](#).

oxyplot: [oxyplot](#).

Propertygrid: [Propertygrid](#).

MSDN MS Office: [MSDN MS Office](#).

Open XML SDK 2.5 for Microsoft Office: [Open XML SDK 2.5 for Microsoft Office](#).

famfamfam-silk-icons 16x16: [famfamfam-silk-icons](#).

famfamfam-silk-icons 32x32: [famfamfam-silk-icons](#).

silk-companion-1-icons: [silk-companion-1-icons](#).

Fugue Icons: [Fugue Icons](#).

Wix Toolset: [Wix Toolset](#).

Wix Edit: [Wix Edit](#).

Speech: [Speech Synthesizer](#).

A-Calculation-Engine-for-NET: [A-Calculation-Engine-for-NET](#).

D.6 To Do

D.6.1 New GUI

- To be developed with SharpDevelop
- all mpFormulaPy panels to be used as standalone (with messaging, from mp.), as in-process COM server, as .NET dll.
- mpFunction panel for selection of functions and macros (procedures)
- mpChart panel for selection of XLM charts
- mpOptions panel for selection of options
- mpFormulaPy pane: Display of images (EMF, JPG)
- mpFormulaPy pane: Display of HTML output, incl PDF, navigation pane, export to Word
- mpFormulaPy pane: Display of tabular output
- mpFormulaPy pane: Navigator for the output of all zzFormulas, including ranges, matrices and charts, and inspector of full precision in decimal and binary
- optional: Support for simple scripts (VB.NET, cSharp, JScript2010, ReoScript, VBScript, JScript).
- optional: Transfer of simple VB.Net and CSharp scripts to SharpDevelop or Visual Studio.
- optional: Transfer of charts to SharpDevelop for editing.

D.6.2 GUI

- Work out and implement starting Java with jni4net.
- Add item for F# in project explorer
- Redesign the project explorer so that it becomes usable if only certain types of files are present
- Redesign the GUI as a DLL that can be run from other applications.
- Figure out necessary changes to GUI to support ultra high resolutions (DPI aware)
- Add "Program finished" and timing after returning from "Program started. Please stand by.."
- Avoid opening of output after call to forms or just opening word.
- Display an error message when an executable like Matlab cannot be found BEFORE trying to run the program
- Implement property pages for code modules, including .exe, linkage, post-build actions and post-run actions, save and load as xml
- Implement full options module including saving and loading as xml.

- Implement live connection to workbooks as new category in project tree. Implement `mp.GetActiveWorkbook()`.
- Implement starting VBA files as Add-ins (needs to activate "Trust access to the VBA object model")

D.6.3 C++ Library

- Fix Win64 Eigen Decimal output crashes
- Fix Win64 Eigen mpfrb crashes
- Implement Random module

D.6.4 Manual

- Make a release version of the manual, removing all currently unsupported features, with automatic generation of interface files
- Add a roadmap appendix to the manual
- Re-introduce description of all Toolbox components in introduction (like NetOffice etc)
- Describe how to make the build for the `mpFormulaPy.msi` file
- Describe how to make the build for the GUI, including the changes to ScintillaNET etc.
- Describe how to make the build for the xll function, the main dialogue, and the ribbon and task pane project for Excel using Excel DNA
- Descriptions of nonparametric procedures, as currently implemented in Stats32
- Complete chapter on discrete distributions
- Transfer remaining text from Word
- Figure out recursion relation for noncentral t, and document them

D.6.5 New Programming

- Figure out recursion relation for noncentral t, and implement it in doubly noncentral t
- Implement doubly noncentral F with recursion relations
- Wilks noncentral: Implement algorithm by Butler and multivariate hypergeometric functions
- Implement 2nd noncentral moment of Spearman rho
- Implement R^2 as in Benton/Boost
- Implement integral versions of noncentral Chi2, t, F
- Implement hypergeometric functions from AMath

Appendix E

Acknowledgements

E.1 Contributors to libraries used in the numerical routines

E.1.1 Contributors to mpMath

The following text has been copied from the mpMath manual (0.19):

xxxx

E.1.2 Contributors to gmpy2

The following text has been copied from the gmpy2 manual (2.05):

xxxx

E.2 Contributors to libraries used in the GUI

E.2.1 Contributors to Sharp Develop

The site: <https://github.com/icsharpcode/SharpDevelop/wiki/Contributors> states the following:

”Non-Developers: Christoph Wille (PM), Bernhard Spuida (Kalfaktor).

Developers: Daniel Grunwald (Technical Lead), Matt Ward, David Srbecký (Debugger), Siegfried Pammer, Martin Koníček, Peter Forstmeier (SharpDevelop Reports), Tomáš Linhart, Kumar Devvrat, Eusebiu Marcu.

Past developers:

This list is by no means exhaustive: Mike Krüger (Project Founder), Alexandre Semenov, Andrea Paatz, Christian Hornung, David Alpert, Denis ERCHOFF, Dickon Field, Georg Brandl, Ifko Kovalčík, Itai Bar-Haim, Ivan Shumilin, John Reilly, John Simons, Justin Dearing, Markus Palme, Mathias Simmack, Matt Everson, Nathan Allan, Nikola Kavaldjiev, Philipp Maihart, Poul Staunsgaard, Robert Pickering, Robert Zaunere, Roman Taranchenko, Russell Wilkins, Scott Ferrett, Sergej Andrejev, Shinsaku Nakagawa, Tomasz Tretkowski, Troy Simpson.”

E.2.2 Contributors to Unmanaged Exports

The main author of Unmanaged Exports is Robert Giesecke.

E.2.3 Contributors to NetOffice

The main author of NetOffice is Sebastian Lange.

E.2.4 Contributors to Excel-DNA

The main author of Excel-DNA is Govert van Drimmelen.

E.2.5 Contributors to jni4net

The main author of jni4net is Pavel Šavara.

E.2.6 System.Data.SQLite

The main author of System.Data.SQLite is R. Hipp.

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:

âĂća) The modified work must itself be a software library. âĂćb) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change. âĂćc) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License. âĂćd) 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.1.3 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 New BSD License (for mpMath)

New BSD License (for mpMath)

Copyright (c) 2005-2013 Fredrik Johansson and mpmath contributors
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. c. Neither the name of mpmath nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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.

F.2.4 Excel-DNA License

Excel-DNA License

Excel-DNA License Copyright (C) 2005-2009 Govert van Drimmelen

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Govert van Drimmelen govert@icon.co.za

Appendix G

Mathematical Notation

Table G.1: Notation related to Sets

\mathbb{N}	Set of natural numbers
\mathbb{Z}	Set of integer numbers
\mathbb{Q}	Set of rational numbers
\mathbb{R}	Set of real numbers
\mathbb{C}	Set of complex numbers

Part VII

Back Matter

Bibliography

- Abramowitz, M., & Stegun., I.A. 1970. *Handbook of Mathematical Functions*. New York: Dekker. Available as <http://www.math.sfu.ca/~cbm/aands/>. 184, 265, 364
- Bailey, David H., Jeyabalan, Karthik, & Li, Xiaoye S. 2004. A comparison of three high-precision quadrature schemes. *Experimental Mathematics*, **14**, 317–329. Available as <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/quadrature.pdf>. 546
- Bailey, D.H. 2006. *Tanh-Sinh High-Precision Quadrature*. Manuscript. Available as <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/dhb-tanh-sinh.pdf>. 546
- Benninga, Simon. 2008. *Financial Modeling*. 3rd edn. The MIT Press. Online resource: <http://simonbenninga.com/fm3contents.htm>. 567
- Benninga, Simon. 2010. *Principles of Finance with Excel*. 2nd edn. Oxford University Press. Online resource: <http://simonbenninga.com/pfe2instructorpage.htm>. 567
- Box, G.E.P. 1949. A general distribution theory for a class of likelihood criteria. *Biometrika*, **36**, 317–346. 244
- 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>. 398, 404
- Canal, L. 2005. A normal approximation for the chi-square distribution. *Computational Statistics & Data Analysis*, **48**, 803–808. Available at <http://www.sciencedirect.com/science/article/pii/S0167947304001069>. 235
- Carlson, B. C., & Gustafson, J. L. 1994. Asymptotic Approximations for Symmetric Elliptic Integrals. *Siam Journal on Mathematical Analysis*, **25**. 431
- Carlson, B.C. 1995. Numerical computation of real or complex elliptic integrals. *Numerical Algorithms*, **10**(1), 13–26. 431
- 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>. 251
- 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. 245

- Cheney, Ward, & Kincaid, David. 2008. *Numerical Mathematics and Computing*. 6th edn. Thomson Brooks/Cole. 216
- Chernick, Michael R. 2008. *Bootstrap methods : a guide for practitioners and researchers*. 2nd edn. John Wiley & Sons. 216
- 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>. 251
- 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*. 350
- Cuyt, A.A.M., Verdonk, B., Becuve, S., & Kuterna, P. 2001. A remarkable example of catastrophic cancellation unraveled. *Computing*, **66**(3), 309–320. 6
- Davis, A.W. 1971. Percentile approximations for a class of likelihood ratio criteria. *Biometrika*, **58**, 349–356. 244
- Day, Alastair L. 2010. *Mastering financial mathematics in Microsoft Excel : a practical guide for business calculations*. 2nd edn. Pearson Education Limited. 567
- 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>. 334
- 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>. 456
- Feuersänger, Christian. 2014. *Manual for Package pgfplots: 2D/3D Plots in LATEX, Version 1.11*. Available at <http://sourceforge.net/projects/pgfplots>. 7
- 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>. 395
- 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://citeserx.ist.psu.edu/viewdoc/download?doi=10.1.1.212.6321&rep=rep1&type=pdf>. 6
- Gleser, L. J. 1976. A canonical representation for the noncentral Wishart distribution useful for simulation. *Journal of the American Statistical Association*, 690–695. 237
- 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://citeserx.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. 273
- Guenther, William C. 1977. Calculation of Factors for Tests and Confidence Intervals Concerning the Ratio of Two Normal Variances. *The American Statistician*, **31**, 175–177. 244

- Harkness, W. L., & Katz, L. 1964. Comparison of the Power Functions for the Test of Independence in 2 \times 2 Contingency Tables. *The Annals of Mathematical Statistics*, **35**(3), 1115–1127. Article Stable URL: <http://www.jstor.org/stable/2238241>. 251
- 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 Reconfgurable Supercomputing Network Workshop Queen's University, Belfast, January 15-16, 2009*. Available as <http://cpc.cs.qub.ac.uk/MRSN/higham.pdf>. 5
- 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. 6
- Johansson, Fredrik, *et al.* 2013 (December). *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*. <http://mpmath.org/>. 2
- Johnson, N. L., Kotz, S., & Balakrishnan, N. 1994.. *Continuous Univariate Distributions, Volume 1*. 2nd edn. Wiley & Sons, Inc., New York-London-Sydney-Toronto. 216
- Johnson, N. L., Kotz, S., & Balakrishnan, N. 1995.. *Continuous Univariate Distributions, Volume 2*. 2nd edn. Wiley & Sons, Inc., New York-London-Sydney-Toronto. 216
- Klemens, Ben. 2008. *Modeling with Data: Tools and Techniques for Scientific Computing*. 1st edn. Princeton University Press. Available as http://ben.klemens.org/pdfs/gsl_stats.pdf. 278
- 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#>. 251
- 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>. 60
- Krüger, M., Grunwald, D., Ward, M., Srbecký, D., Pammer, S., Forstmeier, P., *et al.* 2013. *SharpDevelop: a free IDE for .NET programming languages*. Online resource: <http://www.icsharpcode.net/OpenSource/SD/>. 3
- Lange, Kenneth. 2010. *Numerical Analysis for Statisticians*. 2nd edn. Springer Science+Business Media. 216
- Lange, Sebastian. 2012. *NetOffice: .NET Wrapper Assemblies for accessing MS Office applications*. Online resource: <http://netoffice.codeplex.com/>. 3
- 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. 251

- McKinney, Wes. 2012. *Python for Data Analysis*. O'Reilly Media. Online resource: <http://shop.oreilly.com/product/0636920023784.do>. 611
- Monahan, John F. 2011. *Numerical methods of statistics*. 2nd edn. Cambridge University Press. 216
- Muller, Keith E. 2001. Computing the confluent hypergeometric function, $M(a,b,x)$. *Numerische Mathematik*, **90**(1), 179–196. 395
- Ogita, T., Rump, S.M., & Oishi, S. 2005. Accurate sum and dot product. *SIAM J. Sci. Comput.*, **26**, 1955–1988. Available as <http://www.ti3.tu-harburg.de/paper/rump/0gRuUi05.pdf>. 278
- 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/>. 184, 438
- 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. 258
- Pearson, J. 2009. Computation of Hypergeometric Functions. *Masters thesis*. Available as http://people.maths.ox.ac.uk/porterm/research/pearson_final.pdf. 395
- 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. 233, 272
- Rinne, H. 2008. *Taschenbuch der Statistik*. 4 edn. Frankfurt, M. : Deutsch. 216, 276
- Sahai, H., & Thompson, W.O. 1974. Comparisons of approximations to the percentiles of the t, chi-square, and F distributions. *JSCS*, **3**, 81–93. 243
- Temme, N. M. 1979. The asymptotic expansion of the incomplete gamma functions. *SIAM J. Math. Anal.*, **10**, 757–766. 328
- Temme, N. M. 1994. A Set of Algorithms for the Incomplete Gamma Functions. *Probability in the Engineering and Informational Sciences*, **8**(4), 291–307. 328
- 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>. 223
- 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>. 251
- van Drimmelen, Govert. 2013. *Excel-DNA: an independent project to integrate .NET into Excel*. Online resource: <http://excel-dna.net/>. 3
- Van Hauwermeiren, M., & Vose, D. 2009. *A Compendium of Distributions*. [ebook]. Vose Software, Ghent, Belgium. Available from <http://www.vosesoftware.com>. 216

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>. 216, 227, 231

Wikipedia contributors. 2013. *Pareto distribution* — Wikipedia, The Free Encyclopedia. Available at http://en.wikipedia.org/w/index.php?title=Pareto_distribution&oldid=559561732.

Nomenclature

$\chi_{\nu,\alpha}^2$	α quantile of the central χ^2 -distribution with ν degrees of freedom (page 188)
$\Gamma(x)$	Gamma Function (page 154)
$\Gamma_p(x)$	Multivariate Gamma Function (page 592)
\mathbb{C}	Set of complex numbers (page 887)
\mathbb{N}	Set of natural numbers (page 887)
\mathbb{Q}	Set of rational numbers (page 887)
\mathbb{R}	Set of real numbers (page 887)
\mathbb{Z}	Set of integer numbers (page 887)
$\Phi(x)$	CDF of the standardized normal distribution (page 217)
$\phi(x)$	pdf of the standardized normal distribution (page 216)
$\Phi^{-1}(\alpha)$	Inverse CDF of the standardized normal distribution (page 217)
$F_F(m, n, x)$	CDF of the central F -distribution (page 196)
$f_F(m, n, x)$	pdf of the central F -distribution (page 196)
$F_N(x; \mu, \sigma^2)$	CDF of the normal distribution with mean μ and variance σ^2 (page 217)
$F_N(x; \mu, \sigma^2)$	pdf of the normal distribution with mean μ and variance σ^2 (page 216)
$F_N^{-1}(\alpha; \mu, \sigma^2)$	Inverse CDF of the normal distribution with mean μ and variance σ^2 (page 217)
$f_R(r, N; \rho)$	pdf of the Distribution of the Sample Correlation Coefficient (page 616)
$F_t(n, x)$	CDF of the central t -distribution (page 224)
$f_t(n, x)$	pdf of the central t -distribution (page 224)
$F_{\chi^2}(n, x)$	CDF of the central chi-square distribution (page 187)
$f_{\chi^2}(n, x)$	pdf of the central chi-square distribution (page 187)
$F_{\chi^2}(n, x; \lambda)$	CDF of the noncentral chi-square distribution (page 642)
$f_{\chi^2}(n, x; \lambda)$	pdf of the noncentral chi-square distribution (page 641)
$F_{\nu_1, \nu_2, \alpha}$	α quantile of the central F -distribution with ν_1 and ν_2 degrees of freedom (page 196)
$F_{\text{Beta}'}(x; a, b, \lambda)$	CDF of the (singly) noncentral Beta-distribution (page 649)
$f_{\text{Beta}'}(x; a, b, \lambda)$	pdf of the (singly) noncentral Beta-distribution (page 649)
$F_{\text{Beta}}(a, b, x)$	CDF of the central Beta-distribution (page 178)
$f_{\text{Beta}}(a, b, x)$	pdf of the central Beta-distribution (page 178)
$F_{\text{Bin}}(n, k; p)$	CDF of the binomial distribution (page 183)
$f_{\text{Bin}}(n, k; p)$	pmf of the binomial distribution (page 183)
$F_{\text{NegBin}}(n, k; p)$	CDF of the negative binomial distribution (page 213)
$f_{\text{NegBin}}(n, k; p)$	pmf of the negative binomial distribution (page 213)
$f_{F''}(x; m, n)$	pdf of the doubly noncentral F -distribution (page 671)
$f_{F'}(x; m, n)$	CDF of the (singly) noncentral F -distribution (page 665)
$f_{F'}(x; m, n)$	pdf of the (singly) noncentral F -distribution (page 665)
$F_{R^2}(x; p, n, \rho^2)$	CDF of the Square of the Multiple Sample Correlation Coefficient (page 625)
$f_{R^2}(x; p, n, \rho^2)$	pdf of the Square of the Multiple Sample Correlation Coefficient (page 625)

$F_{t''}(t; n; \delta, \theta)$	CDF of the doubly noncentral t-square distribution (page 660)
$f_{t''}(t; n; \delta, \theta)$	pdf of the doubly noncentral t-square distribution (page 659)
$F_{t'}(n, x, \delta)$	CDF of the (singly) noncentral t-distribution (page 653)
$f_{t'}(n, x, \delta)$	pdf of the (singly) noncentral t-distribution (page 652)
$I_\nu(z)$	Modified Bessel function of the first kind of real order ν (page 152)
$J_\nu(z)$	Bessel function of the first kind of real order ν (page 152)
$K_\nu(z)$	Modified Bessel function of the second kind of real order ν (page 153)
$N_{Rho}(\alpha, \beta, \tilde{\rho})$	Sample size function of the noncentral t -distribution for a given confidence level α , power β and modified noncentrality parameter $\tilde{\rho}$ (page 623)
$N_{t''}(\alpha, \beta, \tilde{\rho})$	Sample size function of the doubly noncentral t -distribution for a given confidence level α , power β and modified noncentrality parameter $\tilde{\rho}$ (page 658)
$N_{t''}(\alpha, \beta, \tilde{\rho})$	Sample size function of the doubly noncentral t -distribution for a given confidence level α , power β and modified noncentrality parameter $\tilde{\rho}$ (page 663)
$t_{\nu, \alpha}$	α quantile of the central t -distribution with ν degrees of freedom (page 225)
$T_{Owen}(a, b)$	Owen's T-Function (page 633)
$t_{n, \delta; \alpha}$	α quantile of the noncentral t -distribution with ν degrees of freedom and noncentrality parameter δ (page 655)
$Y_\nu(z)$	Bessel function of the second kind of real order ν (page 152)
z_α	α quantile of the standardized normal distribution (page 217)
${}_0\tilde{F}_1(b; x)$	Regularized Confluent Hypergeometric Limit Function (page 422)
${}_0F_1(a; \Omega)$	Confluent Hypergeometric Limit Function for Matrix Argument (page 596)
${}_1\tilde{F}_1(a, b; z)$	Kummer's Regularized Confluent Hypergeometric Function (page 424)
${}_1F_1(a, b; \Omega)$	Kummer's Confluent Hypergeometric Function for Matrix Argument (page 595)
${}_1F_1(a, b; z)$	Kummer's Confluent Hypergeometric Function (page 423)
${}_2\tilde{F}_1(a, b; c; z)$	Gauss Regularized Hypergeometric Function (page 431)
${}_2F_1(a, b; c; \mathbf{T})$	Gauss Hypergeometric Function of Matrix Argument (page 594)
${}_2F_1(a, b; c; x)$	Gauss Hypergeometric Function (page 429)
CDF	cumulative distribution function (page 170)
pdf	probability density function (page 170)
pmf	probability mass function (page 170)

Index

- Functions
- EFFECT, 650
- Multiprecision Functions
- , 101
 - *, 103, 128
 - +, 97, 127
 - .Div, 107
 - .DotProd, 103
 - .EQ, 111
 - .GE, 111
 - .GT, 111
 - .LE, 111
 - .LSH, 103
 - .LT, 112
 - .Minus, 101
 - .Mod, 108
 - .NE, 112
 - .Plus, 97, 127
 - .Pow, 109
 - .RSH, 107
 - .Times, 103
 - .TimesMat, 103, 128
 - .nint_distance, 118
 - /, 107
 - =, 111, 112
 - $\hat{}$, 109
 - ABS, 93
 - abs, 93
 - ACCRINT, 583
 - ACCRINTM, 584
 - ACOS, 186
 - acos, 186
 - ACOSH, 194
 - acosh, 194
 - ACOT, 191
 - acot, 191
 - ACOTH, 196
 - acoth, 196
 - AGGREGATE, 298
 - agm, 351
 - airyai, 370
 - airyaizero, 373
 - airybi, 371
 - airybizer0, 373
 - almosteq, 112
 - altzeta, 443
 - AMORDEGRC, 597
 - AMORLINC, 596
 - angerj, 366
 - ANOVA1, 302
 - ANOVA2, 303
 - apery, 134
 - appellf1, 418
 - appellf2, 419
 - appellf3, 419
 - appellf4, 420
 - ARABIC, 77
 - arange, 121
 - arg, 94
 - ASIN, 184
 - asin, 184
 - ASINH, 193
 - asinh, 193
 - AssociatedLaguerreMpMath, 394
 - ATAN, 188
 - atan, 188
 - ATAN2, 190
 - Atan2, 190
 - ATANH, 195
 - atanh, 195
 - AVEDEV, 289
 - AVERAGE, 285
 - AVERAGEA, 285
 - AVERAGEIF, 285
 - AVERAGEIFS, 285
 - backlunds, 450
 - barnesg, 323
 - BASE, 80

bei, 362
 bell, 469
 ber, 362
 bernfrac, 464
 bernoulli, 463
 bernpoly, 465
 BESSELI, 198
 besseli, 355
 BesselIeMpMath, 352
 BESSELJ, 198
 besselj, 352
 besseljzero, 358
 BESSELK, 199
 besselk, 356
 BesselKeMpMath, 352
 BESSELY, 198
 bessely, 354
 besselyzero, 359
 Beta, 201
 beta, 333
 BETA.DIST, 223
 BETA.INV, 225
 BETADIST, 223
 BetaDist, 223
 BetaDistInfo, 225
 BetaDistInv, 224
 BetaDistRandom, 226
 betainc, 333
 BETAINV, 224
 bihyper, 416
 BIN2DEC, 77
 BIN2HEX, 77
 BIN2OCT, 78
 BINOM.DIST, 228
 BINOM.DIST.RANGE, 229
 BINOM.INV, 230
 BINOMDIST, 228
 binomial, 318
 BinomialDist, 228
 BinomialDistInfo, 230
 BinomialDistInv, 229
 BinomialDistRandom, 231
 BITAND, 110
 BITLSHIFT, 106
 BITOR, 110
 BITRSHIFT, 108
 BITXOR, 110
 catalan, 134
 cbrt, 151
 CDist, 232
 CDistInfo, 236
 CDistInv, 234
 CDistRan, 237
 ceil, 84
 CEILING, 83
 CEILING.MATH, 84
 CEILING.PRECISE, 83
 Chart, 15
 chebyfit, 553
 chebyt, 387
 chebyu, 387
 chi, 342
 CHIDIST, 232
 CHIINV, 234
 CHISQ.DIST, 233
 CHISQ.DIST.2T, 233
 CHISQ.DIST.RT, 233
 CHISQ.INV, 234
 CHISQ.INV.RT, 235
 CHISQ.TEST, 303
 CHISQDIST, 232
 CHISQINV, 235
 CHITEST, 303
 cholesky, 210
 chop, 112
 ci, 340
 clcos, 457
 clsinlog, 456
 cohen_alt, 529
 COMBIN, 202
 COMBINA, 202
 COMPLEX, 91
 CONFIDENCE, 300
 CONFIDENCE.NORM, 300
 CONFIDENCE.T, 300
 conj, 96
 CONVERT, 280
 CORREL, 305
 COS, 160
 cos, 160
 COSH, 174
 cosh, 174
 cosm, 483
 COT, 168
 cot, 168
 COTH, 182
 coth, 182
 coulombc, 377

coulombf, 376
coulombg, 377
COUNT, 277
COUNTA, 277
COUNTBLANK, 277
COUNTIF, 278
COUNTIFS, 278
COUPDAYBS, 579
COUPDAYS, 580
COUPDAYSNC, 580
COUPNCD, 580
COUPNUM, 581
COUPPCD, 581
COVAR, 305
COVARIANCE.P, 305
COVARIANCE.S, 305
CRITBINOM, 229
CSC, 166
csc, 166
CSCH, 180
csch, 180
CUMIPMT, 601
CUMPRINC, 602
cyclotomic, 475
DATE, 571
DATEVALUE, 571
DAVERAGE, 311
DAY, 568
DAYS, 569
DAYS360, 576
DB, 595
DCOUNT, 310
DCOUNTA, 311
DDB, 594
DebyeMpMath, 456
DEC2BIN, 78
DEC2HEX, 78
DEC2OCT, 78
DECIMAL, 81
degree, 134
DEGREES, 156
degrees, 156
DELTA, 113
DEVSQ, 289
DGET, 310
diff, 531
differint, 535
diffs, 533
diffsexp, 534
diffsprod, 534
digamma, 332
dilog, 456
dirichlet, 444
DirichletBetaMpMath, 444
DirichletEtaMpMath, 443
DirichletLambdaMpMath, 444
DISC, 584
DMAX, 313
DMIN, 313
DOLLARDE, 608
DOLLARFR, 608
DPRODUCT, 310
DSTDEV, 312
DSTDEVP, 312
DSUM, 311
DURATION, 581
DVAR, 312
DVARP, 312
e, 134
e1, 337
EASTERSUNDAY, 571
EDATE, 572
EFFECT, 602
ei, 336
eig, 493
eigh, 494
ellipe, 427
ellipef, 427
ellipf, 425
ellipfun, 438
ellipk, 425
ellippi, 428
ellippif, 429
elliprc, 432
elliprd, 434
elliprf, 431
elliprg, 434
elliprj, 433
EOMONTH, 572
ERF, 199
erf, 343
ERF.PRECISE, 199
ERFC, 199
erfc, 344
ERFC.PRECISE, 200
erfi, 344
erfinv, 345
euler, 134

eulernum, 467
 eulerpoly, 467
 Eval, 15
 EVEN, 86
 EXP, 136
 exp, 136
 exp10, 140
 exp2, 140
 expint, 337
 expj, 138
 expjpi, 139
 expm, 481
 expm1, 139
 EXPON.DIST, 238
 EXPONDIST, 238
 ExponentialDist, 238
 ExponentialDistInfo, 239
 ExponentialDistInv, 239
 ExponentialDistRandom, 240
 F.DIST, 241
 F.DIST.RT, 242
 F.INV, 243
 F.INV.RT, 243
 F.TEST, 302
 fabs, 93
 fac, 315
 fac2, 316
 FACT, 202
 FACTDOUBLE, 202
 Factorial, 315
 fadd, 97
 FDIST, 241
 FDist, 241
 FDistInfo, 245
 FDistInv, 242
 FDistRan, 245
 fdiv, 107
 fdot, 105
 FermiDirac3HalfMpMath, 453
 FermiDiracHalfMpMath, 453
 FermiDiracIntMpMath, 453
 FermiDiracPHalfMpMath, 453
 ff, 320
 fib, 461
 fibonacci, 461
 findpoly, 562
 findroot, 499
 FINV, 243
 FISHER, 306
 FISHERINV, 307
 FLOOR, 84
 floor, 85
 FLOOR.MATH, 85
 FLOOR.PRECISE, 84
 fmod, 109
 fmul, 103
 fneg, 102
 FORECAST, 308
 fourier, 555
 fouriereval, 556
 fprod, 105
 frac, 87
 fraction, 120
 FREQUENCY, 278
 fresnelc, 348
 fresnels, 348
 frexp, 90
 fsub, 101
 fsum, 98
 FTEST, 302
 FV, 598
 FVSCHEDULE, 603
 GAMMA, 200
 gamma, 324
 GAMMA.DIST, 247
 GAMMA.INV, 248
 GAMMADIST, 247
 GammaDist, 247
 GammaDistInfo, 249
 GammaDistInv, 248
 GammaDistRandom, 249
 gammainc, 327
 GAMMAINV, 248
 GAMMALN, 200
 GAMMALN.PRECISE, 200
 GammaPDerivativeMpMath, 328
 GammaPinvMpMath, 330
 GammaPMPmath, 328
 gammaprod, 325
 GammaQinvMpMath, 330
 GammaQMPmath, 328
 GAUSS, 262
 GCD, 203
 gegenbauer, 390
 GeneralizedExponentialIntegralEpMpMath, 338
 GEOMEAN, 286
 GESTEP, 113

glaisher, 134
grampoint, 450
GROWTH, 207
hankel1, 361
hankel2, 361
HARMEAN, 286
harmonic, 332
hermite, 392
HEX2BIN, 79
HEX2DEC, 79
HEX2OCT, 79
Histogram, 279
HOUR, 568
hurwitz, 441
hyp0f1, 396
hyp1f1, 398
hyp1f2, 407
hyp2f0, 400
hyp2f1, 404
hyp2f2, 407
hyp2f3, 408
hyp3f2, 408
hyper, 410
hyper2d, 417
hypercomb, 411
hyperfac, 322
Hypergeometric0F1RegularizedMpMath, 397
Hypergeometric1F1RegularizedMpMath, 399
Hypergeometric2F1RegularizedMpMath, 406
HypergeometricDist, 251
HypergeometricDistInfo, 253
HypergeometricDistInv, 252
HypergeometricDistRandom, 253
hyperu, 400
HYPGEOM.DIST, 252
HYPGEOMDIST, 251
hypot, 150
IBetaMpMath, 335
IBetaNonNormalizedMpMath, 334
identify, 557
im, 92
IMABS, 93
IMAGINARY, 92
IMARGUMENT, 94
IMCONJUGATE, 95
IMCOS, 160
IMCOSH, 174
IMCOT, 168
IMCOTH, 182
IMCSC, 166
IMCSCH, 180
IMDIV, 107
IMEXP, 136
IMLN, 141
IMLOG10, 143
IMLOG2, 144
IMPOWER, 146
IMPRODUCT, 105
IMREAL, 92
IMSEC, 164
IMSECH, 178
IMSIN, 158
IMSINH, 172
IMSQRT, 149
IMSUB, 101
IMSUM, 98
IMTAN, 162
IMTANH, 176
inf, 134
INT, 86
INTERCEPT, 308
INTRATE, 585
InverseTangentMpMath, 454
IPMT, 601
IRR, 604
ISEVEN, 117
isfinite, 115
isinf, 116
isint, 117
isnan, 116
isnormal, 115
ISNUMBER, 115
ISODD, 117
ISOWEEKNUM, 570
ISPMT, 603
j0, 354
j1, 354
jacobi, 389
jtheta, 436
kei, 363
ker, 362
kfrom, 424
khinchin, 134
kleinj, 440
KURT, 291
laguerre, 393
LaguerreLMpMath, 394
lambertw, 350

LARGE, 293
 LCM, 204
 ldexp, 90
 legendre, 382
 LegendreChiMpMath, 454
 legenp, 383
 legenq, 384
 lerchphi, 452
 levin, 525
 li, 339
 limit, 520
 LINEST, 206
 linspace, 121
 LN, 141
 ln, 142
 LnBetaMpMath, 333
 lnp1, 144
 LOG, 142
 log, 142
 LOG10, 143
 log10, 144
 log2, 144
 Logb, 142
 LOGEST, 207
 loggamma, 326
 LOGINV, 256
 logm, 487
 LOGNORM.DIST, 255
 LOGNORM.INV, 256
 LogNormalDist, 254
 LognormalDistInfo, 256
 LognormalDistInv, 255
 LognormalRandom, 257
 LOGNORMDIST, 255
 lommels1, 368
 lommels2, 368
 lu, 214
 lu_solve, 212
 mag, 118
 mangoldt, 476
 matrix, 123
 MatrixAdd, 127
 MatrixMul, 128
 MAX, 292
 MAXA, 292
 MDETERM, 205
 MDURATION, 582
 MEDIAN, 292
 mijerg, 413
 mertens, 134
 mfrom, 423
 MIN, 292
 MINA, 292
 MINUTE, 568
 MINVERSE, 205
 MIRR, 605
 MMULT, 128
 mnorm, 208
 MOD, 109
 mod, 108
 MODE, 293
 MODE.MULT, 293
 MODE.SNGL, 293
 MONTH, 569
 mpc, 91
 MROUND, 88
 MULTINOMIAL, 203
 MUNIT, 123
 nan, 135
 ncdf, 347
 NDist, 261
 NDistInv, 263
 NegativeBinomialDist, 258
 NegativeBinomialDistInfo, 259
 NegativeBinomialDistInv, 259
 NegativeBinomialDistRandom, 260
 NEGBINOM.DIST, 258
 NEGBINOMDIST, 258
 NETWORKDAYS, 576
 NETWORKDAYS.INTL, 577
 nint, 87
 NOMINAL, 602
 NonNormalisedGammaPMpMath, 329
 NonNormalisedGammaQMpMath, 329
 norm, 208
 NORM.DIST, 261
 NORM.INV, 264
 NORM.S.DIST, 262
 NORM.S.INV, 264
 NormalDistInfo, 264
 NormalRandom, 265
 NORMDIST, 261
 NORMINV, 263
 NORMSDIST, 262
 NORMSINV, 264
 NOW, 573
 npdf, 347
 NPER, 599

nprod, 517
NPV, 605
nsum, 506
nsum2d, 512
nsum3d, 513
nthroot, 151
nzeros, 447
OCT2BIN, 80
OCT2DEC, 80
OCT2HEX, 80
ODD, 86
ODDFPRICE, 585
ODDFYIELD, 586
ODDLPRICE, 587
ODDLYIELD, 587
odefun, 548
Options, 15
pade, 552
pcfd, 379
pcfū, 379
pcfū, 380
pcfū, 381
PDURATION, 600
PEARSON, 306
PERCENTILE, 294
PERCENTILE.EXC, 294
PERCENTILE.INC, 294
PERCENTRANK, 295
PERCENTRANK.EXC, 295
PERCENTRANK.INC, 295
PERMUT, 203
PERMUTATIONA, 203
phase, 94
PHI, 262
phi, 134
PI, 133
pi, 133
PMT, 599
POISSON, 266
POISSON.DIST, 266
PoissonDist, 266
PoissonDistInfo, 267
PoissonDistInv, 267
PoissonDistRan, 268
polar, 91
polyexp, 458
polygamma, 331
polylog, 455
polyroots, 497
polyval, 496
POWER, 146
power, 146
powm, 489
powm1, 147
PPMT, 601
PRICE, 588
PRICEDISC, 589
PRICEMAT, 589
primepi, 472
primepi2, 472
primezeta, 459
PROB, 297
PRODUCT, 104
psi, 331
pslq, 564
PV, 598
qbarfrom, 423
qfac, 478
qfrom, 422
qgamma, 478
qhyper, 480
qp, 477
qr, 214
quad, 537
quad2d, 540
quad3d, 541
quadosc, 542
QUARTILE, 295
QUARTILE.EXC, 296
QUARTILE.INC, 296
QUOTIENT, 89
RADIANS, 156
radians, 157
RAND, 120
rand, 120
RANDBETWEEN, 120
RANK, 296
RANK.AVG, 297
RANK.EQ, 297
RATE, 600
re, 92
RECEIVED, 590
rect, 91
RelativePochhammerMpMath, 319
residual, 212
rf, 319
rgamma, 325
richardson, 522

riemannr, 473
 ROMAN, 77
 root, 151
 ROUND, 83
 ROUNDDOWN, 88
 ROUNDUP, 88
 RRI, 604
 RSQ, 306
 schur, 492
 scorergi, 374
 scorerhi, 374
 SEC, 164
 sec, 164
 SECH, 178
 sech, 178
 SECOND, 567
 secondzeta, 459
 SERIESSUM, 100
 shanks, 523
 shi, 342
 si, 340
 siegeltheta, 449
 siegelz, 449
 SIGN, 95
 sign, 95
 SIN, 158
 sin, 158
 SINH, 172
 sinh, 172
 sim, 484
 SKEW, 290
 SKEW.P, 290
 SLN, 594
 SLOPE, 308
 SMALL, 294
 spherharm, 385
 SQRT, 149
 sqrt, 150
 sqrtm, 485
 SQRTPI, 157
 square, 145
 STANDARDIZE, 282
 STDEV, 288
 STDEV.P, 289
 STDEV.S, 288
 STDEVA, 288
 STDEV.P, 288
 STDEVPA, 289
 STEYX, 309
 stieljes, 446
 stirling1, 470
 stirling2, 470
 struveh, 364
 struvel, 365
 SUBTOTAL, 298
 SUM, 284
 SUMA, 284
 sumap, 515
 sumem, 514
 SUMIF, 284
 SUMIFS, 284
 SUMPRODUCT, 100
 SUMSQ, 100
 SUMX2MY2, 99
 SUMX2PY2, 99
 SUMXMY2, 99
 superfac, 321
 svd, 491
 SYD, 595
 T.DIST, 269
 T.DIST.2T, 270
 T.DIST.RT, 270
 T.INV, 271
 T.INV.2T, 271
 T.TEST, 302
 Table, 15
 TAN, 162
 tan, 162
 TANH, 176
 tanh, 176
 taufrom, 424
 taylor, 551
 TBILLEQ, 592
 TBILLPRICE, 592
 TBILLYIELD, 592
 TDIST, 269
 TDist, 269
 TDistInfo, 272
 TDistInv, 271
 TDistRan, 273
 TIME, 573
 TIMEVALUE, 573
 TINV, 271
 TODAY, 573
 TREND, 206
 TricomiGammaMpMath, 329
 TRIMMEAN, 282
 TRUNC, 86

- TTEST, 301
twinprime, 134
VAR, 286
VAR.P, 287
VAR.S, 287
VARA, 287
VARP, 287
VARPA, 287
VDB, 596
webere, 366
WEEKDAY, 569
WEEKNUM, 570
WEEKNUM-ADD, 570
WEIBULL, 274
WEIBULL.DIST, 274
WeibullDist, 274
WeibullDistInfo, 275
WeibullDistInv, 275
WeibullDistRandom, 276
whitm, 402
whitw, 402
WORKDAY, 574
WORKDAY.INTL, 574
XIRR, 606
XNPV, 606
YEAR, 569
YEARFRAC, 578
YIELD, 590
YIELDDISC, 591
YIELDMAT, 591
Z.TEST, 301
ZernikeRadialMpMath, 390
zeta, 441
zetazero, 447
ZTEST, 301
- Spreadsheet Functions
- ABS, 93, 636
ACCRINT, 583, 649
ACCRINTM, 584, 649
ACOS, 186, 636
ACOSH, 194, 636
ACOT, 191, 636
ACOTH, 196, 636
AGGREGATE, 298, 645
AMORDEGRC, 597, 649
AMORLINC, 596, 649
ARABIC, 77, 636
ASIN, 184, 636
ASINH, 193, 636
ATAN, 188, 636
ATAN2, 190, 636
ATANH, 195, 636
AVEDEV, 289, 641
AVERAGE, 285, 641
AVERAGEA, 285, 641
AVERAGEIF, 285, 641
AVERAGEIFS, 285, 641
BASE, 80, 636
BESSELI, 198, 639
BESSELJ, 198, 639
BESSELK, 199, 639
BESSELY, 198, 639
BETA.DIST, 223, 645
BETA.INV, 225
BETA.INV, 645
BETADIST, 223, 641
BETAINV, 224, 641
BIN2DEC, 77, 639
BIN2HEX, 77, 639
BIN2OCT, 77, 639
BINOM.DIST, 228, 645
BINOM.DIST.RANGE, 229, 645
BINOM.INV, 230, 645
BINOMDIST, 228, 641
BITAND, 110, 639
BITLSHIFT, 106, 639
BITOR, 110, 639
BITRSHIFT, 108, 639
BITXOR, 110, 639
CEILING, 83, 636
CEILING.MATH, 84, 636
CEILING.PRECISE, 83, 645
CHIDIST, 232, 641
CHIINV, 234, 641
CHISQ.DIST, 232, 645
CHISQ.DIST.2T, 233
CHISQ.DIST.RT, 233, 645
CHISQ.INV, 234, 645
CHISQ.INV.RT, 235, 645
CHISQ.TEST, 303, 645
CHISQDIST, 232, 644
CHISQINV, 235, 644
CHITEST, 303, 641
COMBIN, 202, 636
COMBINA, 202, 636
COMPLEX, 91, 639
CONFIDENCE, 300, 641

CONFIDENCE.NORM, 300, 645
 CONFIDENCE.T, 300
 CONFIDENCE.T, 645
 CONVERT, 280, 639
 CORREL, 305, 641
 COS, 160, 636
 COSH, 174, 636
 COT, 168, 636
 COTH, 182, 636
 COUNT, 277, 641
 COUNTA, 277, 641
 COUNTBLANK, 277, 641
 COUNTIF, 278, 641
 COUNTIFS, 278, 641
 COUPDAYBS, 579, 649
 COUPDAYS, 580, 649
 COUPDAYSNC, 580, 649
 COUPNCD, 580, 649
 COUPNUM, 581, 649
 COUPPCD, 581, 649
 COVAR, 305, 641
 COVARIANCE.P, 305, 645
 COVARIANCE.S, 305, 645
 CRITBINOM, 229, 641
 CSC, 166, 636
 CSCH, 180, 636
 CUMIPMT, 601, 649
 CUMIPMT_ADD, 649
 CUMPRINC, 602, 649
 CUMPRINC_ADD, 649
 DATE, 571, 652
 DATEVALUE, 571, 652
 DAVERAGE, 311, 648
 DAY, 568, 652
 DAYS, 569, 652
 DAYS360, 576, 652
 DB, 595, 649
 DCOUNT, 310, 648
 DCOUNTA, 311, 648
 DDB, 594, 649
 DEC2BIN, 78, 639
 DEC2HEX, 78, 639
 DEC2OCT, 78, 639
 DECIMAL, 81, 636
 DEGREES, 156, 636
 DELTA, 113, 639
 DEVSQ, 289, 642
 DGET, 310, 648
 DISC, 584, 649
 DMAX, 313, 648
 DMIN, 313, 648
 DOLLARDE, 608, 649
 DOLLARFR, 608, 649
 DPRODUCT, 310, 648
 DSTDEV, 312, 648
 DSTDEVP, 312, 648
 DSUM, 311, 648
 DURATION, 581, 650
 DVAR, 312, 648
 DVARP, 312, 648
 EASTERSUNDAY, 571, 652
 EDATE, 572, 652
 EFFECT, 602
 EFFECT_ADD, 650
 EOMONTH, 572, 652
 ERF, 199, 639
 ERF.PRECISE, 199, 645
 ERFC, 199, 639
 ERFC.PRECISE, 199, 645
 EVEN, 86, 636
 EXP, 136, 636
 EXPON.DIST, 238, 645
 EXPONDIST, 238, 642
 F.DIST, 241, 645
 F.DIST.RT, 241, 646
 F.INV, 243, 646
 F.INV.RT, 243, 646
 F.TEST, 302, 646
 FACT, 202, 637
 FACTDOUBLE, 202, 637
 FDIST, 241, 642
 FINV, 243, 642
 FISHER, 306, 642
 FISHERINV, 307, 642
 FLOOR, 84, 637
 FLOOR.MATH, 85, 637
 FLOOR.PRECISE, 84, 646
 FORECAST, 308, 642
 FREQUENCY, 278, 642
 FTTEST, 302, 642
 FV, 598, 650
 FVSCHEDULE, 603, 650
 GAMMA, 200, 639
 GAMMA.DIST, 247, 646
 GAMMA.INV, 248, 646
 GAMMADIST, 247, 642
 GAMMAINV, 248, 642
 GAMMALN, 200, 642

GAMMALN.PRECISE, 200, 646
GAUSS, 262, 643
GCD, 203, 637
GCD_ADD, 637
GEOMEAN, 286, 642
GESTEP, 113, 639
GROWTH, 207, 642
HARMEAN, 286, 642
HEX2BIN, 79, 639
HEX2DEC, 79, 639
HEX2OCT, 79, 639
Histogram, 279
HOUR, 568, 652
HYPGEOM.DIST, 252, 646
HYPGEOMDIST, 251, 642
IMABS, 93, 639
IMAGINARY, 92, 639
IMARGUMENT, 94, 639
IMCONJUGATE, 95, 639
IMCOS, 160, 639
IMCOSH, 174, 639
IMCOT, 168, 639
IMCOTH, 182
IMCSC, 166, 639
IMCSCH, 180, 640
IMDIV, 107, 640
IMEXP, 136, 640
IMLN, 141, 640
IMLOG10, 143, 640
IMLOG2, 144, 640
IMPOWER, 146, 640
IMPRODUCT, 105, 640
IMREAL, 92, 640
IMSEC, 164, 640
IMSECH, 178, 640
IMSIN, 158, 640
IMSINH, 172, 640
IMSQRT, 149, 640
IMSUB, 101, 640
IMSUM, 98, 640
IMTAN, 162, 640
IMTANH, 176
INT, 86, 637
INTERCEPT, 308, 642
INTRATE, 585, 650
IPMT, 601, 650
IRR, 604, 650
ISEVEN, 117
ISNUMBER, 115
ISODD, 117
ISOWEEKNUM, 570, 652
ISPMT, 603, 650
KURT, 291, 642
LARGE, 293, 642
LCM, 204, 637
LCM_ADD, 637
LINEST, 206, 642
LN, 141, 637
LOG, 142, 637
LOG10, 143, 637
LOGEST, 207, 642
LOGINV, 256, 642
LOGNORM.DIST, 255, 646
LOGNORM.INV, 256
LOGNORM.INV, 646
LOGNORMDIST, 254, 642
MAX, 292, 642
MAXA, 292, 642
MDETERM, 205, 637
MDURATION, 582, 650
MEDIAN, 292, 642
MIN, 292, 642
MINA, 292, 642
MINUTE, 568, 652
MINVERSE, 205, 637
MIRR, 605, 650
MMULT, 128, 637
MOD, 109, 637
MODE, 293, 642
MODE.MULT, 293, 646
MODE.SNGL, 293, 646
MONTH, 569, 652
MROUND, 88, 637
MULTINOMIAL, 203, 637
MUNIT, 123, 637
NEGBINOM.DIST, 258, 646
NEGBINOMDIST, 258, 642
NETWORKDAYS, 576, 652
NETWORKDAYS.INTL, 577, 646
NOMINAL, 602, 650
NOMINAL_ADD, 650
NORM.DIST, 261
NORM.DIST, 646
NORM.INV, 264, 646
NORM.S.DIST, 262, 646
NORM.S.INV, 264, 646
NORMDIST, 261, 642
NORMINV, 263, 642

NORMSDIST, 262, 642
 NORMSINV, 264, 642
 NOW, 573, 652
 NPER, 599, 650
 NPV, 605, 650
 OCT2BIN, 80, 640
 OCT2DEC, 80, 640
 OCT2HEX, 80, 640
 ODD, 86, 637
 ODDFPRICE, 585, 650
 ODDFYIELD, 586, 650
 ODDLPRICE, 587, 650
 ODDLYIELD, 587, 650
 PDURATION, 600, 650
 PEARSON, 306, 642
 PERCENTILE, 294, 642
 PERCENTILE.EXC, 294, 646
 PERCENTILE.INC, 294, 646
 PERCENTRANK, 295, 642
 PERCENTRANK.EXC, 295, 646
 PERCENTRANK.INC, 295, 647
 PERMUT, 203, 643
 PERMUTATIONA, 203, 643
 PHI, 262, 643
 PI, 133, 637
 PMT, 599, 650
 POISSON, 266, 643
 POISSON.DIST, 266, 647
 POWER, 146, 637
 PPMT, 601, 650
 PRICE, 588, 650
 PRICEDISC, 589, 650
 PRICEMAT, 589, 651
 PROB, 297, 643
 PRODUCT, 104, 637
 PV, 598, 651
 QUARTILE, 295, 643
 QUARTILE.EXC, 296
 QUARTILE.EXC, 647
 QUARTILE.INC, 296, 647
 QUOTIENT, 89, 637
 RADIANS, 156, 637
 RAND, 120, 637
 RANDBETWEEN, 120, 637
 RANK, 296, 643
 RANK.AVG, 297, 647
 RANK.EQ, 297, 647
 RATE, 600, 651
 RECEIVED, 590, 651
 ROMAN, 77, 637
 ROUND, 83, 637
 ROUNDDOWN, 88, 637
 ROUNDUP, 88, 637
 RRI, 604, 651
 RSQ, 306, 643
 SEC, 164, 637
 SECH, 178, 637
 SECOND, 567, 652
 SERIESSUM, 100, 637
 SIGN, 95, 637
 SIN, 158, 637
 SINH, 172, 637
 SKEW, 290, 643
 SKEW.P, 290, 647
 SLN, 594, 651
 SLOPE, 308, 643
 SMALL, 294, 643
 SQRT, 149, 637
 SQRTPI, 157, 637
 STANDARDIZE, 282, 643
 STDEV, 288, 643
 STDEV.P, 289, 647
 STDEV.S, 288, 647
 STDEVA, 288, 643
 STDEVP, 288, 643
 STDEVPA, 289, 643
 STEYX, 309, 643
 SUBTOTAL, 298, 637
 SUM, 284, 637
 SUMA, 284
 SUMIF, 284, 638
 SUMIFS, 284, 638
 SUMPRODUCT, 100, 638
 SUMSQ, 100, 638
 SUMX2MY2, 99, 638
 SUMX2PY2, 99, 638
 SUMXMY2, 99, 638
 SYD, 595, 651
 T.DIST, 269, 647
 T.DIST.2T, 270, 647
 T.DIST.RT, 269, 647
 T.INV, 271, 647
 T.INV.2T, 271, 647
 T.TEST, 302, 647
 TAN, 162, 638
 TANH, 176, 638
 TBILLEQ, 592, 651
 TBILLPRICE, 592, 651

TBILLYIELD, 592, 651
TDIST, 269, 643
TIME, 573, 652
TIMEVALUE, 573, 652
TINV, 271, 643
TODAY, 573, 652
TREND, 206, 643
TRIMMEAN, 282, 643
TRUNC, 86, 638
TTEST, 301, 643
VAR, 286, 643
VAR.P, 287, 647
VAR.S, 286, 647
VARA, 287, 643
VARP, 287, 643
VARPA, 287, 643
VDB, 596, 651
WEEKDAY, 569, 652
WEEKNUM, 570, 652
WEEKNUM-ADD, 570
WEIBULL, 274, 643
WEIBULL.DIST, 274, 647
WORKDAY, 574, 652
WORKDAY.INTL, 574, 647
XIRR, 606, 651
XNPV, 606, 651
YEAR, 569, 652
YEARFRAC, 578, 652
YIELD, 590, 651
YIELDDISC, 591, 651
YIELDMAT, 591, 651
Z.TEST, 301, 647
ZTEST, 301, 643