

mle

A Programming Language for Building Likelihood Models

Version 2.1

Volume 2. Reference Manual

Darryl J. Holman

mle

A Programming Language for Building Likelihood Models

Version 2.1

Volume 2. Reference Manual

© Copyright 1991–2003

Darryl J. Holman

Department of Anthropology
Center for Studies in Demography and Ecology
Center for Statistics and the Social Sciences
The University of Washington
Box 353100
Seattle, WA 98195

djholman@u.washington.edu

The software and manual for *mle* version 2 is distributed in electronic form free of charge for personal and academic use. Permission to use, copy, and distribute this software and documentation is hereby granted for personal and non-commercial academic use provided that the above copyright notice appears on all copies and that both the copyright notice and this permission notice appear in the supporting documentation. Other uses of this manual or software are prohibited unless the author grants written permission. This software may not be sold or repackaged for sale in whole or in part without permission of the author.

This software is provided "as is", without warranty. In no event shall the Author be liable for any damages, including but not limited to special, consequential or other damages. The Author specifically disclaims all other warranties, expressed or implied, including but not limited to the determination of suitability of this product for a specific purpose, use, or application. The user is responsible for ensuring the accuracy of any results. Sound engineering, scientific, and statistical judgment is the user's responsibility.

Suggested citation: Holman, Darryl J. (2003) *mle: A Programming Language for Building Likelihood Models*. Version 2.1. Vol. 2: Reference Manual.
<http://faculty.washington.edu/~djholman/mle>.

mle List: There is an email list for *mle* users to receive update and bug notices. To subscribe, send an email message to majordomo@pop.psu.edu with the text "subscribe mle" as the body of the email message.

Preface

This second volume of the *mle* manual systematically documents procedures, functions, predefined variables, predefined probability density functions and other features of the language. This part of the manual is largely devoid of explanatory material that can be found in the User's manual, yet incorporates details missing from the quick reference card.

Brief table of contents

PREFACE	II
BRIEF TABLE OF CONTENTS	III
TABLE OF CONTENTS	IV
INTRODUCTION.....	1
EXPRESSIONS.....	9
NUMBER FORMATS	21
PREDEFINED VARIABLES AND CONSTANTS.....	27
SIMPLE FUNCTIONS	35
SPECIAL FUNCTIONS	85
STATEMENTS.....	111
PROCEDURES.....	135
CONTINUOUS PROBABILITY DENSITY FUNCTIONS.....	146
DISCRETE PROBABILITY DENSITY FUNCTIONS.....	273
MATHEMATICAL SYMBOLS AND FUNCTIONS	289
ERROR AND WARNING MESSAGES	297
REFERENCES	307

Table of contents

PREFACE	II
BRIEF TABLE OF CONTENTS	III
TABLE OF CONTENTS	IV
INTRODUCTION.....	1
A PROGRAM	1
STATEMENTS.....	1
EXPRESSIONS.....	1
CATEGORIZED STATEMENTS, FUNCTIONS, PROCEDURES AND PREDEFINED VARIABLES	2
<i>Variable, function and procedure definitions.....</i>	2
<i>Flow control and boolean functions.....</i>	3
<i>Logical (bitwise) functions.....</i>	3
<i>File and data set operations.....</i>	3
<i>Input and output operations.....</i>	4
<i>String conversion</i>	4
<i>Date and time operations.....</i>	4
<i>General math functions.....</i>	4
<i>Trigonometric functions.....</i>	5
<i>Coordinate system conversions.....</i>	5
<i>Probability and Statistics.....</i>	5
<i>Graphing.....</i>	6
EXPRESSIONS.....	9
INTRODUCTION.....	9
<i>Identifiers.....</i>	9
<i>Simple functions.....</i>	9
<i>Special functions</i>	10
<i>Algebraic operators.....</i>	10
<i>Expressions.....</i>	12
OPERATOR REFERENCE	13
OPERATOR PRECEDENCE	19
NUMBER FORMATS.....	21
NOTATION.....	21
NUMBER FORMAT REFERENCE	21
PREDEFINED VARIABLES AND CONSTANTS.....	27
REFERENCE	27
SIMPLE FUNCTIONS.....	35
SIMPLE FUNCTIONS REFERENCE	35
SPECIAL FUNCTIONS.....	85
SPECIAL FUNCTIONS REFERENCE.....	85
<i>DATA.....</i>	85

Table of contents

<i>DERIVATIVE</i>	86
<i>FINDMIN</i>	87
<i>FINDZERO</i>	88
<i>IF ... THEN</i>	89
<i>INTEGRATE</i>	90
<i>LEVEL</i>	93
<i>LEVELDELTA</i>	94
<i>PARAM</i>	95
<i>PDF</i>	99
<i>PHAZARD</i>	102
<i>PPDF</i>	103
<i>PREASSIGN and POSTASSIGN</i>	104
<i>PRODUCT</i>	105
<i>QUANTILE</i>	106
<i>QDF</i>	107
<i>SUMMATION</i>	108
STATEMENTS	111
STATEMENTS	111
<i>Assignment statement</i>	111
<i>BEGIN statement</i>	113
<i>BREAK statement</i>	113
<i>CONTINUE statement</i>	114
<i>CURVE statement</i>	114
<i>DATA statement</i>	119
<i>EXIT statement</i>	121
<i>FOR statement</i>	121
<i>FUNCTION declaration (user defined)</i>	123
<i>IF statement</i>	125
<i>INCLUDE statement</i>	125
<i>MODEL statement</i>	127
<i>PLOT statement</i>	130
<i>PROCEDURE declaration</i>	131
<i>Procedure call</i>	132
<i>REPEAT statement</i>	133
<i>WHILE statement</i>	133
PROCEDURES	135
BUILT-IN PROCEDURES	135
CONTINUOUS PROBABILITY DENSITY FUNCTIONS	146
NOTATION AND FORMAT	146
ALPHA	148
ARCSINE	150
ASYMPTOTICRANGE	152
BETA	154
BIRNBAUMSAUNDERS	157
BIVNORMAL	159
CAUCHY	160
CHI	162
CHISQUARED	164
COMPOUNDEXTREME	166
DANIELS	168
DISK	170
EXPONENTIAL	172
FOLDEDNORMAL	174
GAMMA	176
GAMMAFRAIL	179
GENGAMMA	181

Table of contents

GENGUMBEL	183
GOMPERTZ.....	185
HORSESHOE.....	188
HYPERBOLICSECANT	190
HYPER2EXP.....	192
HYPO2EXP	194
INVBETA1	196
INVBETA2	198
INVCHI.....	200
INVGAMMA	202
INVGAUSSIAN	204
LAPLACE.....	206
LARGEEXTREME1 AND GUMBEL.....	208
LARGEEXTREME2.....	210
LINEARHAZARD	212
LNGAMMA	214
LNLOGISTIC.....	216
LOGISTIC.....	218
LOGNORMAL or LNNORMAL.....	220
LOWMAX.....	222
MAKEHAM.....	224
MAXWELL.....	226
MIXMAKEHAM.....	228
NORMAL or GAUSSIAN	230
PARETO	232
POWERFUNCTION.....	234
RAISEDCOSINE.....	236
RANDOMWALK	238
RAYLEIGH.....	240
REVPOWERFUNCTION	242
RINGINGEXP0	245
RINGINGEXP180.....	247
SHIFTEXPONENTIAL	249
SHIFTGAMMA.....	251
SHIFTLOGNORMAL	253
SHIFTWEIBULL	255
SILER	257
SMALLEXTREME1	259
SMALLEXTREME2	261
SUBBOTIN	263
UNIFORM or RECTANGULAR (CONTINUOUS)	265
VONMISES.....	267
WEIBULL	269
DISCRETE PROBABILITY DENSITY FUNCTIONS.....	273
NOTATION AND FORMAT	273
BERNOULLITRIAL	274
BINOMIAL	275
GEOMETRIC.....	276
HYPERGEOMETRIC	277
LOGSERIES.....	278
NEGBINOMIAL	279
NEGHYPERGEOMETRIC.....	280
PASCAL	281
POISSON	282
POLYAEGGENBERGER.....	283
STERILE or IMMUNE	284
THOMAS	285
ZIPF	286

Table of contents

MATHEMATICAL SYMBOLS AND FUNCTIONS	289
SYMBOLS	289
CONSTANTS.....	290
FUNCTION DEFINITIONS	290
THE GREEK ALPHABET	292
METRIC PREFIXES.....	292
INTERNATIONAL ELECTROTECHNICAL COMMISSION PREFIXES FOR BINARY MULTIPLES.....	292
SELECTED SYSTÈME INTERNATIONAL D'UNITÉS	293
ANGLES.....	294
TIME	294
TEMPERATURE CONVERSIONS.....	294
ERROR AND WARNING MESSAGES	297
MESSAGES FROM COMMAND LINE OPTIONS.....	297
WARNING MESSAGES	298
RUN-TIME ERRORS	299
ERRORS FROM THE PARSER	301
ERROR MESSAGES FROM DATA ROUTINES	303
ERROR MESSAGES FROM FUNCTION CALLS:	304
ERROR MESSAGES FROM SYMBOL TABLE ROUTINES	304
REFERENCES	307

Chapter 1

Introduction

This chapter provides an overview of the *mle* programming language. Elements of the language are given according to category. Information on using *mle* can be found in the User's Manual.

A Program

The *mle* language is composed of a series of statements enclosed within `MLE` and `END`.

`MLE <statements> END`

mle is an interpreted language. Unlike many interpreted languages that read, parse, and execute one statement at a time, *mle* reads the entire `MLE ... END` sequence. Following the `END`, and assuming no errors have been encountered, the statements will be executed. Additional `MLE ... END` programs can follow.

Statements

Statements direct the *mle* interpreter to do something. Statements define and assign values to variables, control program flow, cause the program to repeatedly execute a set of statements (i.e. loop through statements), etc.

Each statement can be thought of as a stand-alone instruction. Even so, a statement may contain other statements within it.

An important type of statement is a procedure call. For example, the `WRITELN(...)` procedure call is a statement onto itself. Yet, throughout this manual, procedures will be described separately from other statements. This somewhat artificial division makes it easier to document. All statements are covered in one chapter, and another chapter describes all procedures.

Expressions

An expression is anything that, after being “evaluated” returns a value. In an assignment statement, for example, the right-hand side is an expression. It is easiest to think of expressions as something that can appear on the right-hand side of an assignment statement.

An expression can be as simple as a number (2, 3.14, -9) or a boolean constant (TRUE, FALSE). Just slightly up on the complexity scale is a variable or predefined constant name (PI, oo, myvariable).

A more complex expression is a call to one of the many built-in functions or a call to a user-defined function. Finally, expressions can be composed of algebraic expressions composed of algebraic operators (+, -, *, ^, etc.).

This manual contains a chapter that documents expressions. Two separate chapters document functions. One chapter describes all simple functions. These functions have the form `xxxx()` or `xxxx`, where `xxxx` is the function name, and the parentheses enclose an argument list. Simple functions always have a fixed number of arguments that are passed to them, and they return a value.

Another chapter describes special functions. Special functions have a variety of different forms and a non-uniform syntax. For example the `INTEGRATE` function can have the form `INTEGRATE <var> (<expr> <expr>) <expr> END`. Variants of the integrate function have more arguments than those shown here.

Finally, two entire chapters will be devoted to documenting the different probability density functions built into *mle*. The special functions `PDF`, `QUALTILE`, `PPDF`, and `QPDF` return values for a specified density function. *mle* “knows about” many type of probability distributions, and reference chapters document these distributions. One chapter documents continuous probability distributions and another chapter documents discrete probability distributions.

Categorized statements, functions, procedures and predefined variables

In this section, statements, functions, procedures and predefined variables are listed in functional categories. Details are omitted, however. Some items are listed in multiple categories. I have not listed all predefined variables, as some just don’t warrant a mention.

Variable, function and procedure definitions

Statements

<code><var> = <expr></code>	{simple assignment}
<code><var>:<type></code>	{simple declaration}
<code><var>:<type> = <expr></code>	{simple declaration with initialization}
<code><var>:<type>[i TO j]</code>	{array declaration}
<code><var>:<type>[i TO j] = <expr></code>	{array declaration with initialization}
<code><var>:<type>[i TO j, ...]</code>	{multidimensional array declaration}
<code><var>:<type>[i TO j, ...] = <expr></code>	{ ... and with initialization}
<code><var>:<type>[i TO j] = [<expr> [<expr>...]]</code>	{ ... with data initialization}
<code><var>:<type>[i TO j, ...] = [[<expr> [<expr>...]], [...]]</code>	
<code>PROCEDURE <name> ... END</code>	{user-defined procedure}
<code>PROCEDURE <name>([var]<variable>:<type>...) ... END</code>	
<code>FUNCTION <name>:<type> ... END</code>	{user-defined function}
<code>FUNCTION <name>([var]<var>:<type>...)<type> ... END</code>	

Procedures

DUMPSYMBOL(x)

DUMPTABLE

Functions

ARGCOUNT
 ARGSTRING(x)
 ENVCOUNT

ENVSTRING
 GETENV(x)
 EULERSC

GRAVITATIONALC
 ISANIINFINITY
 ISINFINITY

ISNAN

ISNEGINFINITY

Predefined variables and constants

ATOMICMASSU
 AVOGADROSN
 BOHRMAGNETON
 BOHRRADIUS
 BOLTZMANNSC
 EPSILON
 INFINITY
 LARGE_ZERO
 LIGHTC
 MACHINE_EPSILON
 MAXINT
 MAX_BOOLEANS

MAX_CHARS
 MAX_INTEGERS
 MAX_REALS
 MAX_STRINGS
 NAN
 NEGINFINITY
 oo
 PARSE_ONLY
 PLANCKINV2PI
 PLANCKSC
 PROGRAM_NAME
 RELEASE

REVISION
 RYDBERG
 SQRT_EPSILON
 SYMBOLICINFIN
 SYSTEM
 TITLE
 UNIVERSALGASC
 VERBOSE
 VERSION
 DEBUG
 DEBUG_PARSE
 DEBUG_SYM

Flow control and boolean functions*Statements*

BEGIN <statements> END
 BREAK
 CONTINUE
 EXIT
 FOR <var> = <expr> TO <expr> DO <statements> END
 FOR <var> = <expr> TO <expr> STEP <expr> DO <statements> END
 FOR <var> = <expr> TO <expr> STEPS <iexpr> DO <statements> END
 FOR <var> = <array> DO <statements> END
 IF <bexpr> THEN <statements>
 [ELSEIF <bexpr> THEN <statements>...]
 [ELSE <statements>]
 END
 <Procedure call>
 REPEAT <statements> UNTIL <bexpr>
 WHILE <bexpr> DO <statements> END
 HALT

Special functions

IF <bexpr> THEN <expr> [ELSEIF <bexpr> THEN <expr>...] ELSE <expr> END
 POSTASSIGN <expr> <statement> END
 PREASSIGN <statement> <expr> END

Simple functions

ANDF(x, y)	ISINFINITY(x)	ISNEGINFINITY(x)
ISANINFINITY(x)	ISLE(x, y)	ISODD(x)
ISEQ(x, y)	ISLT(x, y)	NOTF(x)
ISEVEN(x)	ISNAN(x)	ORF(x, y)
ISGE(x, y)	ISNE(x, y)	XORF(x, y)
ISGT(x, y)	ISNEAR(x, b, δ)	

Predefined variables and constants

D_IDX
 FALSE

LINE_NUMB
 TRUE

Logical (bitwise) functions*Simple functions*

ANDF(x, y)	ORF(x, y)	SHIFTRIGHT(x, y)
NOTF(x)	SHIFTLEFT(x, y)	XORF(x, y)

File and data set operations*Statements*

DATA
 <var> [FIELD i [LINE j]] [= <expr>] [DROPIF <expr> | KEEPIF <expr> ...]
 ...
 END

Procedures

CHDIR(name)	EXEC(s, s)	OPENWRITE(f, x2)
CLOSE(f)	FLUSH(f)	OUTFILE(<namestr>)
DATAFILE(name)	MKDIR(name)	PLOTFILE(<namestr>)
DATATOARRAY(x1, x2,...)	OPENAPPEND(f, name)	RENAME(name1, name2)
ERASE(name)	OPENREAD(f, name)	RMDIR(name)

Simple functions

DEFALULTOUTNAME	EXEC(s, s)	FILESIZE(x)
DEFAULTPLOTNAME	EXISTS(s)	GETDIR

Special functions

```
DATA [FORM = SUMLL | SUM[MATION] | PROD[UCT]] <expr> END
LEVEL <bexpr> THEN [FORM = SUMLL | SUM[MATION] | PROD[UCT]] <expr> END
LEVELDELTA <expr> THEN [FORM = SUMLL | SUM[MATION] | PROD[UCT]] <expr> END
```

Predefined variables and constants

CREATE_OBS	FOUTFILE	N_OBS
D_IDX	FOUTPUT	N_VARS
DATADELIMITERS	FPLOT	OUTFILENAME
DATAFILE	FPLOTDATA	TOTAL_OBS
DEBUG_DATA	INPUT_SKIP	
DROPPED_OBS	LINES_PER_OBS	
FDATA	MLEFILEBASE	

Input and output operations*Procedures*

PRINT(x1, x2,...)	READLN([f,] x1, x2,...)	WRITEPLOT(x1, x2,...)
PRINTLN(x1, x2,...)	WRITE([f,] x1, x2,...)	WRITEPLOTLN(x1, x2,...)
READ([f,] x1, x2,...)	Writeln([f,] x1, x2,...)	

Simple functions

EOF(f)	EOLN(f)	PUT(x)
--------	---------	--------

Predefined variables and constants

COMPLEXDECIMALS	PRINT_COUNTS	REALDECIMALS
COMPLEXWIDTH	PRINT_DATA_STATS	REALWIDTH
DATADELIMITERS	PRINT_FIELDS	SYMBOLICINFIN
DATAFILENAME	PRINT_OBS	WRITEDELIMITER
OUTFILENAME	READDELIMITERS	

String conversion*Simple functions*

BOOL2STR(x)	REAL2STR(x, l, s)	TOBASE(x)
CHR(x)	RIGHTSTRING(x, y)	TOLOWER(x)
CONCAT(x1, x2)	STRING2INT(s)	TOUPPER(x)
INT2STR(x)	STRING2REAL(s)	TRIM(x)
LEFTSTRING(x, y)	STRINGLEN(x)	TRIML(x)
ORD(c)	SUBSTRING(x, y, z)	TRIMR(x)

Date and time operations*Procedures*

GETDATE(y [m [d]])	GETTIME(h [m [s [s100]]])
--------------------	---------------------------

Simple functions

DMYTOJ(x, y, z)	JULIANY(x)	MONTHDAYS(m, y)
JULIAND(x)	LEAPYEAR(y)	WEEKDAY(x)
JULIANM(x)	LUNARPHASE(j)	YEARDAY(x)

Predefined variables and constants

SECONDSPERDAY

General math functions*Procedures*

DEC(x)	INC(x)
--------	--------

Special functions

DERIVATIVE [(<i><expr></i>)] <i><var></i> = <i><expr></i> [<i><expr></i> [<i><expr></i>]] END	{numerical derivative}
FINDMIN <i><var></i> (<i><expr></i> <i><expr></i> [<i><expr></i> [<i><expr></i> [<i><expr></i>]]]) <i><expr></i> END	{find minimum of function}
FINDZERO <i><var></i> (<i><expr></i> <i><expr></i> [<i><expr></i> [<i><expr></i> [<i><expr></i>]]]) <i><expr></i> END	{find zero of function}
INTEGRATE <i><var></i> (<i><expr></i> <i><expr></i> [<i><expr></i>]) <i><expr></i> END	{numerical integration}
PRODUCT <i><var></i> (<i><expr></i> <i><expr></i> [<i><expr></i>]) <i><expr></i> END	{Product of a series}
SUMMATION <i><var></i> (<i><expr></i> <i><expr></i> [<i><expr></i>]) <i><expr></i> END	{Sum of a series}

Simple functions

ABS(x)	IDIV(x, y)	MULTIPLY(x, y)
ADD(x, y)	IM(x)	NEGATE(x)
ARG(x)	INC(x)	POWER(x, y)
CEIL(x)	INT(x)	RE(x)
COMP(x)	INVERT(x)	REMAINDER(x, y)
COMPEN(x, n)	LCM(x, y)	ROOT(x, y)
DEC(x)	LN(x)	ROUND(x)
DELTA(x, y)	LOG(x)	SGN(x)
DIVIDE(x, y)	LOG10(x)	SIGN(x, y)
EXP(x)	LOGBASE(x, y)	SQR(x)
FLOOR(x)	MAX(x, y)	SQRT(x)
FRAC(x)	MIN(x, y)	SUBTRACT(x, y)
GCF(x, y)	MIX(p, x, y)	TRUNC(x)
HEAVISIDE(x)	MODULO(x, y)	

Predefined variables and constants

E	I_TRAP_CLOSED	LNINFINITY
FIND_EPS	I_TRAP_OPEN	LOG_10
FIND_MAXITER	INTEGRATE_METHOD	PI
I_AQUAD	INTEGRATE_N	
I_SIMPSON	INTEGRATE_TOL	

Trigonometric functions*Simple functions*

ARCCOS(x)	ARCSIN(x)	CSCH(x)
ARCCOSH(x)	ARCSINH(x)	SEC(x)
ARCCOT(x)	ARCTAN(x)	SECH(x)
ARCCOTH(x)	ARCTANH(x)	SIN(x)
ARCCSC(x)	COS(x)	SINH(x)
ARCCSCH(x)	COSH(x)	TAN(x)
ARCSEC(x)	COT(x)	TANH(x)
ARCSECH(x)	COTH(x)	

Predefined variables and constants

PI	DEGREESPERRADIAN	RADIANSPERDEGREE
----	------------------	------------------

Coordinate system conversions*Simple functions*

DMSTOD(x, y, z)	RECTTOSPHEREA1(x, y, z)
DMSTOR(x, y, z)	RECTTOSPHEREA2(x, y, z)
DTOR(x)	RECTTOSPHERER(x, y, z)
EARTHDIST(x1 x2 x3 x4)	RTOD(x)
POLARTORECTX(r, a)	SPHERETORECTX(r, a1, a2)
POLARTORECTY(r, a)	SPHERETORECTY(r, a1, a2)
RECTTOPOLARA(x, y)	SPHERETORECTZ(r, a1, a2)
RECTTOPOLARR(x, y)	

Predefined variables and constants

DEGREESPERRADIAN	RADIANSPERDEGREE
------------------	------------------

Probability and Statistics*Statements*

```

MODEL
  <expr>                                     {the likelihood}
RUN [ THEN <statements> END ]
FULL [ THEN <statements> END ]
  | REDUCE <paramname>=<expr> [<paramname>=<expr>...] [ THEN <statements> END ]
  | WITH <paramname> ...[ THEN <statements> END ]
  | WITH [<paramname> | (<paramname>...) ...] ... [ THEN <statements> END ]

```

```
...
END
```

Procedures

```
PTRANSFORM(<var1>, <var2>, <expr>)
SEED(x)
```

Special functions

```
PARAM <var> [HIGH=<expr>].[LOW=<expr>] [START=<expr>] [TEST=<expr>]
          [FORM=<paramform>]
          [COVAR <expr> <expr> . . . ]
END
QUANTILE <PDF name> (<expr> [<expr> [<expr>]])
          <expr> <expr> ...
          [HAZARD COVAR <var> <expr> [COVAR <var> <expr>...]]
END
QDF <PDF name>(<q_expr> [<expr>, <expr>]) <expr> <expr>... END
QPDF <PDF name>(<q_expr> [<expr>, <expr>]) <expr> <expr>... END
PDF <PDF name> (<expr> <expr> ... )
          <expr> <expr> ...
          [HAZARD COVAR <var> <expr> [COVAR <var> <expr>...]]
END
```

Simple functions

BETA(v, ω)	GAMMA(x)	LNGAMMA(x)
BESSELI(x, y)	IBETA(p, v, ω)	LOGISTIC(x)
BESSELJ(x, y)	IBETAC((p, v, ω)	LOGIT(x)
BESSELK(x, y)	IGAMMA(x, y)	PERMUTATIONS(x, y)
BESSELY(x, y)	IGAMMAC(x1, x2)	RAND
COMB(x, y)	IGAMMAE(x1, x2)	RRAND(x, y)
ERF(x)	INVCHISQ(p, d)	SETRANSFORM(expr)
ERFC(x)	INVNORMAL(p)	STANDARDIZE(x, μ , σ)
FACT(x)	IRAND(x, y)	ZETA(x)
FISHER(x)	LNFACT(x)	
FISHERINV(x)		

Predefined variables and constants

AIC_SELECT	EVALS	PRINT_SE
AICC_SELECT	EXP_HAZARD	PRINT_SHORT
ALT_LOGISTIC	FREE_PARAMS	PRINT_VCV
ANNEALING	HIGH_DEFAULT	RANDOM_SEED
BIC_SELECT	IC_SAMPLE_SIZE	READSTARTFILE
BRENT_ITS	INFO_METHOD1	SA_ADJ_CYCLES
BRENT_MAGIC	INFO_METHOD2	SA_ADJ_LOWERBOUND
CGRADIENT1	ITERATION_PRINT	SA_ALT_ADJUSTMENT
CGRADIENT2	ITERATIONS	SA_COOLING
CI_CHISQ	LARGEST_LIKELIHOOD	SA_EPS_NUMBER
CI_CONVERGE	LARGEST_LLIKELIHOOD	SA_STEPLength
CI_EVALS	LOGLIKELIHOOD	SA_STEPLength_ADJ
CI_LIMIT_DELTA	LOW_DEFAULT	SA_STEPS
CI_MAXITS	MAKEVALS	SA_TEMPERATURE
CONVERGENCE	MAXITER	SIMPLEX
DEBUG_INT	METHOD	SIMPLEX_ALPHA
DEBUG_LIK	METHOD_LOOP	SIMPLEX_BETA
DELTA_LL	MIN_SIGNIFICANT	SIMPLEX_GAMMA
DIFF_DX	MINIMUM_ITS	SMALLEST_LIKELIHOOD
DIRECT	NEWTON	SMALLEST_LLIKELIHOOD
DIST_DX_SCALE	NOTSINGULAR	SMALLEST_NUMBER
DIST_T_END	POWELL	START_DEFAULT
DIST_T_N	PRINT_BASIC	SURFACE_POINTS
DIST_T_START	PRINT_CI	TEST_DEFAULT
DX_MAXITS	PRINT_DISTs	VCV_EVALS
DX_START	PRINT_FREE_PARAMS	VCV_WIDTH
DX_TOOBIG	PRINT_INFO	WRITESTARTFILE
DX_TOOSMALL	PRINT_LLIKS	

Graphing

Statements

```
CURVE [KEY <string> | WITH <string> | AXES <string>...]
      <ivar> = <iexpr> TO <iexpr> | <rvar> (<expr>, <expr> [<iexpr>])
```

mle Reference manual: Introduction

```
<expr> <expr> [<expr>...] [<string>...]  
END  
MULTIPLY <statements> END  
PLOT <statements> END
```

Procedures

```
FINISHPLOT(x)                               WRITEPLOT(x1, x2,...)  
PLOTFILE(<name>)                           WRITEPLOTLN(x1, x2,...)
```

Simple functions

```
DEFAULTPLOTNAME
```

Predefined variables and constants

FLOT	INTERMLOT	PLOT_DIST
FLOTDATA	MLOTXSCALE	PLOTFILENAME
GNUPLOT	MLOTYSCALE	PLOTINIT
GNUPLOTINIT	MULTIPLYINIT	PLOTPOINTS

Chapter 2

Expressions

This chapter serves as a reference for basic expressions and operators in *mle*. The chapter begins with an introduction to the elements used in expressions: Identifiers, functions and operators. A complete

Introduction

This chapter describes expressions in the *mle* programming language. The general purpose of an expression is to "return" some value. Expressions are used for calculations of all types. In *mle*, expressions result in a value that is type REAL, COMPLEX, INTEGER, STRING, CHAR, or BOOLEAN.

Identifiers

A very simple type of expression is a constant or a variable. New variables can be defined in a program. Additionally, *mle* predefines a number of built-in variables and constants. Some predefined variables allow you to change and fine-tune the behavior of *mle*. A later chapter discusses many of those variables. Some constants arise frequently in numerical work, and are predefined for convenience.

Simple functions

Several classes of functions are supported by *mle*. The first is simple functions, like SIN(), EXP(), SUBTRACT(), RAND, etc. These are called simple functions because their format includes a keyword (e.g. SIN) and a list of zero or more arguments. If the function takes arguments, they are enclosed in a list of parenthesis. For example, MIX(p, part1, part2). Here are some examples of simple functions used in assignment statements:

x = FACT(4)	{evaluates to 24}
y = SQRT(44.5)	{evaluates to 6.6708...}
z = BETA(1.2, 9*3/10 + 1)	{evaluates to 0.185...}
q = RAND	{evaluates to a random number from 0 to 1}

The built-in functions are described in detail in a later chapter.

Special functions

Another class of functions has a more complicated syntax. These functions begin with a keyword and end with an `END`. Compound functions require a more complicated set of arguments or require a variable number of parameters. For example, the `IF ... THEN ... ELSEIF ... THEN ... ELSE ... END` function can take on an unlimited number of `ELSEIF...THEN` clauses. Likewise, the `PDF` function requires that some predefined probability density function be named, followed by a parenthesis-enclosed list of "time" arguments, followed by a list of zero or more intrinsic parameters, sometimes followed by a `HAZARD` clause before the closing `END`.

A reference for all compound functions is given later in this chapter.

Algebraic operators

Expressions can be constructed from algebraic operators like `+`, `/`, `XOR`. Algebraic operators take one or two operands, perform some operation, and return a result. The result can be a number, boolean, string or logical value. Algebraic operators differ from functions in the way they take arguments. They are used in an algebraic format like $(a + b/c)^2$ instead of having arguments specified in a list such as a call to `min(x, b)`.

Algebraic expressions are expressions created using a series of special operators. The operators include algebraic symbols like `+`, `-`, `*`, `/`, `^`, a series of algebraic keywords for integer operations, `DIV`, `MOD`, `SHL`, `SHR`. A list of all operators is given below.

There is a simple function that corresponds to each of the algebraic operators listed below. For example, the expression $1 + \pi * r^2$ is translated by *mle* to a series of simple function calls: `ADD(1, MULTIPLY(PI, POWER(r, 2)))`.

Variable Types

There are seven different *types* supported by *mle*: `REAL`, `INTEGER`, `COMPLEX`, `BOOLEAN`, `STRING`, `CHAR` (character), and `FILE`.

A variable's *type* refers to the domain of values that the variable can take on. For example, `INTEGER` variables can take on a limited range of integer values, `BOOLEAN` variables can only take on the values `TRUE` and `FALSE`. Variables can be defined for each of the seven types and expressions always take on one of these types. Here is an explanation of each:

- **Real** variables represent the continuous real number line. Many mathematical functions like `SIN()`, `EXP()`, and `BESSELI()` return real values, and so the variable to which these functions are assigned must be type `REAL` as well. Integer variables and functions can always be assigned to real variables—they are automatically converted to real values on assignment. Integer variables, on the other hand, must use the `ROUND()` or `TRUNC()` functions to convert a real number to an integer value.

- **Integer** variables take on whole number values over a machine-dependent range of numbers. For most versions of *mle* this range is [-2,147,483,648 to 2,147,483,647]. Arguments to some functions *require* INTEGER type variables, like IDIV().
- **Complex** variables include a real number part and an imaginary part. Complex numbers are specified by expressions such as `1.2 + 0.4i`. Most mathematical functions are defined for complex types. For example, `SQRT(-1 + 0i)` returns `0.000+1.000i`. There is no natural ordering for complex variables, so that the comparisons `<`, `<=`, `>`, and `>=` are undefined.
- **Boolean** variables take on one of two states: TRUE or FALSE. No other value is allowed or recognized. Boolean expressions are frequently used to test conditions. For example, the `IF...THEN...ELSE...END` function evaluates the first expression (between the `IF` and `THEN`) to either TRUE or FALSE and decides which of the remaining two expressions will be evaluated and returned.
- **String** variables hold a sequence of character constants. A string written as a constant is a sequence of characters, enclosed within quotes ("). The single quote character (') can be used as well for strings greater than one character. String variables are typically used to assign file names, titles, etc. Some functions take on string (or character) variables, other functions return strings. For example, the `CONCAT(s1, s2)` function will add together two string variables and return it as a longer string.
- **Character** variables take on the value of a single character. When written as a constant in a program, character constants consist of a single character enclosed within single quotes ('). Character constants are not typically used within a user's program, but are available if needed. Usually, character constants and variables can be used anywhere string variables are allowed.
- **File** variables are used to reference files. Most of the time, file variables are transparent, and you need not explicitly define or manipulate file variables. This is because *mle* defines and does the bookkeeping for the data file, the output file, the plot file, and the screen (or standard output) file. File variables can be created should you wish to create and manipulate other files.

When a variable is first used in an assignment statement, its *type* will be determined by the *type* returned from the expression on the right-hand side. Here are some examples to illustrate the point:

<code>large_data = N_OBS > 5000</code>	{large_data will be type BOOLEAN}
<code>subtitle = "Analysis: " + DEFAULTTOUTNAME</code>	{subtitle will be type STRING}
<code>nine = 3 * 3.0</code>	{nine will be REAL}
<code>five = 2 + 3</code>	{five will be INTEGER}

You can explicitly define the *type* for a variable when it is first referenced in an assignment statement. Here are some examples:

```

c:STRING = 'x'      {c would default to CHAR, but instead will be a STRING variable}
nine:REAL = 3 * 3    {nine would default to INTEGER, but will be a REAL variable}
t:BOOLEAN = TRUE     {t is explicitly declared as Boolean; this is the default}
ang:REAL = SIN(2*pi) {ang is explicitly declared as real; this is the default}

```

Expressions

Algebraic Expressions

Algebraic expressions have many uses in *mle*. For example, the right hand side of an assignment statement takes an expression. The likelihood specified in the `MODEL` statement is an expression. The right-hand sides of the following assignment statements are valid expressions:

```

n = 2*3
n = (HOURS/60)^2
n = 12.5*first - 10*second
i = mask SHL 4
i = 23 DIV 4

```

Boolean Expressions

Boolean expressions evaluate to either `TRUE` or `FALSE`. The operators for creating boolean expressions are `>`, `<`, `>=`, `<=`, `==`, `=`, `<>`, and boolean keywords, `AND`, `OR`, `XOR`, and `NOT` and some simple functions. These operators are used in the same way as they are in many other programming languages. Notice that the `==` operator can be used in place of `=` for boolean expressions. The right-hand sides of the following assignment statements are examples of boolean expressions:

```

b = FALSE
b = a <> 42^2
b = (a <> 12) AND (a >= 0)

```

Logical Expressions

The difference between boolean and logical expressions is that boolean expressions work with the values `TRUE` and `FALSE` only, whereas logical expressions work with bits. For example, `NOT TRUE` is equal to `FALSE`; but `NOT 767` is equal to `-768`. How does this work? The number 767 is represented by the computer as the binary sequence 0000000000000000000000001011111111. The logical `NOT` operator flips all 1s to 0s and 0s to 1s, so that the number becomes 1111111111111111111111110100000000. The first (left most) bit denotes a negative value, so the value is `-768`. The logical `AND`, `OR`, and `XOR` functions act bit-by-bit, as well. Thus, the binary values 2X101101 AND 2X111000 (which is the same as 45 AND 56) evaluates to 40 (or 2X101000). The `SHL` and `SHR` operators shift bits to the left and right. So, 2X000111 SHL 3 (i.e. 7 SHL 3) evaluates to 56 (or 2X111000).

You might wonder how *mle* decides whether an operator is boolean or logical. The answer is simple: if both operands are boolean types, the operator will be boolean. If both operands are integers, the operator will be logical. If one operator is boolean and one is logical, an error will result. For the expression `(x >= 4) OR (y <= 2)`, each of the expressions in parenthesis will evaluate to `TRUE` or `FALSE`, so that the `OR` will be a boolean operator.

Operator reference

- (*uniary negation*)

Returns:	-x
Constraints:	x is an integer, real or complex expression.
Examples:	-2 returns -2 --2 returns 2 -3.0 returns -3.0000 -(3+0i) returns -3.0000+0.00000i
Range:	Returns a real value for real x, an integer value for integer x, and a complex value for a complex arguments.
See also:	function NEGATE, +

+ (*uniary positive*)

Returns:	x
Constraints:	x is an integer, real or complex expression.
Examples:	+2 returns 2 +-2 returns -2
Range:	Returns a real value for real x, an integer value for integer x, and a complex value for a complex arguments.
See also:	function NEGATE, -

^ (*power function*)

Returns:	x^y , that is x raised to the y power.
Constraints:	x and y are integer, real or complex expressions. If $x < 0$ then y must have no fractional part ($\text{FRAC}(y)$ must be 0)
Examples:	2.0^3 returns 8.0 2.0^-3 returns 0.125 (-2.0)^3 returns -8.0 (-2.0)^-3 returns -0.125 2.0^0.5 returns 1.414213562 (-2.0)^0.5 returns an error (2+3i)^2 returns -5.0+12.0i 2^0 returns 1.0 0^0 returns 1.0
Notes:	Exponentiation of large or small numeric values can cause floating point overflow or underflow. Such overflows are not currently handled in an elegant way.

Range:	Returns a real value for real and integer x and a complex value for a complex arguments.
See also:	functions POWER, ROOT, SQR, SQRT
<i>* (multiplication)</i>	
Returns:	$x*y$, x multiply by y.
Constraints:	x and y are integer, real or complex expressions.
Examples:	<p>2.0*3.0 returns 6.0</p> <p>2.0*-3 returns -6.0</p> <p>-2.0*3 returns -6.0</p> <p>-2.0*-3 returns 6.0</p>
Range:	Returns a real value for real x, an integer value for integer x, and a complex value for a complex arguments.
Notes:	Multiplication of large real or complex values can cause floating point overflow; multiplication of very small real or complex values can cause floating point underflow. Multiplication of very large integer values can cause integer overflow. Such overflows are not currently handled in an elegant way.
See also:	functions MULTIPLY, DIVIDE; operators /, *
<i>/ (division)</i>	
Returns:	x/y , x divided by y.
Constraints:	$y \neq 0$. x and y are integer, real or complex expressions.
Examples:	<p>4.0/2.0 returns 2.0</p> <p>2.0/-4.0 returns -0.5</p>
Range:	Returns a real value for real or integer x and a complex value for a complex arguments.
Notes:	Division of large or small real or complex values can cause floating point overflow or underflow. Such overflows are not currently handled in an elegant way.
See also:	functions DIVIDE, MULTIPLY, IDIV, MODULO; operators DIV, MOD, *
<i>DIV</i>	
Returns:	The integer value of x/y .
Constraints:	$y \neq 0$. x and y are integer expressions.
Examples:	<p>4 DIV 2 returns 2</p> <p>2 DIV 4 returns 0</p>
Range:	Returns an integer value.
See also:	functions DIVIDE, IDIV, MODULO; operators /, MOD

MOD

Returns: The integer value of x modulo y . The `MOD` operator returns $x1 - y * \text{FLOOR}(x/y)$ ($y \neq 0$, otherwise 0).

Constraints $y \neq 0$. x and y must be integers

Examples: `4 MOD 2` returns 0
`2 MOD 4` returns 0

Range: Returns an integer value.

See also: functions `DIVIDE`, `IDIV`, `MODULO`, `REMAINDER`; operators `/`, `DIV`

AND

Returns: The boolean or logical AND function.

Constraints For a boolean AND, both operands must be type boolean. For the logical AND, both operands must be type integer

Examples: `TRUE AND FALSE` returns `FALSE`
`143 AND 327` returns 67

Range: Returns a boolean for boolean operands, returns an integer for integer operands value.

See also: function `ANDF`; operators `OR`, `XOR`

SHL

Returns: Logical (bitwise) shift to the left; `SHIFTLEFT(x, y)` Bits are shifted away from the right (least significant) to the left (most significant). Bits can be shifted off of the left end of the integer.

Constraints Operands must be type integer

Examples: `2 SHL 10` returns 2048
`2X100 SHL 2` returns 16
`MAXINT SHL 1` returns -2.

Range: Returns an integer value.

See also: function `SHIFTLEFT`, `SHIFTRIGHT`; operators `SHR`

SHR

Returns: Logical (bitwise) shift to the right; `SHIFTRIGHT(x, y)` Bits are shifted away from the left (most significant) to the right (least significant). Bits can be shifted off of the right end of the integer.

Constraints Operands must be type integer

Examples: `2048 SHR 10` returns 2
`2 SHR 2` returns 0

Range: Returns an integer value.

See also: function `SHIFTRIGHT`, `SHIFTLEFT`; operators `SHL`

+ (addition)

Returns: The sum of x and y . `ADD(x, y)`

Constraints:	x and y are integer, real, complex, or string expressions.
Examples:	<p>4.0 + 2.0 returns 6.0</p> <p>2 + -4 returns -2</p> <p>(3+2i) + (4+3i) returns 7.0 + 5.0i</p> <p>"Car" + "pet" returns the string "Carpet"</p>
Range:	For numeric operands, returns an integer if both operands are integer. Returns a complex if any operand is complex, otherwise returns a real if any operand is a real. Returns a string for string operands. Numeric and string operands cannot be combined.
Notes:	Addition of large numbers can cause overflows. Such overflows are not currently handled in an elegant way.
See also:	functions ADD, CONCAT, SUBTRACT; operators -
- (<i>subtraction</i>)	
Returns:	The difference x minus y. SUBTRACT(x, y)
Constraints:	x and y are integer, real, or complex expressions.
Examples:	<p>4.0 - 2.0 returns 2.0</p> <p>2 - -4 returns 2</p> <p>(3+2i) - (4+3i) returns -1.0 - 1.0i</p>
Range:	Returns an integer if both operands are integer. Returns a complex if any operand is complex, otherwise returns a real if any operand is a real.
Notes:	Subtraction can cause overflows. Such overflows are not currently handled in an elegant way.
See also:	functions SUBTRACT, ADD; operators +
OR	
Returns:	The boolean or logical OR function; ORF(x, y)
Constraints	For a boolean OR, both operands must be type boolean. For the logical OR, both operands must be type integer
Examples:	<p>TRUE OR FALSE returns TRUE</p> <p>143 OR 247 returns 255</p>
Range:	Returns a boolean for boolean operands, returns an integer for integer operands value.
See also:	function ORF; operators XOR, AND
XOR	
Returns:	The boolean or logical exclusive OR function, XORF(x, y).
Constraints	For a boolean XOR, both operands must be type boolean. For the logical XOR, both operands must be type integer
Examples:	TRUE XOR FALSE returns TRUE

	TRUE XOR TRUE returns FALSE
	143 XOR 247 returns 120
Range:	Returns a boolean for boolean operands, returns an integer for integer operands value.
See also:	function XORF; operators OR, AND
<code>==</code> or <i>(is equal)</i>	
Returns:	A boolean indicating the equality of the operands; the ISEQ(x, y) function.
Constraints	Operands can be complex, real, integer, boolean or string. Numeric types can be mixed, and will be converted to the highest type.
Examples:	<p>3 == 2 returns FALSE</p> <p>3 == 3 returns TRUE</p> <p>3 == 3.0 returns TRUE</p> <p>3 == (3.0 + 0i) returns TRUE</p> <p>'a' == 'a' returns TRUE</p> <p>'a' == 'ab' returns FALSE</p> <p>'a' == 'A' returns FALSE</p>
Range:	Returns a boolean.
See also:	function ISEQ; operators <>, <= >=, <, >
<code><></code> <i>(is not equal)</i>	
Returns:	A boolean indicating nonequality of the operands; the ISNE(x, y) function.
Constraints	Operands can be complex, real, integer, boolean or string. Numeric types can be mixed, and will be converted to the highest type.
Examples:	<p>3 <> 2 returns TRUE</p> <p>3 <> 3 returns FALSE</p> <p>3 <> 3.0 returns FALSE</p> <p>3 <> (3.0 + 0i) returns FALSE</p> <p>'a' <> 'a' returns FALSE</p> <p>'a' <> 'ab' returns TRUE</p> <p>'a' <> 'A' returns TRUE</p>
Range:	Returns a boolean.
See also:	function ISNE, ISEQ; operators =, <= >=, <, >
<code><</code> <i>(is less than)</i>	
Returns:	A boolean indicating equality of the operands; the ISLT(x, y) function.

Constraints	Operands can be real, integer, boolean or string. Numeric types can be mixed, and will be converted to the highest type. Note that this function is undefined for complex numbers.
Examples:	<p>3 < 2 returns FALSE</p> <p>3 < 3 returns FALSE</p> <p>2 < 3 returns TRUE</p> <p>2 < 3.0 returns TRUE</p> <p>'a' < 'b' returns TRUE</p> <p>'a' < 'A' returns FALSE</p>
Range:	Returns a boolean.
See also:	function ISLT, ISLE, ISGT, ISGE; operators <=, >, >=, =, <>
> (is greater than)	
Returns:	A boolean indicating equality of the operands; the ISGT(x, y) function.
Constraints	Operands can be real, integer, boolean or string. Numeric types can be mixed, and will be converted to the highest type. Note that this function is undefined for complex numbers.
Examples:	<p>3 > 2 returns TRUE</p> <p>3 > 3 returns FALSE</p> <p>2 > 3 returns FALSE</p> <p>3.0 > 2 returns TRUE</p> <p>'b' > 'a' returns TRUE</p> <p>'A' > 'a' returns FALSE</p>
Range:	Returns a boolean.
See also:	function ISGT, ISLT, ISLE, ISGE; operators <=, <, >=, =, <>
<= (is less than or equal to)	
Returns:	A boolean indicating equality of the operands; the ISLE(x, y) function.
Constraints	Operands can be real, integer, boolean or string. Numeric types can be mixed, and will be converted to the highest type. Note that this function is undefined for complex numbers.
Examples:	<p>3 <= 2 returns FALSE</p> <p>3 <= 3 returns TRUE</p> <p>2 <= 3 returns TRUE</p> <p>2 <= 3.0 returns TRUE</p> <p>'a' <= 'b' returns TRUE</p> <p>'a' <= 'A' returns FALSE</p>

Range:	Returns a boolean.
See also:	function ISLE, ISLT, ISGT, ISGE; operators <, >, >=, =, <>
<i>>= (is greater than or equal to)</i>	
Returns:	A boolean indicating equality of the operands; the ISGE(x, y) function.
Constraints	Operands can be real, integer, boolean or string. Numeric types can be mixed, and will be converted to the highest type. Note that this function is undefined for complex numbers.
Examples:	3 >= 2 returns TRUE 3 >= 3 returns TRUE 2 >= 3 returns FALSE 3.0 >= 2 returns TRUE 'b' >= 'a' returns TRUE 'A' >= 'a' returns FALSE
Range:	Returns a boolean.
See also:	function ISGE, ISGT, ISLT, ISLE; operators >, <=, <, =, <>

Operator precedence

An important consideration when using algebraic operators is the relative precedence with which operations are performed. For example, the expression `10 + 20 DIV 5` could reasonably give the result 6 (if operations are performed left to right) or 14 (if `DIV` has precedence over `+`). *mle*, like most programming languages, defines a precedence among algebraic operators that corresponds to ordinary mathematical usage. Table 1 shows the operator precedence for *mle*. Higher precedence operators will always be evaluated before lower precedence operators. Thus, the expression `-2^2` returns -4, not 4 (because exponentiation has precedence over negation). Of course, parentheses can be used to override operator precedence, so that `(-2)^2` returns 4.

Table 1. Operator precedence.

Operator(s)	Precedence	Category
<code>^</code>	high	Exponent operator
<code>- + not</code>		Unary operators
<code>* / div mod and shl shr</code>		Multiplying operators
<code>+ - or xor</code>		Adding operators
<code>= (or ==) <> < > <= >=</code>	low	Relational operators

Chapter 3

Number formats

There are a remarkable number of ways one can express numbers. They take the form of dates, times, fractions, real numbers, integers, numbers in alternative bases, and they can take on a variety of suffices. *mle* tries to accommodate the most common types of number formats (and a few odd ones as well). This chapter describes formats for numbers permitted in *mle*.

Notation

The following notation is used for this section of the manual:

- *d* is a strings of one or more positive digits
- *s* is a one or two character case-sensitive metric or percent suffix
- *v* is a string of one or more Roman numeral digits {IVXLCDM}
- *y* is a string of one or more characters
- *mmm* is a 3-character English month name (jan, Feb, MAR, etc)
- The degree (°) and micro (μ) characters are available on some hardware platforms. On DOS/Windows platforms, these characters are ASCII codes 230 and 248 respectively. The characters are available by holding down the <ALT> key and type the code on the numeric keypad.

Number format reference

Integer

Format:	<i>d</i> , a string of integers
Constraints:	Integers on current versions are 32 bits, and range from -2147483647 to 2147483647. The constant <code>MAXINT</code> specifies the maximum integer possible.
Example:	200 corresponds to the integer 200.
Range:	Results in an integer

Real

Format:	<i>d.d, d.</i>
Constraints:	Real numbers on current versions are 64 bits, and range from -5.949×10^{4931} to 5.949×10^{4931} .
Example:	3.1415, 3.
Range:	Results in a real 64 bit real number.

Exponential format

Format:	<i>dEd, dE-d, d.dEd, d.dE-d, d.Ed, d.E-d.</i> <i>xEy</i> is taken to a real number as $x \times 10^y$.
Constraints:	Real numbers on current versions are 64 bits, and range from -5.949×10^{4931} to 5.949×10^{4931} .
Examples:	<p>3e23 returns 3.0E23</p> <p>511E-10 returns 5.11E-8</p> <p>31.416e-1 returns 3.1416</p> <p>7.0E-10 returns 7.1E-10</p> <p>12.e-6 returns 0.000012</p> <p>1.45E-3 returns 0.00145</p> <p>1.0E0 returns 1.0</p>
Range:	Results in a real 64 bit real number.

Suffices

Format:	<i>ds, -ds, d.ds, -d.ds</i> provides for numbers with modifying suffices. A suffix can be appended to any integer, real or exponential format number.
Constraints:	The constraints depend on the final type (integer or real)
Examples:	<p>14% returns 0.14</p> <p>23.7M returns 23700000.0</p> <p>45.7da returns 457.0</p> <p>2n returns 2.000E-0009</p> <p>2.418E returns 2.418E+0018</p>

Notes: The following table lists possible suffices.

Suffix	Name	Conversion	Suffix	Name	Conversion	Suffix	Name	Conversion
da	deka	$\times 10$	d	deci	$\times 10^{-1}$	Ki	kibi	$\times 2^{10}$
h	hecto	$\times 10^2$	c, %	centi, percent	$\times 10^{-2}$	Mi	mebi	$\times 2^{20}$
k	kilo	$\times 10^3$	m	milli	$\times 10^{-3}$	Gi	gibi	$\times 2^{30}$
M	mega	$\times 10^6$	μ , u	micro	$\times 10^{-6}$	Ti	tebi	$\times 2^{40}$
G	giga	$\times 10^9$	n	nano	$\times 10^{-9}$	Pi	pebi	$\times 2^{50}$
T	tera	$\times 10^{12}$	p	pico	$\times 10^{-12}$	Ei	exbi	$\times 2^{60}$
P	peta	$\times 10^{15}$	f	femto	$\times 10^{-15}$			
E	exa	$\times 10^{18}$	a	atto	$\times 10^{-18}$			
Z	zetta	$\times 10^{21}$	z	zepto	$\times 10^{-21}$			
Y	yotta	$\times 10^{24}$	y	yocto	$\times 10^{-24}$			

Roman format

Format: `0R ν`

Constraints: ν must be in the set of Roman digits: IVXLCDM

Examples: `0RXLVII` returns 47
`0rMXVI` returns 1016
`0rmdclxvi` returns 1666

Range: Returns a 32 bit integer

Base 2 to 36 format

Format: `dXy`. Specifies y in base d (from 2 to 36)

Constraints: d must be 2 to 36. y can contain any digit in the specified base from from 0-9, A-Z

Examples: `2x1001` (binary) returns 9
`8X3270` (octal) returns 1720
`16xA4CC` (hex) returns 42188
`32x3vq4h` (base 32) returns 4188305

Range: Returns a 32 bit integer

Fractions

Format: `d_d/d`

Constraints: The three numbers that make up the fraction must be integers

Examples: `12_5/16` returns 12.3125
`3_2/3` returns 3.6...
`0_1/7` returns 0.1428571428571
`-4_1/16` returns -4.0625

Range: Returns a real number

24 hour time format

Format:	<i>d:d.d, d:d:d.d, d:d, d:d.d</i> . Specifies time in a 24 hour format.
Constraints:	Hours must be an integer from 0 to 24, minutes can be integer or real (without seconds) in the range 0 to 60. Seconds can be real or integer in the range 0 to 60.
Examples:	10:42 returns 10.7 14:55:32 returns 14.925... 10:40:23.4 returns 10.67316... 16:53.2 returns 16.886...
Range:	Returns a real number

12 hour time format

Format:	<i>d:d:dAM, d:d:dPM, d:d:d.dAM, d:d:d.dPM, d:dPM, d:dAM, d:d.dAM, d:d.dPM</i> . Specifies a 12-hour time with AM and PM suffixes into hours.
Constraints:	Hours must be an integer from 0 to 12, minutes can be integer or real (without seconds) in the range 0 to 60. Seconds can be real or integer in the range 0 to 60.
Examples:	10:42AM returns 10.7 2:55:32pm returns 14.925... 10:40:23.4am returns 10.67316...
Range:	Returns a real number

Hour-minute-second format

Format:	<i>dHd'd", dHd'd.d", dHd', dHd.d", dHd.d"</i> . Specifies a number in degree (or hour), minute, second format, converted to radians.
Constraints:	Degrees/hours must be an integer, minutes can be integer or real (without seconds) in the range 0 to 60. Seconds can be real or integer in the range 0 to 60.
Examples:	230h16'32" returns 230.275... 14H32'6" returns 14.535 100h22' returns 100.36... 30H32.2' returns 30.536... 0h12' returns 0.2 0H12'3" returns 0.20083...
Range:	Returns a real number.

Degree-minute-second format

Format:	<i>d`d'd", d`d'd.d", d`d', d`d.d", -d`d'd", d`, d.d`, d°d'd", d°d'd.d", d°d', d°d.d", d°, d.d°, d'd", d'd.d", d', d.d', d", d.d"</i> . Degree-minute-second, minute-second, and second format, converted to radians
Examples:	230`16'32" returns 4.0190666313036

14`32'6" returns 0.2536836067774
100`22' returns 1.7517287925850
30`32.2' returns 0.5329653759173
14` returns 0.2443460952792
230°16'32" returns 4.0190666313036
14°32'6" returns 0.2536836067774
270°10'0" returns 4.7152978624713
30°18.2' returns 0.5288929409960
3.4° returns 0.0593411945678
12'32" returns 0.0036457988819
166'12.9" returns 0.0483499836034
19' returns 0.0055268759646
14.7' returns 0.0042760566674
12" returns 0.0000581776417
607.3" returns 0.0029442734854

Range: Returns a real number in the range 0 to 2π

Date formats

Formats: *dDdMdY*, *dMdDdY*, *dYdMdD*, *dmmmy*. Converts dates to a Julian day.

Constraints: Days must be a proper range for the month. Months are 1 to 12.

Examples: 16d12m1944y returns 2431441
1D6M1800Y returns 2378648
12m16d1944y returns 2431441
6M1D1800Y returns 2378648
1944y12m16d returns 2431441
1800Y6M1D returns 2378648
14Dec1999 returns 2451527
30jun1961 returns 2437481
1MAY1944 returns 2431212

Range: Returns an integer Julian day

Chapter 4

Predefined variables and constants

Many variables and constants are predefined in *mle*. Some of these constants are simply useful reference values (like `PI` and `AVOGADROSN`). Other values control aspects of the way parameters are found, how data sets are created, and the amount of output written to the output file or the screen. This chapter documents the names, initial values, and purpose of predefined variables and constants.

Reference

<code>AIC_SELECT</code>	<code>FALSE</code>	Reports Bayesian model averaged parameters based on Akaike's information criterion.
<code>AICC_SELECT</code>	<code>FALSE</code>	Reports Bayesian model averaged parameters based on sample-size corrected Akaike's information criterion.
<code>ALT_LOGISTIC</code>	<code>FALSE</code>	Controls how logistic transformation are done in function and parameter form <code>LOGISTIC</code> . If set to <code>false</code> , they return $p=1/[1+\exp(x)]$. If <code>true</code> , the transformation is $p=\exp(x)/[1+\exp(x)]$.
<code>ANNEALING</code>	<code>ANNEALING</code>	A string constant for the simulated annealing method.
<code>ATOMICMASSU</code>	<code>1.660 538 73E-27</code>	Atomic mass unit constant
<code>AVOGADROSN</code>	<code>6.022 141 99E-23</code>	Avogadro's number
<code>BATCHMODE</code>	<code>FALSE</code>	When <code>false</code> , the keyboard is monitored for keystrokes in order to invoke the interactive debugger. When <code>true</code> , the monitoring is disabled.
<code>BIC_SELECT</code>	<code>FALSE</code>	Reports Bayesian model averaged parameters based on Bayesian information criterion.
<code>BOHRMAGNETON</code>	<code>9.274 008 99E-24</code>	Bohr's magneton
<code>BOHRRADIUS</code>	<code>5.291 772 083E-11</code>	Bohr's radius
<code>BOLTZMANNSC</code>	<code>1.380 650 3×10⁻²³</code>	Boltzmann's constant.
<code>BRENT_ITS</code>	<code>200</code>	Defines the maximum number if iterations allowed for a single dimensional maximization when using the <code>BRENT</code> one-dimensional maximizer in the <code>DIRECT</code> and <code>CGRADIENT</code> methods.
<code>BRENT_MAGIC</code>	<code>0.381966</code>	$[3 - \sqrt{5}]/2$, a constant used by the one-dimensional maximizer
<code>CGRADIENT1</code>	<code>CGRADIENT1</code>	A <code>METHOD</code> string so that the conjugate gradient method (type 1) is used

CGRADIENT2	CGRADIENT2	A METHOD string so that the conjugate gradient method (type 2) is used
CI_CHISQ	5.02389	Chi square value for likelihood CIs. This value defines a 95% CI.
CI_CONVERGE	0.00005	The maximum allowable difference between the likelihood at the confidence limit and CHISQ_C
CI_EVALS	0	Used to record the number of function evaluations when computing confidence intervals.
CI_LIMIT_DELTA	0.0	When confidence intervals are computed, <i>mle</i> adds this value to the lower parameter value and subtracts this parameter from the upper parameter value, thus preventing the confidence interval from being evaluated at the boundary. Negative values will extend the range over which the CI will be evaluated.
CI_MAXITS	30	The maximum number of iterations allowed for finding a confidence limit.
COMPLEXDECIMALS	2	The number of decimal places in printing out complex numbers
COMPLEXWIDTH	7	The field width in printing out complex numbers
CONVERGE_FLAG	0	Records the conditions under which the model ended
CREATE_OBS	0	For positive values, creates that number of observations rather than reading them from a data file; otherwise, observations are read from a file.
D_IDX	1	The index into DATA. This variable can be used to loop through data.
DATADELIMITERS	<space><tab>,	A string of delimiters that separate items read by the DATA statement
DATAFILENAME		A string of delimiters that separate items read by the DATA statement
DEBUG_DATA	FALSE	When TRUE, debugging is turned on for the routines that read in data files (also -dd on the command line).
DEBUG_ECHO	FALSE	When TRUE, each statement in turn is written to the screen.
DEBUG_EXEC	FALSE	When TRUE, debug information is written for each statement.
DEBUG_INT	FALSE	When TRUE, turns on debugging for the integration routines (also -di on the command line).
DEBUG_LIK	FALSE	When TRUE, turns on debugging for likelihoods, and individual likelihoods are printed (also -dl on the command line).
DEBUG_PARSE	FALSE	When TRUE, turns on debugging for the language parsing routines (also -dp on the command line)
DEBUG_SYM	FALSE	When TRUE, turns on debugging for the symbol table routines (also -ds on the command line).
DEBUG	0	A integer debug level. The higher this value, the more trees that die. Values of 5 and 11 are useful. (also -d ## on the command line)
DEGREESPERRADIAN	57.2957795	The number of degrees in one radian = $\pi/180$
DELTA_LL	∞	Used to record the change in likelihood when solving models
DIFF_DX	0.001	The initial (largest) value of dx used for the DERIVATIVE function.
DIRECT	"DIRECT"	A METHOD string so that likelihoods are solved by the direct method
DIST_DX_SCALE	0.25	Multiplied by the standard error of a parameter when computing the derivative of distributions with respect to each parameter.
DIST_ERR_C	1	Multiplied by the standard error of a parameter when computing distributions with respect to each parameter. When set to 2, two standard errors are printed.
DIST_T_END	10.00	The default end time when distributions are printed. (See PRINT_DISTS and DIST_T_START)
DIST_T_N	10	The number of time points (between DIST_T_START and DIST_T_END) for which the distributions are printed. (See PRINT_DISTS).
DIST_T_START	1.00	The default start time when distributions are printed. (See PRINT_DISTS and DIST_T_END)

DROPPED_OBS	0	A count of observation lines dropped by the DATA statement
DX_MAXITS	12	The maximum number of iterations allowed for finding a reasonable sized Δx when computing numerical derivatives.
DX_START	0.1	The starting value for Δx used in finding numerical derivatives.
DX_TOOBIG	0.3	The maximum change in log likelihood allowed for computing a reasonable sized Δx for numerical derivatives.
DX_TOOSMALL	0.01	The minimum change in log likelihood allowed for computing a reasonable sized Δx for numerical derivatives.
E	2.71828...	The value e .
EPSILON	0.00001	The maximum change in log likelihood for convergence when estimating parameters for a model.
EULERSC	0.57721567	Euler's constant
EVALS	0	The number of function evaluations used to solve a model
EXP_HAZARD	True	Changes the way proportional hazards are modeled on the baseline hazard. if TRUE, $h(t) = h(t)'e^p$ otherwise $h(t) = h(t)p$.
FALSE	FALSE	A boolean constant
FDATA		File variable for data file
FIND_EPS	0.00001	The convergence criterion for the FINDZERO and FINDMIN functions
FIND_MAXITS	100	Maximum number of iterations for the FINDZERO and FINDMIN functions
FINPUT		Standard input file
FOUTFILE		Output file (used by procedure OUTFILE)
FOUTPUT		Standard output file, defaults to the screen
FPLOT		Plot file (used by procedure PLOTFILE)
FPLOTDATA		File variable used in creating plot data files
FREE_PARAMS		The number of free parameters while solving a model
GNUPLOT	gnuplot.exe	Name of the executable <i>GNUPLOT</i> program
GNUPLOTINIT		A string of <i>GNUPLOT</i> commands that is written to plot files. For Windows, this string is "set terminal windows; reset; set data style lines; set autoscale; set nokey"
GRAVITATIONALC	6.673E-11	Universal gravitational constant
HIGH_DEFAULT	oo	The default value for the HIGH parameter limit. If no HIGH = . . . is set for a parameter, the value will be taken from HIGH_DEFAULT.
I_AQUAD	3	Constant for adaptive quadrature integrator (default)
I_SIMPSON	0	Constant for Simpson integrator
I_TRAP_CLOSED	1	Constant for trapezoidal (closed endpoint) integrator
I_TRAP_OPEN	2	Constant for trapezoidal (open endpoint) integrator
IC_SAMPLE_SIZE	0.0	Sample size to use for AICC_SELECT and BIC_SELECT computations.
INFINITY	(hardware)	A constant for the largest real number available on the computer. This number is determined when <i>mle</i> begins.
INFO_METHOD1	True	Computes the local Fisher's information matrix using Nelson's (1983:394) first derivative method. This method does not work for nested likelihoods, but is fairly robust for non-nested models.
INFO_METHOD2	False	When TRUE, computes the local Fisher's information matrix using numerical perturbation for the second partial derivatives.

INPUT_SKIP	0	Number of initial rows to skip when reading in data files. This is useful when, for example, columns of numbers in the input file have one or more lines of headings to describe the columns.
INTEGRATE_METHOD	I_AQUAD	Sets the method of integration.
INTEGRATE_N	15	Number of points in the Simpson and trapezoidal integrators.
INTEGRATE_TOL	0.000001	The tolerance for the Simpson and adaptive quadrature integrators.
INTERMPLOT	""	A string with <i>GNUPLLOT</i> commands to insert between <i>MULTIPLOTS</i>
INVERT_FLAG	FALSE	Flags whether the a variance-covariance matrix has been inverted. False if last attempt was singular or never done.
ITERATION_PRINT	0	When set to <i>n</i> , every <i>n</i> th iteration will print out a partial result.
ITERATIONS	0	The number of iterations taken to solve a model.
LARGE_ZERO	(hardware)	A characteristic of the hardware floating point math.
LARGEST_LIKELIHOOD	(hardware)	The largest likelihood acceptable from a function
LARGEST_LLIKELIHOOD	(hardware)	The largest loglikelihood acceptable from a function
LIGHTC	299792458	The speed of light
LINE_NUMB	0	Set to each data file line while reading in data. Afterward, it is set to the number of lines in the data file.
LNINFINITY	(hardware)	The log of the largest representable real number
LOG_10	2.3025850	A constant.
LOGLIKELIHOOD	0.0	The loglikelihood found in solving the model
LOW_DEFAULT	-oo	The default value for the <i>LOW</i> parameter limit. If no <i>LOW</i> = . . . is set for a parameter, the value will be taken from <i>LOW_DEFAULT</i> .
MACHINE_EPSILON	(hardware)	Value associated with round-off error for the particular hardware being used. This number is determined when <i>mle</i> begins.
MAX_BOOLEANS	(hardware)	Fixed maximum size of a boolean array
MAX_CHARS	(hardware)	Fixed maximum size of a character array
MAX_INTEGERS	(hardware)	Fixed maximum size of an integer array
MAX_REALS	(hardware)	Fixed maximum size of a real array
MAX_STRINGS	(hardware)	Fixed maximum size of a string array
MAXEVALS	100000	The maximum number of function evaluations for solving a likelihood. Upon hitting <i>MAXEVALS</i> function evaluations, <i>mle</i> will terminate even if the convergence criterion has not been met.
MAXINT	(hardware)	Value of the greatest integer for the current architecture.
MAXITER	100	The maximum number of iterations allowed for estimating the parameters of a model. Upon hitting <i>MAXITER</i> iterations, <i>mle</i> will terminate even if the convergence criterion has not been met.
MAXTIME	0	Maximum time (seconds) for a model to run. Zero disables this convergence criterion.
METHOD_LOOP	FALSE	Turns on looping through methods until convergence is reached.
METHOD	DIRECT	<i>METHOD</i> takes on the value of one of several strings to define what method will be used for solving likelihoods.
MIN_SIGNIFICANT	4	The <i>minimum</i> number of significant digits to print for most fields in the parameter estimate reports.
MINIMUM_ITS	1	The minimum number of iterations when solving a model.
MLEFILEBASE		String with the base name of the current <i>mle</i> file.
MPLTXSCALE	1	Scales the horizontal size of plots for <i>MULTIPLLOT</i>

MPLOTYSCALE	1	Scales the vertical size of plots for MULTIPLOT
MULTIPLOTINIT	set size 1,1; set origin 0,0	String of Gnuplot commands that is written to plot files when plotting MULTIPLOTS
N_OBS		The number of observations read and kept from the input file
N_VARS		The number of variables read and kept from the input file
NAN		A constant set to the value "Not a Number" (NaN)
NEGINFINITY	(hardware)	The most negative real number supported by the machine. This value is determined when <i>mle</i> begins.
NEWTON	"NEWTON"	A METHOD used for finding parameter estimates.
oo	(hardware)	The greatest positive real number. Also INFINITY
OSSEP	(operating system)	The standard filename separator; '/' in Unix and '\' in Dos
OSVERSION	(operating system)	The version of the operating system.
OUTFILENAME	(operating system)	The name of the output file. Usually OUTFILE is defined in the <i>mle</i> program code using the procedure OUTFILE () . If OUTFILE is not defined in the program, the output will be sent to the standard output. OUTFILE can also be defined on the command line.
PARSE_ONLY	FALSE	When set to TRUE, <i>mle</i> parses the program file and then terminates. If syntax and other errors are encountered during parsing, <i>mle</i> will print the errors; otherwise, <i>mle</i> will simply terminate with error. The same effect is achieved by using -p on the <i>mle</i> command line.
PI	3.14159	The value π .
PLANCKINV2PI	1.054 571 596 E-34	Planck's constant divided by $2 \times \pi$.
PLANCKSC	6.626 087 6 E-34	Planck's constant
PLOT_DISTS	FALSE	Toggles plotting the survivorship, hazard, and PDF distributions at user-specified time points. When PLOT_DISTS is set to TRUE, values should also be set for DIST_T_START, DIST_T_END and DIST_T_N. See PLOT_DISTS.
PLOTFILENAME		The name of the plot file.
PLOTINIT	""	A string in <i>GNU PLOT</i> code for initializing a plot.
PLOTPOINTS	100	The default number of points in a plot.
POWELL	"POWELL"	A METHOD used for finding parameter estimates.
PRINT_BASIC	TRUE	Toggles printing of basic model information.
PRINT_CI	TRUE	Toggles printing of confidence interval report.
PRINT_COUNTS	TRUE	Toggles printing of variable counts from input variables.
PRINT_DATA_STATS	TRUE	Toggles printing of mean, standard deviation, minimum, maximum statistics for each input variable.
PRINT_DISTS	FALSE	Toggles printing of values for the survivorship, hazard, and PDF distributions at user-specified time points. When PRINT_DISTS is set to TRUE, values should also be set for DIST_T_START, DIST_T_END and DIST_T_N. The following code fragment will print the SDF, PDF, and hazard distributions at 100 time points from 100 to 300.

```
PRINT_DISTS = TRUE
DIST_T_START = 100.0
DIST_T_END = 300.0
DIST_T_N = 100
```

PRINT_FIELDS	FALSE	Toggles printing information about the field in the input file.
PRINT_FREE_PARAMS	FALSE	Toggles printing a list of free parameters sent to the maximizer. This is usually used for debugging purposes only.
PRINT_INFO	TRUE	Toggles printing of some information to the output file.

mle Reference manual: Predefined variables and constants

PRINT_LLIKS	TRUE	Toggles printing of individual likelihoods (1 per obs.) to the output file.
PRINT_OBS	FALSE	Toggles printing of the observations from the input file after transformations. When TRUE, prints the final values for all observations.
PRINT_PARAMS	FALSE	Toggles printing a report of parameters estimates (with no standard errors or confidence intervals).
PRINT_REDUCED	FALSE	Toggles printing of parameters reduced out of a model.
PRINT_SE	TRUE	Toggles printing of standard error report.
PRINT_SHORT	FALSE	When FALSE, <i>mle</i> prints a detailed reports for parameter estimates. When TRUE, <i>mle</i> prints a one-line report for the report. This option is useful when the results are to be manipulated directly by another program. The number of fields in the output report depend on how many parameters are estimated and whether the Standard Error report or the Confidence Limit report is generated.
PRINT_VCV	FALSE	Toggles printing of variance-covariance matrix. The rows and columns of the VCV matrix are in the same order as free parameters are defined.
PROGRAM_NAME	mle	
RADIANS PER DEGREE	0.01745329	The number of radians per degree = $\pi/180$
RANDOMSEED	-1	The initial random seed. This must be set to a positive number (use the SEED () procedure) before using the random number generator. Note that the simulated annealing maximizer must have a random seed set.
READDELIMITERS	<space><tab>,	Delimiters used to separate items in procedure READ and READLN
READSTARTFILE	FALSE	Toggles reading initial parameter values from a start file.
REALDECIMALS	5	The minimum number of digits written by WRITE, WRITELN, PRINT and PRINTLN procedures.
REALWIDTH	12	The width of numbers written by WRITE, WRITELN, PRINT and PRINTLN procedures.
RELEASE	—	The release number for <i>mle</i> .
REVISION	—	The revision number for <i>mle</i> .
RYDBERG C	1.097 373 156 854 9E7	Rydberg's constant
SA_ADJ_CYCLES	20	For simulated annealing, this is the number of step length adjustment steps every cooling cycle (iteration).
SA_ADJ_LOWERBOUND	0.4	For simulated annealing, picks the lower and upper percentages of accepted and rejected evaluations between which no step length adjustment is made.
SA_ALT_ADJUSTMENT	false	For simulated annealing, uses an alternative adjustment formula
SA_COOLING	0.85	This is the rate of cooling for each cooling cycle. $T_{n+1} = T_n \times SA_COOLING$. Values > 1 can be used to explore a good starting temperature.
SA_EPS_NUMBER	4	For simulated annealing, this is the number of function points that will be compared for determining convergence.
SA_STEPLength_ADJ	2.0	For simulated annealing, this is the steplength adjustment constant
SA_STEPLength	1.0	For simulated annealing, this is the steplength constant.
SA_STEPS	5	For the simulated annealing method, this is the number of steps of random parameter perturbations before entering an adjustment cycle.
SA_TEMPERATURE	1000.0	For the simulated annealing method, this is the initial temperature. This value is conservatively high for most functions.
SECONDS PER DAY	86400	The number of seconds in a day.
SIMPLEX_ALPHA	1.0	The simplex maximizer's reflection coefficient
SIMPLEX_BETA	0.5	The simplex maximizer's contraction coefficient
SIMPLEX_GAMMA	2.0	The simplex maximizer's extrapolation coefficient
SIMPLEX	"SIMPLEX"	A METHOD string to denote Nelder and Mead's (1965) simplex maximizer.

SMALLEST_LIKELIHOOD	(hardware)	The greatest value allowed for a likelihood
SMALLEST_LLIKELIHOOD	(hardware)	The greatest value allowed for a log likelihood
SMALLEST_NUMBER	(hardware)	The smallest positive number greater than zero supported by the hardware.
SQRT_EPSILON	(hardware)	A small number used for computing derivatives.
START_DEFAULT	0.5	The default <i>START</i> value for parameters used in the event no <i>START</i> = . . . is used in a parameter definition.
SURFACE_POINTS	20	Number of <i>x</i> and <i>y</i> points in a surface plot.
SYMBOLINFIN	TRUE	Toggles whether infinity is printed oo or numerically.
SYSTEM	—	Name of the operating system.
TERMFILE	FALSE	Toggles whether or not a termination file is used in solving models
TEST_DEFAULT	0.0	The default <i>t</i> -test value against which the parameters are tested. This value is used when the <i>TEST</i> = is not used in a parameter definition.
TITLE		A string that is printed for the <i>mle</i> main program and each model. When <i>TITLE</i> is defined before a <i>DATA</i> statement, the title will be printed to the output as the global title. The title can be redefined before each <i>MODEL</i> statement as a model-specific statement.
TOTAL_OBS	0	Total observations computed by summing <i>FREQUENCY</i> for each observation.
TRUE	TRUE	A boolean constant.
UNIVERSALGASC	8.314472	The universal gas constant.
VCV_EVALS	0	The number of function evaluations in computing the variance-covariance matrix.
VCV_WIDTH	5	The number of elements printed on one line for the variance-covariance matrix.
VERBOSE	FALSE	If true, <i>mle</i> prints out status information as it works. This is useful for following the progress of <i>mle</i> .
VERSION	—	The version number of <i>mle</i> .
WRITEDELIMITER	“”	A string with the default delimiter used by <i>WRITE</i> , <i>WRITELN</i> , <i>PRINT</i> and <i>PRINTLN</i> procedures.
WRITESTARTFILE	FALSE	Toggles whether start files are written while estimating models.

Chapter 5

Simple functions

This chapter describes the types and uses of simple functions available in *mle*. All simple functions are listed in alphabetical order. The descriptions are subdivided into a number of fields.

XXX(<i>x</i>)	This line describes the function name (XXX), and the list of arguments (<i>x</i>).
Returns:	Describes the purpose of the function.
Constraints:	Describes constraints on the argument(s) to the function. Constraints include mathematical limits, and constraints in the type of variable allowed (integer, real, complex, string, character, etc.).
Examples:	Gives examples of a call to the function and the resulting returned value.
Range:	Describes the type (integer, real, etc) and range of values returned by a function.
Notes:	Provides additional description the function or special cases of the function.
Reference:	Gives a reference to algorithm, definition, or other information about the function.
See also:	Gives a list other <i>mle</i> functions, procedures, and variables that are related to the function.

Simple Functions Reference

<i>ABS</i>(<i>x</i>)	
Returns:	Absolute value of <i>x</i> . The absolute value of complex value is defined as $\text{ABS}(a + bi) = \sqrt{a^2 + b^2}$, which is the Euclidean distance from the origin to the point to the image point in the complex plane.
Constraints:	<i>x</i> is integer, real or complex
Examples:	<code>ABS(-4)</code> returns 4

`ABS(4)` returns 4

`ABS(-4.0)` returns 4.0

`ABS(4 + 5i)` returns 6.4031242374

Range: Returns an integer for an integer value. Returns a real value for both real and complex arguments. The returned value is always ≥ 0 .

ADD(x , y)

Returns: The sum of two numbers, or the concatenation of strings or characters. This is the functional form of $x + y$.

Constraints: x and y numeric argumentss (integer, real or complex), or are both string (char) arguments.

Examples: `ADD(1, 5)` returns 6. The result is an integer.

`ADD(2.5, 2.5)` returns 5.0. The result is a real number.

`ADD(2, 2.5)` returns 4.5. The result is a real number.

`ADD('a', " string")` returns a string.

`ADD(2.5 + 3i, 4 - 1i)` returns 6.50000+2.00000i.

Range: Returns an integer if both arguments are integers. A real value is returned if at least one argument is real and the other is real or integer. A complex number is returned if either argument is complex. A string is returned if both arguments are either strings or characters.

See also: SUMMATION

ANDF(x , y)

Returns: Logical or boolean AND function. This is the functional from of x AND y .

Constraints: x and y are integers or both boolean values.

Examples: `ANDF(TRUE, TRUE)` returns TRUE

`ANDF(15, 28)` returns 12

`ANDF(2x010101, 2x000111)` returns 5

Range: If both x and y are integer types, `ANDF(x , y)` returns the bitwise (logical) AND of the two numbers. If x and y are boolean types, `ANDF(x , y)` returns the boolean AND of the two numbers.

See also: ORF, NOTF, XORF

ARCCOS(x)

Returns: Inverse cosine of x , which is the angle (in radians) whose cosine is x

Constraints: $-1 \leq x \leq 1$ for real arguments

Examples: `ARCCOS(0.5)` returns 1.0.

`ARCCOS(-1/2)` returns 2.0943951023932

`ARCCOS(1)` returns 0.000

`ARCCOS(0.5 + 2i)` returns 1.34978-1.46572i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: `COS`, `ARCCOSH`, `ARCSIN`

ARCCOSH(x)

Returns: Inverse hyperbolic cosine of x .

Constraints: $x \geq 1$ for real arguments

Examples: `ARCCOSH(2)` returns 1.3169578969248

`ARCCOSH(1.0)` returns 0.0

`ARCCOSH(1.0i)` returns 0.88137+1.57080i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: `COSH`, `ARCCOS`, `ARCSINH`

ARCCOT(x)

Returns: Inverse cotangent of x .

Constraints: $x \neq 0$

Examples: `ARCCOT(3)` returns 0.3217505543966

`ARCCOT(3 - 3i)` returns 0.16965+0.16348i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: `COT`, `ARCCOS`, `ARCSIN`

ARCCOTH(x)

Returns: Inverse hyperbolic cotangent of x .

Constraints: $x \neq 0$

Examples: `ARCCOTH(2)` returns 0.5493061443341

`ARCCOTH(2 - 2i)` returns 0.23888+0.25957i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: `COTH`, `ARCCOSH`, `ARCSINH`, `ARCCOT`

ARCCSC(x)

Returns: Inverse cosecant of x .

Constraints: $x \neq 0$

Examples: `ARCCSC(5)` returns 0.2013579207903

`ARCCSC(2 - 2i)` returns 0.24452+0.25490i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also:	CSC, ARCCOS, ARCSIN
<i>ARCCSCH</i> (x)	
Returns:	Inverse hyperbolic cosecant of x .
Constraints:	$x \neq 0$
Examples:	ARCCSCH(5) returns 0.1986901103 ARCCSCH(2 - 2i) returns 0.25490+0.24452i
Range:	Returns a real value from $-\infty$ to ∞ (but $\neq 0$) for integer or real arguments. Returns a complex value for complex arguments.
See also:	ARCCSC
<i>ARCSEC</i> (x)	
Returns:	Inverse secant of x .
Constraints:	$x \neq 0$
Examples:	ARCSEC(1) returns 0.0 ARCSEC(2 + 2i) returns -1.3263-0.25490i
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.
See also:	ARCCSCH, CSCH
<i>ARCSECH</i> (x)	
Returns:	Inverse hyperbolic secant of x .
Constraints:	$0 < x \leq 1$ for real arguments
Examples:	ARCSECH(0.5) returns 1.3169578969 ARCSECH(2 + 2i) returns 0.25490-1.32627i
Range:	Returns a real value between 0 to ∞ for integer or real arguments. Returns a complex value for complex arguments.
See also:	ARCCSC, SECH
<i>ARCSIN</i> (x)	
Returns:	Inverse sine of x , or the number whose angle (in radians) is x
Constraints:	$-1 \leq x \leq 1$ for real arguments
Examples:	ARCSIN(1) returns 1.5707963267949 ARCSIN(0.5) returns 0.5235987755983
Range:	Returns a real value for integer or real arguments from $-\pi/2$ to $\pi/2$. Returns a complex value for complex arguments.
See also:	ARCCOS, SIN, ARCSINH
<i>ARCSINH</i> (x)	
Returns:	Inverse hyperbolic sine of x , which is the value whose hyperbolic sine is x .

Constraints: x is integer, real, or complex.

Examples: `ARCSINH(-2.5)` returns -1.647231146371
`ARCSINH(0)` returns 0.0

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: `ARCCOSH`, `SIN`, `ARCSIN`

ARCTAN(x)

Returns: Inverse tangent of x , which is the angle (in radians) whose tangent is x

Constraints: x is integer, real or complex

Examples: `ARCTAN(0)` returns 0.0
`ARCTAN(1)` returns 0.7853981633974

Range: Returns a real value for integer or real arguments from $-\pi/2$ to $\pi/2$. Returns a complex value for complex arguments.

See also: `ARCSIN`, `ARCCOS`, `TAN`, `ARCTANH`

ARCTANH(x)

Returns: Inverse hyperbolic tangent of x .

Constraints: $-1 < x < 1$

Examples: `ARCTANH(0)` returns 0.0
`ARCTANH(0.5)` returns 0.5493061443341
`ARCTANH(0.5 + -0.5i)` returns 0.40236-0.55357i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: `ARCSINH`, `ARCCOSH`, `TANH`, `ARCTAN`

ARG(x)

Returns: The argument of complex x , which is the angle (in radians) of x in the complex plane.

Constraints: $x \neq 0i$

Examples: `ARG(1i)` returns 1.5707963268 (which is $\pi/2$).
`ARG(4-2i)` returns -0.463647609

Range: Returns a real value for integer, real, or complex arguments.

See also: `ARCTAN`, `ABS`

ARGCOUNT

Returns: A count of arguments passed to the program.

Examples: For the command line “mle -v prog.mle arg1 arg2 arg3”, `ARGCOUNT` returns 3: one for each argument following the program name, `prog.mle`.

Range:	Returns a non-negative integer value.
See also:	ARGSTRING, ENVCOUNT
<i>ARGSTRING(x)</i>	
Returns:	The x -th argument passed to the program. This capability is used to pass user-defined arguments to the program.
Constraints:	x is an integer
Examples:	<p>For the command line “mle -v -dl prog.mle arg1 arg2 arg3”</p> <p>ARGSTRING(1) returns the first argument after the program name, “arg1”</p> <p>ARGSTRING(3) returns “arg3”</p> <p>ARGSTRING(4) (and all values over 3) returns the empty string, “”</p> <p>ARGSTRING(0) returns the program name, “prog.mle”</p> <p>ARGSTRING(-1) returns the first argument before the program name, “-dl”</p> <p>ARGSTRING(-3) returns the third argument before the program name, which in this case is the mle program path and name “C:\MLE\MLE.EXE”</p> <p>ARGSTRING(-4) (and all values below -3) returns the empty string, “”</p>
Notes:	The particular way of grouping and quoting arguments is operating system specific.
See also:	ARGCOUNT, ENVSTRING
<i>BESSELI(n, y)</i>	
Returns:	The modified Bessel function of the first kind I (integer order n) of y .
Constraints:	n is a non-negative integer. y is integer or real.
Range:	Returns a real value.
Reference:	Press et al. (1989)
See also:	BESSELJ, BESSELK, BESSELY
<i>BESSELJ(n, y)</i>	
Returns:	The Bessel function of the first kind J (integer order n) of real y .
Constraints:	n is a non-negative integer. y is integer or real.
Range:	Returns a real value.
Reference:	Press et al. (1989)
See also:	BESSELI, BESSELK, BESSELY
<i>BESSELK(n, y)</i>	
Returns:	The modified Bessel function of the second kind K (integer order n) of real y .
Constraints:	n is a non-negative integer. y is integer or real.

Range:	Returns a real value.
Reference:	Press et al. (1989)
See also:	BESSELI, BESSELJ, BESSELY
<i>BESSELY</i> (<i>n</i> , <i>y</i>)	
Returns:	The Bessel function of the second kind <i>Y</i> (integer order <i>n</i>) of real <i>y</i> .
Constraints:	<i>n</i> is a non-negative integer. <i>y</i> is integer or real.
Range:	Returns a real value.
Reference:	Press et al. (1989)
See also:	BESSELI, BESSELJ, BESSELK
<i>BETA</i> (<i>v</i> , <i>ω</i>)	
Returns:	Euler's beta function. $\text{BETA}(v, \omega) = \int_0^1 z^{v-1} (1-x)^{\omega-1} dx.$
Constraints:	<i>v</i> > 0, <i>ω</i> > 0, integer or real arguments
Examples:	<code>BETA(5, 2)</code> returns 0.0095238095231 <code>BETA(4.0, 8.0)</code> returns 0.0007575757575
Range:	Returns a real value for integer or real values.
Reference:	Press et al. (1989)
See also:	BETA, IBETA, GAMMA, PDF BETA
<i>BOOL2STR</i> (<i>x</i>)	
Returns:	A string from boolean expression <i>x</i> .
Constraints:	<i>x</i> must be a boolean argument
Examples:	<code>BOOL2STR(1 <> 1)</code> returns the string “FALSE” <code>BOOL2STR(NOT FALSE)</code> returns the string “TRUE”
Range:	Returns string values “TRUE” or “FALSE”
See also:	INT2STR, REAL2STR
<i>CEIL</i> (<i>x</i>)	
Returns:	The least integer greater than or equal to <i>x</i> . For complex arguments, $\text{ceil}(a + bi) = -\text{floor}(-a) - \text{floor}(-b)i$
Constraints:	<i>x</i> is an integer, real or complex expression
Examples:	<code>CEIL(1.9)</code> returns 2.0 <code>CEIL(2.0)</code> returns 2.0 <code>CEIL(2.1)</code> returns 3.0 <code>CEIL(-1.9)</code> returns -1.0 <code>CEIL(-2.0)</code> returns -2.0 <code>CEIL(-2.1)</code> returns -2.0

`CEIL(2.4 + 3.1i)` returns 3.0+4.0i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments

See also: `FLOOR`, `ROUND`, `INT`

CHISQ(x, df)

Returns: The upper tail area (beyond x) of a χ^2 distribution with df degrees of freedom.

Constraints: $x \geq 0$, $df > 0$, x and df may be integer or real expressions

Examples: `CHISQ(3.84, 1)` returns 0.0500435311

`CHISQ(28.41, 20)` returns 0.1000435251

Range: Returns a real value for integer or real arguments.

See also: `INVCHISQ`, `STUDENTT`, `FDIST`, `PDF CHISQUARED`

CHR(x)

Returns: A character that is ASCII character x

Constraints: $0 \leq x \leq 255$; x is an integer expression

Examples: `CHR(66)` returns 'B'

`CHR(248)` returns '°' on DOS/Windows systems with extended ASCII character codes.

`CHR(248)` returns 'Ø' on some Unix systems.

Notes: Characters over 127 are operating-system specific.

Range: Returns a char value

See also: `ORD`

CLOCKSEED

Returns: A pseudo-random number based on system date and time

Notes: Concatenates the approximate number of days since 1 BC and adds the number of centiseconds in the current day to arrive at a somewhat random number. This function is useful for generating seeds for the random number generator.

Range: Returns an integer

See also: `RAND`, `SEED`

COMB(x, y)

Returns: The binomial coefficient, which is combinations of $x/$ elements taken $x2$ at a time, which is $x!/[y! (x - y)!]$. The result is 0.0 for $x \leq 0$, $y < 0$ or $x < y$. The result is 1.0 for $y = 0$.

Constraints: Integer arguments x and y .

Examples: `COMB(13, 10)` returns 286.0

`COMB(5, 5)` returns 1.0

Range:	Returns a real value
See also:	PERMUTATIONS
<i>COMP(x)</i>	
Returns:	Complement of x , which is $\text{SIGN}(1 - \text{ABS}(x), x)$.
Constraints:	x is an integer or real expression.
Examples:	<code>COMP(0)</code> returns 1.0 <code>COMP(0.25)</code> returns 0.75 <code>COMP(1)</code> returns 0
Range:	Returns a real for integer or real arguments.
See also:	COMPN
<i>COMPN(x, y)</i>	
Returns:	The y complement of x , which is $\text{SIGN}(y - \text{ABS}(x), x)$.
Constraints:	x is an integer or real expression.
Examples:	<code>COMPN(4, 3)</code> returns 1.0 <code>COMPN(-10, 2)</code> returns -8.0
Range:	Returns a real for integer or real arguments.
See also:	COMP
<i>CONCAT(x1, x2)</i>	
Returns:	The concatenation of two strings or characters.
Constraints:	$x1$ and $x2$ are string or char expressions
Examples:	<code>CONCAT("hello", " world")</code> returns the string hello world <code>CONCAT('a', 'b')</code> returns the string ab.
Range:	Returns a string value.
See also:	ADD
<i>COS(x)</i>	
Returns:	Cosine of x .
Constraints:	x is an integer, real or complex argument.
Examples:	<code>COS(0)</code> returns 1 <code>COS(1)</code> returns 0.5403023058681 <code>COS(DTOR(60))</code> returns 0.5
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.
See also:	SIN, TAN, ARCCOS
<i>COSH(x)</i>	
Returns:	The hyperbolic cosine of x . $\text{COSH}(x) = (e^x + e^{-x})/2$

Constraints	x is an integer, real or complex argument.
Examples:	<p><code>COSH(0)</code> returns 1</p> <p><code>COSH(4)</code> returns 27.308232836016</p> <p><code>COSH(1/2 + 1i)</code> returns 0.60926+0.43849i</p>
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.
See also:	<code>SINH</code> , <code>COS</code> , <code>ARCCOSH</code>
<i>COT</i> (x)	
Returns:	Cotangent of x (in radians).
Constraints	x is an integer, real or complex argument.
Examples:	<p><code>COT(10)</code> returns 1.5423510454</p> <p><code>COT(4)</code> returns 0.8636911545</p> <p><code>COT(1/2 + 1i)</code> returns 0.26117-1.12569i</p>
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.
See also:	<code>TAN</code> , <code>ARCCOT</code>
<i>COTH</i> (x)	
Returns:	The hyperbolic cotangent of x .
Constraints	x is an integer, real or complex argument.
Examples:	<p><code>COTH(0)</code> returns +∞</p> <p><code>COTH(-∞)</code> returns 0.8636911545</p> <p><code>COTH(-1)</code> returns -1.313035285</p> <p><code>COTH(1 + 0.5i)</code> returns 1.12569-0.26117i</p>
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.
See also:	<code>COT</code> , <code>ARCCOTH</code>
<i>CSCH</i> (x)	
Returns:	The hyperbolic cosecant of x .
Constraints	x is an integer, real or complex argument.
Examples:	<p><code>CSCH(0)</code> returns +∞</p> <p><code>CSCH(-∞)</code> returns 0.8636911545</p> <p><code>CSCH(-1)</code> returns -1.313035285</p> <p><code>CSCH(1 + 0.5i)</code> returns 1.12569-0.26117i</p>
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

DEC(x)

- Returns: $x - 1$
- Constraints: x is an integer, real, or complex expression
- Examples: `DEC(42)` returns 41
`DEC(-42)` returns -43
`DEC(3 + 2i)` returns 2+2i
`DEC(4.7)` returns 3.70
- Range: Returns a real, integer or complex value according to the type of x .
- See also: `INC`, procedure `DEC`

DEFAULTOUTNAME

- Returns: A reasonable output file name based on the name of the *mle* program. The function appends ".out" to the program name after stripping off a trailing suffix, if any.
- Examples: For programs called "sample.mle", "sample." or "sample" the function returns the string `sample.out`
- Notes: The line `OUTFILE(DEFAULTOUTNAME)` will automatically pick and assign a useful name for the output file. This is useful when you are constantly modifying and changing the name of a program.
- Range: Returns a string value.
- See also: `OUTFILE`, `DEFAULTPLOTNAME`

DEFAULTPLOTNAME

- Returns: A reasonable plot file name based on the name of the *mle* program. The function appends ".plt" to the program name after stripping off a trailing suffix, if any.
- Examples: For programs called "sample.mle", "sample." or "sample" the function returns the string `sample.plt`
- Notes: The line `PLOTFILE(DEFAULTPLOTNAME)` will automatically pick and assign a useful name for the plot file.
- Range: Returns a string value
- See also: Statement `PLOT`, procedure `PLOTFILE`, `DEFAULTOUTNAME`

DELTA(x , y)

- Returns: The Kronecker's delta function: 1 if $x = y$ otherwise 0.
- Constraints: x and y must be integer, real or complex expressions.
- Examples: `DELTA(10, 10)` returns 1
`DELTA(11, 10)` returns 0
`DELTA(1+2i, 1+2i)` returns 1
- Range: Returns an integer value
- See also: `HEAVISIDE`, `SGN`, `IF...THEN` function

DIREXISTS(*x*)

- Returns: TRUE if directory *x* exists, and FALSE if not.
- Constraints: *x* must be a string or character expression.
- Examples: `DIREXISTS("C:\")` returns true on DOS systems.
`DIREXISTS("C:\autoexec.bat")` (DOS).
`DIREXISTS("/dev")` returns true on Unix systems.
`DIREXISTS("/DEV")` returns false on Unix systems.
`DIREXISTS("/dev/null")` returns false on Unix systems.
- Notes: This function has some operating-system specific behavior. On DOS and Windows systems directory names are not case sensitive. On Unix systems, directory names are case sensitive. Directory separators are operating system specific ('/' on unix and '\' on DOS). DOS-based systems have *d*: as an optional drive prefix that is not applicable to Unix systems.
- Range: Returns a boolean value
- See also: EXISTS, FILESIZE, procedure ERASE

DIVIDE(*x*, *y*)

- Returns: *x* divided by *y*. This is the functional form of *x/y*.
- Constraints: *y* ≠ 0, *x* and *y* are any mix of numeric types (integer, real, complex)
- Examples: `DIVIDE(10, 2)` returns 5.0
`DIVIDE(10+0i, 2)` returns 5.0+0.0i
- Range: Returns a real value if both argument are real or integer, and a complex value if either argument is complex.
- See also: MULTIPLY, MODULO, REMAINDER, operators: DIV, MOD, /, *

DMSTOD(*x*, *y*, *z*)

- Returns: An angle in degrees from an angle in degrees (*x*), minutes (*y*), and seconds (*z*).
- Constraints: *x*, *y*, and *z* integer or real expressions
- Examples: `DMSTOD(34, 15, 10.2)` returns 34.252833333333
`DMSTOD(0, 20, 15)` returns 0.33750
- Notes: Angles specified in degrees and decimal minutes can be used as well, but the third argument must be specified as zero. For example `DMSTOD(43, 8.33, 0)` returns 43.13883.
- Range: Returns a real value
- See also: DMSTOR

DMSTOR(*x*, *y*, *z*)

- Returns: An angle in radians from an angle in degrees (*x*), minutes (*y*), and seconds (*z*).

Constraints	x , y , and z integer or real expressions
Examples:	<code>DMSTOR(34, 15, 10.2)</code> returns 0.5978247198035 <code>DMSTOR(0, 20, 15)</code> returns 0.0058904862255
Notes:	Angles specified in degrees and decimal minutes can be used as well, but the third argument must be specified as zero. For example <code>DMSTOR(122, 18.11, 0)</code> returns 2.13457.
Range:	Returns a real value
See also:	DMSTOD
<i>DMYTOJ</i> (x , y , z)	
Returns:	A Julian day from day (x), month (y), and year (z).
Constraints	$1 \leq x \leq 31$, $1 \leq y \leq 12$, x , y , and z are integer expressions
Examples:	<code>DMYTOJ(15, 1, 2000)</code> returns the Julian day 2451559 <code>DMYTOJ(4, 7, 1776)</code> returns the Julian day 2369916
Notes:	This function is useful for computing durations between two dates in failure time models. For example, the duration between "birth" on 16 Feb 1976 and "death" on 21 Jul 1992 would be computed as <code>DMYTOJ(21, 07, 1992) - DMYTOJ(16, 02, 1976)</code> , which returns exactly 6000 days.
Range:	Returns an integer value
See also:	JULIAND, JULIANM, JULIANY, YEARDAY, WEEKDAY
<i>DTOR</i> (x)	
Returns:	Degrees from radians, $\pi x/180$.
Constraints	x is an integer or real expression.
Examples:	<code>DTOR(30)</code> returns 0.5235987755983 <code>DTOR(180)</code> returns 3.1415926535898
Range:	Returns a real value
See also:	RTOD
<i>EARTHDIST</i> ($lon1$, $lat1$, $lon2$, $lat2$)	
Returns:	The distance between two points on the earth surface.
Constraints	Arguments are integer or real expressions
Examples:	Madison, WI is at 40° 8.33' North and 89° 20.24' West, and Seattle, WA is at 47° 31.79' North and 122° 18.11' West. The distance between these cities is found as <code>EARTHDIST(DMSTOD(40, 8.33, 0), DMSTOD(89, 20.24, 0), DMSTOD(47, 31.79, 0), DMSTOD(122, 18.11, 0))</code> , which returns 2768.737.
Range:	Returns a real value
See also:	DMSTOD

ENVCOUNT

- Returns: A count of operating system environment variables. This function is used to loop through or otherwise access all environment variables using the `ENVSTRING` function.
- Examples: `ENVCOUNT` on a Windows system might return 36. The value can be used to loop through all environment variables.
- Range: Returns an integer value.
- See also: `ENVSTRING`, `GETENV`, `ARGCOUNT`

ENVSTRING(x)

- Returns: The x -th operating system environment variable. This capability is used to search environment variables.
- Constraints: Argument is an integer expression. Returns an empty string if x is outside 0 .. `ENVCOUNT`
- Examples: `ENVSTRING(12)`, for example, might return “CLIENTNAME=Console” on a Windows system.
- `ENVSTRING(5)`, for example, might return “HOSTNAME=wright” on a Unix system.
- Range: Returns a string value.
- See also: `ENVCOUNT`, `GETENV`, `ARGSTRING`

EOF(x)

- Returns: The end-of-file status of file x .
- Constraints: x must be a file open for reading.
- See also: `EOLN`, Procedures `OPENREAD`, `CLOSE`.
- Range: Returns a boolean value.

EOLN(x)

- Returns: The end-of-line status of file x . The status is either `TRUE` or `FALSE`
- Constraints: x must be a file that is open for reading.
- Range: Returns a boolean value.
- See also: `EOF`, Procedure `OPENREAD`.

ERF(x)

- Returns: The error function, which is a normalized integral from 0 to x of an exponential power function.
- Constraints: None, but x should be non-negative in order to return a proper probability
- Examples: `ERF(1)` returns 0.8427006949
- `ERF(2)` returns 0.9953221398
- Range: Returns a real number from 0 to 1 (for non-negative x).
- See also: `ERFC`, `NORMAL`, `PDF NORMAL`

ERFC(*x*)

Returns: The complementary error function, which is $1 - \text{ERF}(x)$.

Constraints: None, but *x* should be non-negative in order to return a proper probability.

Examples: `ERFC(0.75)` returns 0.2888444222308
`ERFC(2)` returns 0.0046778602160
`ERFC(-2)` returns 1.9953221397840, which is not a probability

Range: Returns a real number from 0 to 1 (for non-negative *x*).

See also: `ERF`, `NORMAL`, `PDF NORMAL`

EXEC(*x*, *y*)

Returns: An error code after executing command *x* with arguments *y*

Constraints: *x* and *y* are strings or char arguments.

Examples: `EXEC('command.com', '/c dir')` returns 0 if no error occurs after executing the DOS `dir` command.
`EXEC('sort.exe', ' < f.dat > fs.dat')` returns 0 if no error occurs after executing the DOS `sort` command.
`EXEC('ls', '-l')` returns 0 if no error occurs after executing the Unix `ls` command.

Notes: The `EXEC` function searches the path environment variable to locate the command specified as the first argument. Note that some commands in DOS are built into `command.com`. These commands must be run as an argument to `command.com` (as in the first example). Error values that can be returned are:

2	File not found
3	Path not found
5	Access denied
6	Invalid handle
8	Not enough memory
10	Invalid environment
11	Invalid format
18	No more files

Range: Returns an integer.

See also: Procedure `EXEC`

EXISTS(*x*)

Returns: `TRUE` if file *x* exists, and `FALSE` if not.

Constraints: *x* is a string or char argument.

Examples: `EXISTS("C:\autoexec.bat")` returns `TRUE` on DOS systems.
`EXISTS("/dev/null")` returns `TRUE` on Unix systems.
`EXISTS("/DEV/NULL")` returns `FALSE` on Unix systems.

Notes:	This function has some operating-system specific behavior. On DOS and Windows systems file names are not case sensitive, and directories always return FALSE. On Unix systems file names are case sensitive, and return TRUE for directories.
Range:	Returns a boolean.
See also:	DIREXISTS, FILESIZE, procedure ERASE
<i>EXP(x)</i>	
Returns:	The value e raised to the power x , e^x .
Constraints:	x is integer, real, or complex.
Examples:	<p>EXP(0.2) returns 1.2214027581602</p> <p>EXP(0) returns 1</p> <p>EXP(-0.2) returns 0.1353352832366</p> <p>EXP(0.6 + -0.3i) returns 1.74074-0.53847i</p>
Range:	Returns a real value for an integer or real argument. Returns a complex value for a complex argument.
See also:	LN, POWER
<i>FACT(x)</i>	
Returns:	The factorial function, $x!$, which is $x \times (x-1) \times (x-2) \times \dots \times 2 \times 1$.
Constraints:	The FACT function must be called with integer values
Examples:	<p>FACT(5) returns 120.0</p> <p>FACT(100) returns 9.332622E+0157</p>
Range:	The function always returns a real value
Notes:	The function can overflow the real number representation for modest values of x
See also:	GAMMA, LNGAMMA
<i>FDIST(x, df1, df2)</i>	
Returns:	The upper tail area (beyond x) of an F distribution with $df1$ and $df2$ degrees of freedom.
Constraints:	$x \geq 0$, $df1 > 0$, $df2 > 0$. x , $df1$, and $df2$ can be integer or real.
Examples:	<p>FDIST(7.71, 1, 4) returns 0.0499875403</p> <p>FDIST(3.6, 6, 13) returns 0.0251025332</p>
Range:	The function always returns a real value
Notes:	The F distribution is used to compare sample variances from normal populations.
See also:	INVFDIST, CHISQ, STUDENTT
<i>FILESIZE(x)</i>	
Returns:	The number of bytes in the file named x .

Constraints:	x is a string or character value.
Examples:	<pre>FILESIZE("prog.mle") might return 533</pre> <pre>FILESIZE("/dev/null") returns 0 (on Unix)</pre>
Range	The function always returns an integer
Notes:	Returns zero if the file does not exist or if the file is empty. Filenames are case-sensitive on Unix and are not case sensitive on DOS/Windows.
See also:	EXISTS
<i>FISHER</i> (x)	
Returns:	Fisher's transformation as $\frac{1}{2} \ln \left(\frac{1+x}{1-x} \right)$.
Constraints:	$-1 < x < 1$, where x is an integer or real value
Examples:	<pre>FISHER(0) returns 0.0</pre> <pre>FISHER(0.5) returns 0.5493061443341</pre> <pre>FISHER(-0.99990000) returns -4.951718775643</pre>
Range	The function always returns a real value
Notes:	The Fisher's transformation can be used to transform parameters that must take on values from -1 to 1 into parameters that take on values from $-\infty$ to ∞ .
See also:	FISHERINV
<i>FISHERINV</i> (x)	
Returns:	The inverse Fisher's transformation as .
Constraints:	An integer or real value for x
Examples:	<pre>FISHERINV(4) returns 0.9993292997391</pre> <pre>FISHERINV(0.5) returns 0.4621171572600</pre> <pre>FISHERINV(0) returns 0.0</pre>
Range:	Returns a real value between -1 and 1
See also:	FISHER
<i>FLOOR</i> (x)	
Returns:	The greatest integer less than or equal to x , a real number.
Constraints:	x must be real, complex, or integer.
Examples:	<pre>FLOOR(1.9) returns 1.0</pre> <pre>FLOOR(2.0) returns 2.0</pre> <pre>FLOOR(2.1) returns 2.0</pre> <pre>FLOOR(-1.9) returns -2.0</pre> <pre>FLOOR(-2.0) returns -2.0</pre>

FLOOR(-2.1) returns -3.0

FLOOR(3.3 + 3.3i) returns 3.0 + 3.0i

Range: Returns a real or a complex value

See also: CEIL, ROUND, INT

FRAC(x)

Returns: The fractional part of x .

Constraints: x must be real, complex, or integer.

Examples: FRAC(2.0) returns 0.0

FRAC(2.1) returns 0.1

FRAC(-3.2) returns -0.2

FRAC(2.3 + 4.5i) returns 0.3 + 0.5i

Range: Returns a real value for integer and real arguments, and a complex value for complex arguments. The returned value falls between -1 and 1.

See also: INT

GAMMA(x)

Returns: Euler's gamma function, $\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt$

Constraints: x must be a positive real or integer value.

Examples: GAMMA(4) returns 6

GAMMA(PI) returns 2.2880377950731

GAMMA(2000) returns +oo

Range: Returns a real value between 1 and $\pm\infty$.

See also: IGAMMA, IGAMMAC, IGAMMAE, PDF GAMMA

GCF(x, y)

Returns: The greatest common factor of x and y , which is the greatest value that divides both x and y exactly.

Constraints: x and y must be integer values.

Examples: GCF(81, 36) returns 9

GCF(143, 187) returns 11

Range: Returns an integer value.

See also: LCM

GETDIR

Returns: The name of the current directory.

Examples: GETDIR, might return "D:\work" on a Windows machine

GETDIR, might return “/usr/djholman/work” on a Unix machine.

Range: Returns a string.

Notes: On DOS/Windows systems, a drive letter and colon will be returned and the directory separator will be ‘\’. On Unix systems, a drive letter will not appear and the directory separator will be ‘/’.

See also: MKDIR, CHDIR, RMDIR, ENVCOUNT, ENVSTRING

GETENV(x)

Returns: The value of the operating system environment variable matching x . This capability is used to obtain values of environment variables.

Constraints: x is a string or char argument.

Examples: GENENV(“HOME”), might return “D:\” in Windows
GENENV(“HOME”), might return “/usr/djholman” in Unix.

Range: Returns a string.

Notes: The case of the argument is not significant on DOS and Windows systems. The case of the argument is significant on Unix systems. A non-existent argument results in an empty string being returned.

See also: ENVCOUNT, ENVSTRING

HEAVISIDE(x)

Returns: the Heaviside function, which is 1 if $x \geq 0$ otherwise it returns 0.

Constraints: x must be an integer or real argument.

Examples: HEAVISIDE(3.) returns 1
HEAVISIDE(0) returns 1
HEAVISIDE(-2) returns 0

Range: Returns an integer, either 0 or 1

See also: DELTA, SGN, IF...THEN function

IBETA(p, n, w)

Returns: The normalized Euler's incomplete beta function.

$$\text{BETA}(p, v, \omega) = \frac{\int_0^p x^{v-1} (1-x)^{\omega-1} dx}{\text{BETA}(v, \omega)}$$
, which is the beta cumulative density function.

Constraints: Arguments are real or integer; $0 \leq p \leq 1$, $v > 0$, $\omega > 0$

Examples: IBETA(0.5, 3, 6) returns 0.8554687499873
IBETA(1, 3, 3) returns 1.0

Range: Returns a real number.

See also: BETA, IBETAC, GAMMA, PDF BETA

IBETAC(*p*, *n*, *w*)

Returns: The complement of the normalized Euler's incomplete beta function. $IBETAC(p, v, \omega) = 1 - IBETA(p, v, \omega)$. This is the beta survival function.

Constraints: $0 \leq p \leq 1, v > 0, \omega > 0$

Examples: `IBETAC(0.5, 3, 6)` returns 0.1445312500127
`IBETAC(1, 3, 3)` returns 0.0

Range: Returns a real number.

See also: BETA, IBETA, GAMMA, PDF BETA

IDIV(*x*, *y*)

Returns: The integer part of x/y . This is the same as the algebraic expression $x \text{ DIV } y$.

Constraints: $y \neq 0$. x and y must be integers

Examples: `IDIV(104, 25)` returns 4
`IDIV(-124, 25)` returns -4

Range: Returns an integer.

See also: MODF, MODULO, REMAINDER, DIVIDE

IGAMMA(*x*, *y*)

Returns: Euler's incomplete gamma function: $\Gamma(x)^{-1} \int_0^y e^{-t} t^{x-1} dt$. This is, essentially, the cumulative density function for the gamma function.

Constraints: $x > 0, y \geq 0$.

Examples: `IGAMMA(4, oo)` returns 1.0
`IGAMMA(4, 0)` returns 0.0
`IGAMMA(3, 1)` returns 0.0803013962
`IGAMMA(1, 3)` returns 0.9502129316

Range: Returns a real number from 0.0 to 1.0.

See also: GAMMA, IGAMMAC, IGAMMAE

IGAMMAC(*x1*, *x2*)

Returns: The complement of the Euler's incomplete gamma function.

Constraints: $x > 0, y \geq 0$.

Examples: `IGAMMAC(4, oo)` returns 0.0
`IGAMMAC(4, 0)` returns 1.0
`IGAMMAC(3, 1)` returns 0.9196986038
`IGAMMAC(1, 3)` returns 0.0497870684

Range: Returns a real number from 0.0 to 1.0.

See also:	GAMMA, IGAMMA, IGAMMAE
<i>IGAMMAE</i> ($x1$, $x2$)	
Returns:	IGAMMA($x1$, $x2$)*ROOT($x2$, $x1$).
Constraints:	$x > 0, y \geq 0$.
Examples:	IGAMMAE(2, 2) returns 0.1484985302 IGAMMAE(4, 2) returns 0.0089297835 IGAMMAE(3, 1) returns 0.0803013962 IGAMMAE(1, 3) returns 0.0000000000
Range:	Returns a real number.
See also:	GAMMA, IGAMMA, IGAMMAC
<i>IM</i> (x)	
Returns:	The imaginary part of complex argument x
Constraints:	x must be an integer, real or complex
Examples:	IM(3 - 4i) returns 4.0 IM(3) returns 0.0
Range:	Returns a real number.
See also:	RE
<i>INC</i> (x)	
Returns:	$x + 1$
Constraints:	x must be an integer, real or complex
Examples:	INC(42) returns 43 INC(-42) returns -41 INC(-42.0) returns -41.0 INC(4 + 2i) returns 5.00000+2.00000i
Range:	Returns a real number for integer and real arguments and a complex number for complex arguments.
See also:	DEC, procedure INC
<i>INT</i> (x)	
Returns	The integer part of x as a real number.
Constraints:	x must be an integer, real or complex
Examples:	INT(1.9) returns 1.0 INT(2) returns 2.0 INT(-1.9) returns -1.0 INT(4.3 + 1.3i) returns 4.00000+1.00000i

Range: Returns a real number for integer and real arguments and a complex number for complex arguments.

See also: FRAC, CEIL, ROUND, FLOOR

INT2STR(x)

Returns: A string representation of a number from integer expression x .

Constraints: x must be an integer.

Examples: `INT2STR(-123)` returns the string `-123`
`INT2STR(0xMCMXII)` returns the string `1912`

Range: Returns a string.

See also: BOOL2STR, REAL2STR

INVBETA(p, n, w)

Returns: The inverse of the Beta distribution, probability p with parameters n and w .

Constraints: $0 \leq p \leq 1$, $df1 > 0$, $df2 > 0$

Examples: `INVBETA(0.5, 1, 1)` returns `0.50`
`INVBETA(0.5, 1, 2)` returns `0.2928930918`

Range: Returns a real number from 0.0 to 1.0.

See also: BETA, BETAI, INVCHISQ, INVSTUDENTT, QUANTILE *<pdf>*

INVCHISQ(p, df)

Returns: The inverse of the chi-squared cumulative density function at probability p and df degrees of freedom.

Constraints: $0 \leq p \leq 1$, $df > 0$

Notes: This function can be used to evaluate critical values of statistics that are $\sim \chi^2$.

Examples: `INVCHISQ(0.95, 1)` returns `3.8414592390634`
`INVCHISQ(0.95, 5)` returns `11.070497756345`

Range: Returns a real number.

See also: CHISQ, INVSTUDENTT, INVFDIST, QUANTILE *<pdf>*

INVERT(x)

Returns: $1/x$.

Constraints: x is a real, integer, or complex.

Examples: `INVERT(2)` returns `0.5`
`INVERT(-1.25)` returns `-0.8`
`INVERT(3 + 2i)` returns `0.23077-0.15385i`

Range: Returns a real number for real and integer arguments. Returns a complex value for complex arguments.

INVFDIST(*p*, *df1*, *df2*)

- Returns: The inverse of the F distribution, probability *p* with *df1* and *df2* degrees of freedom.
- Constraints: $0 \leq p \leq 1$, $df1 > 0$, $df2 > 0$
- Notes: The F distribution is used to compare sample variances from normal populations.
- Examples: `INVFDIST(0.05, 1, 10)` returns 4.9646147898
`INVFDIST(0.025, 8, 9)` returns 4.1019409049
- Range: Returns a real number.
- See also: `FDIST`, `INVCHISQ`, `INVSTUDENTT`, `QUANTILE` <*pdf*>

INVNORMAL(*p*)

- Returns: The inverse of the standard normal cumulative density function at probability *p*.
- Constraints: $0 < p < 1$
- Notes: This function can be used to evaluate critical values of statistics that are normally distributed.
- Examples: `INVNORMAL(0.95)` returns 1.6448536392367
`INVNORMAL(0.975)` returns 1.9599639986264
- Range: Returns a real number.
- See also: `QUANTILE` <*pdf*>

INVSTUDENTT(*p*, *df*)

- Returns: The inverse of the Student *t* survival density function at probability *p* and *df* degrees of freedom.
- Constraints: Arguments are real or integer; $0 \leq p \leq 1$, $df > 0$
- Notes: This function can be used to evaluate critical values of statistics of means when the variance is unknown.
- Examples: `INVSTUDENTT(0.05, 1)` returns 6.3137491419, which is the point of the distribution above which 5% of the area occurs.
`INVSTUDENTT(0.95, 5)` returns -2.015048392.
- Range: Returns a real number.
- See also: `STUDENTT`, `INVCHISQ`, `INVFDIST`, `QUANTILE` <*pdf*>

IRAND(*x*, *y*)

- Returns: A random integer from *x* to *y*.
- Constraints: *x* and *y* are integer arguments.
- Examples: `IRAND(5, 10)` return a value uniformly drawn from 5 to 10.
`IRAND(10, 5)` will return a value uniformly drawn from 5 to 10.

Notes: Before `IRAND` or other random number functions can be used, the value of `RANDOMSEED` must be set to a positive constant. Use the `SEED()` procedure to do this.

Range: Returns an integer from x to y .

See also: `RAND`, `RRAND`, procedure `SEED`, `RANDOMSEED`

ISANINFINITY(x)

Returns: `TRUE` if x is either ∞ or $-\infty$.

Constraints: x must be real or integer. If integer, it is converted to real (integer infinity is not defined).

Examples: `ISANINFINITY(-oo)` returns `TRUE`
`ISANINFINITY(-1/0)` returns `TRUE`
`ISANINFINITY(34)` returns `FALSE`

Range: Returns a boolean value for integer or real arguments.

See also: `ISINFINITY`, `ISNEGINFINITY`, `ISNAN`

ISEQ(x, y)

Returns: The boolean $x = y$. This is the functional form of $x = y$ (or $x == y$).

Constraints: x and y must be compatible types: a pair of numeric types, boolean types, or string/char types.

Examples: `ISEQ(10, 10.0)` returns `TRUE`
`ISEQ('a', "a")` returns `TRUE`
`ISEQ(33, 4)` returns `FALSE`

Range: Returns a boolean.

See also: `ISGE`, `ISGT`, `ISLE`, `ISLT`, `ISNE`, `ISNEAR`

ISEVEN(x)

Returns: `TRUE` if integer x is an even number, `FALSE` if x is odd

Constraints: x is an integer.

Examples: `ISEVEN(0)` returns `TRUE`
`ISEVEN(199)` returns `FALSE`

Range: Returns a boolean.

See also: `ISODD`

ISGE(x, y)

Returns: The boolean $x \geq y$. This is the functional form of $x \geq y$.

Constraints: x and y must be compatible types: a pair of numeric types, boolean types, or string/char types.

Examples: `ISGE(10, 10.0)` returns `TRUE`
`ISGE('b', 'a')` returns `TRUE`

`ISGE('a', 'b')` returns FALSE

`ISGE(33, 4)` returns TRUE

Range: Returns a boolean.

See also: `ISEQ, ISGT, ISLE, ISLT, ISNE, ISNEAR`

ISGT(x, y)

Returns: The boolean $x > y$. This is the functional form of $x > y$.

Constraints: x and y must be compatible types: a pair of numeric types, boolean types, or string/char types.

Examples: `ISGT(10, 10.0)` returns FALSE

`ISGT('b', 'a')` returns TRUE

`ISGT('a', 'b')` returns FALSE

`ISGT(33, 4)` returns TRUE

Range: Returns a boolean.

See also: `ISGE, ISEQ, ISLE, ISLT, ISNE, ISNEAR`

ISINFINITY(x)

Returns: TRUE if x is ∞ .

Constraints: x must be real or integer. If integer, it is converted to real (integer infinity is not defined).

Examples: `ISINFINITY(oo)` returns TRUE

`ISINFINITY(1/0)` returns TRUE

`ISINFINITY(-1/0)` returns FALSE

`ISINFINITY(34)` returns FALSE

Range: Returns a boolean value for integer or real arguments.

See also: `ISANINFINITY, ISNEGINFINITY, ISNAN`

ISLE(x, y)

Returns: The boolean $x \leq y$. This is the functional form of $x \leq y$.

Constraints: x and y must be compatible types: a pair of numeric types, boolean types, or string/char types.

Examples: `ISLE(10, 10.0)` returns TRUE

`ISLE('b', 'a')` returns FALSE

`ISLE('a', 'b')` returns TRUE

`ISLE(33, 4)` returns FALSE

Range: Returns a boolean.

See also: `ISGE, ISGT, ISEQ, ISLT, ISNE, ISNEAR`

ISLT(x, y)

Returns: The boolean $x < y$. This is the functional form of $x < y$.

Constraints: x and y must be compatible types: a pair of numeric types, boolean types, or string/char types.

Examples: `ISLT(10, 10.0)` returns FALSE
`ISLT('b', 'a')` returns FALSE
`ISLT('a', 'b')` returns TRUE
`ISLT(33, 4)` returns FALSE

Range: Returns a boolean.

See also: ISGE, ISGT, ISLE, ISEQ, ISNE, ISNEAR

ISNAN(x)

Returns: TRUE if x is not a valid number (NaN).

Constraints: x must be real or integer. If integer, it is converted to real (integer NAN is not defined).

Examples: `ISNAN(-oo)` returns FALSE
`ISNAN(oo + oo)` returns TRUE
`ISNAN(1/oo)` returns TRUE
`ISNAN(34)` returns FALSE

Range: Returns a boolean value for integer or real arguments.

Notes: Most algebraic operations with infinity return a NaN floating point value.

See also: ISANINFINITY, ISINFINITY, ISNEGINFINITY

ISNE(x, y)

Returns: The boolean $x \neq y$. This is the functional form of $x \neq y$.

Constraints: x and y must be compatible types: a pair of numeric types, boolean types, or string/char types.

Examples: `ISNE(10, 10.0)` returns FALSE
`ISNE('b', 'a')` returns TRUE
`ISNE(33, 4)` returns TRUE

Range: Returns a boolean.

See also: ISGE, ISGT, ISLE, ISLT, ISEQ, ISNEAR

ISNEAR(x, b, d)

Returns: TRUE if x is in the interval $[b - d, b + d]$; otherwise it returns FALSE.

Constraints: x and y must be integer or real.

Examples: `ISNEAR(23.5, 20, 4)` returns TRUE
`ISNEAR(23.5, 20, 1)` returns FALSE

Range: Returns a boolean.

See also: ISGE, ISGT, ISLE, ISLT, ISNE, ISEQ

ISNEGINFINITY(x)

Returns: TRUE if x is $-\infty$.

Constraints: x must be real or integer. If integer, it is converted to real (integer infinity is not defined).

Examples: `ISNEGINFINITY(-oo)` returns TRUE
`ISNEGINFINITY(-1/0)` returns TRUE
`ISNEGINFINITY(1/0)` returns FALSE
`ISNEGINFINITY(34)` returns FALSE

Range: Returns a boolean value.

See also: `ISANINFINITY`, `ISINFINITY`, `ISNAN`

ISODD(x)

Returns: TRUE if integer x is an odd number, FALSE if x is even

Constraints: x must be an integer.

Examples: `ISODD(0)` returns FALSE
`ISODD(199)` returns TRUE

Range: Returns a boolean value.

See also: `ISEVEN`

JULIAND(x)

Returns: The day of the month for a Julian day.

Constraints: x must be an integer that represents a Julian day.

Examples: `DMYTOJ(30, 6, 1961)` returns 2437481 so that `JULIAND(2437481)` returns 30.

Range: 1 to 31

See also: `JULIANM`, `JULIANY`, `YEARDAY`, `WEEKDAY`, `DMYTOJ`

JULIANM(x)

Returns: The month for a Julian day.

Constraints: x must be an integer that represents a Julian day.

Examples: `DMYTOJ(30, 6, 1961)` returns 2437481 so that `JULIANM(2437481)` returns 6

Range: 1 to 12

See also: `JULIAND`, `JULIANY`, `YEARDAY`, `WEEKDAY`, `DMYTOJ`

JULIANY(x)

Returns: The year for a Julian day.

Constraints: x must be an integer that represents a Julian day.

Examples: `DMYTOJ(30, 6, 1961)` returns 2437481, so that `JULIANY(2437481)` returns 1961

Range:	Returns an integer that is the year.
See also:	JULIAND, JULIANM, YEARDAY, WEEKDAY, DMYTOJ
<i>LCM</i> (<i>x</i> , <i>y</i>)	
Returns:	The least common multiple of <i>x</i> and <i>y</i> .
Constraints:	<i>x</i> and <i>y</i> must be an integers.
Examples:	LCM(9 , 12) returns 36
Range:	Returns an integer.
See also:	GCF
<i>LEAPYEAR</i> (<i>y</i>)	
Returns:	TRUE if the year specified by <i>y</i> is a leap year.
Constraints:	<i>y</i> is an integer that represents a year.
Examples:	LEAPYEAR(2000) returns TRUE LEAPYEAR(2001) returns FALSE LEAPYEAR(1900) returns FALSE
Range:	Returns a boolean value.
See also:	MONTHDAYS
<i>LEFTSTRING</i> (<i>x</i> , <i>y</i>)	
Returns:	The leftmost substring from <i>x</i> of up to <i>y</i> characters.
Constraints:	<i>x</i> is a string, <i>y</i> is an integer.
Examples:	LEFTSTRING("Probability theory", 11) returns the string Probability LEFTSTRING("Anyway", 3) returns the string Any LEFTSTRING("Anyway", 20) returns the string Anyway
Range:	Returns a string. An empty string is returned if $y \leq 0$ or if <i>x</i> is an empty string.
See also:	RIGHTSTRING, SUBSTRING
<i>LN</i> (<i>x</i>)	
Returns:	The natural (Naperian) logarithm of <i>x</i> (also LOG).
Constraints:	$x \geq 0$. If $x = 0$, $-\infty$ is returned. <i>x</i> is an integer, real, or complex.
Examples:	LN(E) returns 1.0 LN(E^25) returns 25 LN(35) returns 3.5553480614894 LN(2 + 23i) returns 3.13926+1.48406i
Range:	Returns a real number.
See also:	LOG, LOGBASE, EXP

LNFACT(*x*)

Returns: The natural logarithm of $x!$.

Constraints: $x \geq 0$. x is an integer.

Examples: `LNFACT(10)` returns 15.104412573076
`LNFACT(3000)` returns 21024.024853046

Range: Returns a real number.

Notes: The `LNFACT` function is useful for intermediate calculations to avoid numeric overflow. If real values of x are needed, the `LNGAMMA(x + 1)` function can be used.

See also: `FACT`, `LNGAMMA`

LNGAMMA(*x*)

Returns: The natural logarithm of `GAMMA(x)`.

Constraints: $x \geq 0$. x is an integer or real.

Example: `LNGAMMA(11)` returns 15.104412572871
`LNGAMMA(3001)` returns 21024.024853046

Range: Returns a real number.

Notes: The `LNFACT` function is useful for intermediate calculations to avoid numeric overflow. If integer values of x are being used, the `LNFACT(x - 1)` function can be used.

See also: `GAMMA`, `LNFACT`

LOG(*x*)

Returns: The natural (Naperian) logarithm of x (also `LOG`).

Constraints: $x \geq 0$. If $x = 0$, $-\infty$ is returned. x is an integer, real, or complex.

Examples: `LOG(E)` returns 1.0
`LOG(E^25)` returns 25
`LOG(35)` returns 3.5553480614894
`LOG(2 + 23i)` returns 3.13926+1.48406i

Range: Returns a real number.

See also: `LOG`, `LOGBASE`, `EXP`

LOG10(*x*)

Returns: The base-10 logarithm x .

Constraints: $x \geq 0$. If $x = 0$, $-\infty$ is returned. x is an integer, real or complex number.

Examples: `LOG10(10)` returns 1.0
`LOG10(10^25)` returns 25.0
`LOG10(35)` returns 1.5440680443503

`LOG10(3 + 1i)` returns 0.50000+0.13973i

Range: Returns a real number.

See also: LN, LOG, LOGBASE, EXP

LOGBASE(x , y)

Returns: The logarithm (base y) of x .

Constraints: $x \geq 0$. If $x = 0$, $-\infty$ is returned. x is an integer, real or complex number.

Examples: `LOGBASE(10, 10)` returns 1.0
`LOGBASE(10^25, 10)` returns 25
`LOGBASE(35, E)` returns 3.5553480614894

Range: Returns a real number.

See also: LN, LOG, LOG10, EXP

LOGISTIC(x)

Returns: The logistic function: $1/[1 + \exp(x)]$, or its complement if `ALT_LOGISTIC=TRUE`.

Constraints: x is an integer, real or complex number.

Examples: `LOGISTIC(-1)` returns 0.7310585786
`LOGISTIC(-1 + 0.5i)` returns 0.74274-0.09903i
`LOGISTIC(10)` returns 0.0000453979

Range: Returns a real number from 0 to 1.

Note: The `LOGISTIC` function is widely used to transform parameters that are probabilities (range from 0 to 1) into parameters that range from $-\infty$ to ∞ . Logistic transformations are sometimes useful to change a variable in the range $[-\infty, \infty]$ to a transformed variable in the range $[0, 1]$. There are two common ways to construct these transformations. The transformation is $p_1 = 1/(1 + e^t)$, and the second transformation is $p_2 = e^t/(1 + e^t)$. Both transformation are equally useful, and there is no reason except habit to choose one over the other. Mathematically, they are related as complements; that is, $p_1 = 1 - p_2$. By default, *mle* uses the transformation to p_1 for both the `LOGISTIC()` function call, and the `FORM = LOGISTIC` parameter transformation. You can change *mle* to use the p_2 form of the equation by setting the variable `ALT_LOGISTIC = TRUE`. The default value of `ALT_LOGISTIC` is false.

See also: LOGIT, PDF LOGISTIC, variable ALT_LOGISTIC

LOGIT(p)

Returns: The logit transform, $\ln[p/(1 - p)]$.

Constraints	p is a probability, $0 < p < 1$. If $x = 0$.
Examples:	<p><code>LOGIT(1)</code> returns ∞</p> <p><code>LOGIT(0.5)</code> returns 0.0</p> <p><code>LOGIT(0.3)</code> returns -0.847297860387</p> <p><code>LOGIT(0)</code> returns $-\infty$</p>
Notes:	The <code>LOGIT</code> function is the inverse of the <code>LOGISTIC</code> function when <code>ALT_LOGISTIC</code> is <code>TRUE</code> , or the negative inverse when <code>ALT_LOGISTIC</code> is <code>FALSE</code> . Hence, <code>LOGIT(LOGISTIC(5))</code> returns 5.0 or -5.0.
Range:	Returns a real number.
See also:	<code>LOGISTIC</code>
<i>LUNARPHASE(j)</i>	
Returns:	An approximate phase of the moon on Julian date j .
Constraints:	j must be an integer.
Examples:	<p><code>LUNARPHASE(DMYTOJ(15, 1, 2000))</code> returns 0.6055993482011</p> <p><code>LUNARPHASE(DMYTOJ(16, 1, 2000))</code> returns 0.6733257524912</p> <p>A new moon occurred on 3 Mar 2003, hence the smallest value occurs around this day.</p> <p><code>LUNARPHASE(DMYTOJ(2, 3, 2003))</code> returns 0.0508469525</p> <p><code>LUNARPHASE(DMYTOJ(3, 3, 2003))</code> returns 0.0168794517</p> <p><code>LUNARPHASE(DMYTOJ(4, 3, 2003))</code> returns 0.0846058560</p>
Range:	Returns a real number from 0 to 1. 0 denotes a new moon, and 1 denotes a full moon.
Reference:	Press et al. (1989)
See also:	<code>DMYTOJ</code>
<i>MAX(x, y)</i>	
Returns:	The greatest of x and y .
Constraints:	x and y must be compatible types: a pair of integer/real or a pair of string/char values. Complex and boolean values are not permitted.
Examples:	<p><code>MAX(15, 10)</code> returns 15</p> <p><code>MAX(-15, -10)</code> returns -10</p> <p><code>MAX('a', 'b')</code> returns 'b'</p>
Range:	Returns a real number if one of the arguments is real, an integer if both arguments are integer, a string if either argument is a string, or a char if both arguments are chars.
See also:	<code>MIN</code> , <code>IF...THEN</code> function
<i>MIN(x, y)</i>	
Returns:	The least of x and y .

Constraints:	x and y must be compatible types: a pair of integer/real or a pair of string/char values. Complex and boolean values are not permitted.
Examples:	<p><code>MIN(10, 15)</code> returns 10</p> <p><code>MIN(-15, -10)</code> returns -15</p>
Range:	Returns a real number if one of the arguments is real, an integer if both arguments are integer, a string if either argument is a string, or a char if both arguments are chars.
See also:	<code>MAX</code> , <code>IF . . . THEN</code> function
<i>MIX</i> (p , x , y)	
Returns:	x and y weighted (mixed) by probability p . $px + (1 - p)y$.
Constraints:	x and y must be numeric types (integer, real or complex) and p must be integer or real, $0.0 \leq p \leq 1.0$.
Examples:	<p><code>MIX(0.5, 10, -10)</code> returns 0</p> <p><code>MIX(0.25, 1000, 2000)</code> returns 1750</p> <p><code>MIX(0.5, 2+3i, 3+4i)</code> returns 2.5+3.5i</p>
Range:	Returns a real number if x and y are real or integer. Returns a complex number if either x or y are complex.
See also:	<code>PDF STERILE</code>
<i>MODULO</i> (x , y)	
Returns:	The integer remainder of x/y . This is the functional form of the expression $x \text{ MOD } y$.
Constraints:	$y \neq 0$. x and y must be integers
Examples:	<p><code>MODULO(110, 25)</code> returns 10</p> <p><code>MODULO(-124, 25)</code> returns -24</p>
Range:	Returns an integer.
See also:	<code>REMAINDER</code> , <code>IDIV</code>
<i>MONTHDAYS</i> (m , y)	
Returns:	the number of days in month m of year y .
Constraints:	m must be an integer in 1..12 and y must be an integer.
Examples:	<p><code>MONTHDAYS(2, 2000)</code> returns 29</p> <p><code>MONTHDAYS(2, 2001)</code> returns 28</p>
Range:	Returns an integer in the range 28..31.
See also:	<code>LEAPYEAR</code>
<i>MULTIPLY</i> (x , y)	
Returns:	The algebraic product $x \times y$; This is the same as the algebraic expression $x * y$.
Constraints:	x and y must be integer, real or complex arguments.

Examples:	<p><code>MULTIPLY(2, 3)</code> returns 6</p> <p><code>MULTIPLY(2.5, 3)</code> returns 7.5</p> <p><code>MULTIPLY(2.5 + 2i, 3 - 4i)</code> returns 15.5000-4.00000i</p>
Range:	Returns an integer if both argument are integers, real if one argument is real and the other is real or integer, and returns complex if either argument is complex.
See also:	DIVIDE, special function PRODUCT.
<i>NEGATE(x)</i>	
Returns:	-x.
Constraints:	x must be an integer, real or complex argument.
Examples:	<p><code>NEGATE(23)</code> returns -23</p> <p><code>NEGATE(-45)</code> returns 45</p> <p><code>NEGATE(-2 + 6i)</code> returns 2.0-6.0i</p>
Range:	Returns the same type as the argument.
See also:	SUBTRACT, NOTF
<i>NORMAL(x)</i>	
Returns:	The standard normal (Gaussian) probability density function
Constraints:	x must be an integer or real argument.
Examples:	<p><code>NORMAL(0)</code> returns 0.3520653268</p> <p><code>NORMAL(-2)</code> returns 0.0539909665</p>
Range:	Returns a real value from 0.0 to about 0.3521.
Notes:	The function <code>NORMAL(x)</code> corresponds to PDF <code>NORMAL(x) 0, 1 END</code> .
See also:	PDF NORMAL, NORMALCDF, INVNORMAL, STUDENTT, ERF, ERFC
<i>NORMALCDF(x)</i>	
Returns:	The standard normal (Gaussian) cumulative density function
Constraints:	x must be an integer or real argument.
Examples:	<p><code>NORMALCDF(0)</code> returns 0.5</p> <p><code>NORMALCDF(-2)</code> returns 0.0227500620</p>
Range:	Returns a real value from 0.0 to about 1.0.
Notes:	The function <code>NORMALCDF(x)</code> corresponds to PDF <code>NORMAL(-∞, x) 0, 1 END</code> . The function uses a polynomial expansion that is accurate to about six decimal places.
Reference:	Bratley et al. (1983), Abramowitz and Stegun (1972) equation 26.2.17.
See also:	PDF NORMAL, NORMAL, STUDENTT
<i>NOTF(x)</i>	
Returns:	The logical or boolean NOT function.

Constraints:	x must be an integer or boolean argument.
Examples:	<code>NOTF(357)</code> returns -358 <code>NOTF(2x011011)</code> returns -28 <code>NOTF(TRUE)</code> returns FALSE
Range:	Returns an integer result for an integer argument; returns a boolean result for an boolean argument.
Notes:	If x is an integer, <code>NOTF(x)</code> returns the bitwise (logical) NOT of the number. If x is a boolean variable or constant, <code>NOTF(x)</code> returns the boolean NOT of the number.
See also	ORF, ANDF, NEGATE

ORD(c)

Returns:	The ordinal value of character c .
Constraints:	c must be a character argument or a string of length 1.
Examples:	<code>ORD('A')</code> returns 65 <code>ORD('a')</code> returns 97 <code>ORD("@")</code> returns 64
Range	Returns an integer, $0 \leq \text{ORD}(c) \leq 255$.
See also	CHR

ORF(x, y)

Returns:	The logical or boolean or function. This is the functional form of x OR y .
Constraints:	x and y are integers or both boolean values.
Examples:	<code>ORF(456, 123)</code> returns 507 <code>ORF(2x00101, 2x01010)</code> returns 15 <code>ORF(TRUE, FALSE)</code> returns TRUE
Range:	If both x and y are integer variables or constants, <code>ORF(x, y)</code> returns the bitwise (logical) OR of the two numbers. If x and y are boolean variables or constants, <code>ORF(x, y)</code> returns the boolean AND of the two numbers.
See also:	NOTF, ANDF, XORF

PERMUTATIONS(x, y)

Returns:	Permutations: x taken y at a time: $x!/(x-y)!$.
Constraints:	$x \geq y \geq 0$, x and y are integer expressions.
Examples:	<code>PERMUTATIONS(10, 1)</code> returns 10.0 <code>PERMUTATIONS(10, 3)</code> returns 720.0
Range:	Returns a real value.
See also:	COMB

POLARTORECTX(*r*, *a*)

Returns: Rectangular *x* coordinate from polar coordinates *r*, *a*, *rcos(a)*.

Constraints: *r* and *a* are real or integer values.

Examples: *POLARTORECTX*(1/2, *PI*) returns -0.5
POLARTORECTX(1/2, *PI*/3) returns 0.25

Range: Returns a real value.

See also: *POLARTORECTY*, *RECTTOPOLARR*, *RECTTOPOLARA*

POLARTORECTY(*r*, *a*)

Returns: Rectangular *y* coordinate from polar coordinates *r*, *a*: *rsin(a)*.

Constraints: *r* and *a* are real or integer values.

Examples: *POLARTORECTY*(1/2, *PI*) returns 0.0
POLARTORECTY(1/2, *PI*/3) returns 0.4330127019

Range: Returns a real value.

See also: *POLARTORECTY*, *RECTTOPOLARR*, *RECTTOPOLARA*

POWER(*x*, *y*)

Returns: *x* raised to the power *y*, x^y . This is the functional form of the algebraic expression x^y .

Constraints: *x* and *y* are integer, real or complex arguments.

Examples: *POWER*(2, 3) returns 8.0
POWER(3.5, 3.5) returns 80.211780229
POWER(3.5 + 2i, 3.5 - 2i) returns 209.658-306.643i
POWER(-1, 1/2) returns an error; but, *POWER*(-1+0i, 1/2) returns 0.00000+1.00000i.

Range: Returns a real value for real or integer arguments, and a complex value if any argument is complex.

See also: *ROOT*

PUT(*x*)

Returns: the value of *x*, and, as a side-effect, writes the value of *x* to the standard output.

Constraints: *x* is an integer, real, complex, string, char, or boolean.

Examples: *PUT*(2) returns 2, and writes 2 to the standard output.
PUT('c') returns 'c', and writes c to the standard output.

Notes: This function is particularly useful for debugging programs.

See also: *WRITELN*, *WRITE*

RAND

Returns: A random real number from 0 to 1.

Notes: Before `RAND` or other random number functions can be used, the value of `RANDOMSEED` must be set to a positive constant by calling the `SEED` procedure.

Range: Returns a real value from 0 to 1.

See also: `IRAND`, `RRAND`, procedure `SEED`, `RANDOMSEED`.

RE(x)

Returns: The real part of complex argument x

Constraints: x must be an integer, real or complex

Examples: `RE(3 - 4i)` returns 3.0

`RE(3)` returns 3.0

Range: Returns a real number.

See also: `IM`

REAL2STR(x, l, s)

Returns: A string from real expression x . The length of the string is l characters, and s significant digits. See the examples for the way in which s and l are interpreted.

Constraints: x must be a real or integer value. l and s are integer values.

Examples:

If s is positive, at least s significant digits are represented. Whenever possible, the function tries to return the number in decimal format rather than scientific format. A minimum field length l of about 9 is recommended for small or large numbers.

`REAL2STR(PI, 10, 1)` returns "3.14159265"

`REAL2STR(PI, 5, 5)` returns "3.142"

`REAL2STR(1.234567E-8, 20, 2)` returns "0.000000012345670000"

`REAL2STR(1.234567E-8, 9, 8)` returns "1.2E-0008"

`REAL2STR(1.234567E-8, 12, 5)` returns "1.235E-0008"

`REAL2STR(1.2345678, 10, 0)` returns "1"

`REAL2STR(1.2345678E20, 10, 1)` returns "1.2E+0020"

`REAL2STR(1.2345678E20, 10, 2)` returns "1.2E+0020"

`REAL2STR(-1.2345678E-20, 10, 0)` returns "-0"

`REAL2STR(-1.2345678E-20, 10, 0)` returns "-0"

`REAL2STR(-1.2345678E-20, 10, 1)` returns "-1.2E-0020"

If s is negative, s significant digits to the right of the decimal points are returned, with spaces padding on the left to fill all l places. Scientific format will not be forced in the event that the total length is too small to hold the entire number.

`REAL2STR(1.2345678, 10, -5)` returns "1.23457"

```

REAL2STR(1.2345678, 10, -2) returns "      1.23"
REAL2STR(1.2345678, 10, -1) returns "      1.2"
REAL2STR(1.2345678E20, 10, -4) returns
"12345678000000000000.0000"
REAL2STR(1.2345678E20, 10, -3) returns
"12345678000000000000.000"
REAL2STR(1.2345678E20, 10, 0) returns
"12345678000000000000"
REAL2STR(-1.2345678E-20, 10, -4) returns "      -0.0000"
REAL2STR(-1.2345678E-20, 10, -1) returns "      -0.0"

```

If l is negative and s is positive, s significant digits to the right of the decimal points are returned, with spaces padding on the right to fill all l places. Scientific format will not be forced in the event that the total length is too small.

```

REAL2STR(1.2345678, -10, 3) returns "1.235      "
REAL2STR(1.2345678, -10, 9) returns "1.234567800"
REAL2STR(-1.2345678E-20, -10, 1) returns "-0.0      "
REAL2STR(-1.2345678E-20, -10, 9) returns "-0.000000000"
REAL2STR(1.2345678E20, 10, -4) returns
"12345678000000000000.0000"
REAL2STR(1.2345678E20, 10, -1) returns
"12345678000000000000.0"

```

Range: Returns a string of l characters.

See also: BOOL2STR, INT2STR

RECTTOPOLARA(x , y)

Returns: Polar angle (in radians) from rectangular coordinates x and y .

Constraints: x and y must be real or integer values.

Examples: RECTTOPOLARA(5, 6) returns 0.8760580506
RECTTOPOLARA(0, 1) returns 1.5707963268

Range: Returns a real value.

See also: POLARTORECTY, POLARTORECTA, RECTTOPOLARR

RECTTOPOLARR(x , y)

Returns: Polar radius from rectangular coordinates x and y .

Constraints: x and y must be real or integer values.

Examples: RECTTOPOLARR(5, 6) returns 7.8102496759
RECTTOPOLARR(0, 1) returns 1.0

Range: Returns a real value.

See also: POLARTORECTY, POLARTORECTA, RECTTOPOLARA

RECTTOSPHEREA1(x, y, z)

Returns: Spherical angle 1 (in radians) from rectangular coordinates x , y , and z .

Constraints: x , y , and z must be real or integer values.

Examples: `RECTTOSPHEREA1(1, 2, 3)` returns 1.1071487177941
`RECTTOSPHEREA1(1, 1, 1)` returns 0.7853981633974

Range: Returns a real value.

See also: `SPHERETORECTX`, `SPHERETORECTY`, `SPHERETORECTZ`, `RECTTOSPHERER`, `RECTTOSPHEREA2`, `RECTTOPOLARA`, `RECTTOPOLARR`, `POLARTORECTX`, `POLARTORECTY`

RECTTOSPHEREA2(x, y, z)

Returns: Spherical angle 2 (in radians) from rectangular coordinates x , y , and z .

Constraints: x , y , and z must be real or integer values.

Examples: `RECTTOSPHEREA2(1, 2, 3)` returns 0.6405223126794
`RECTTOSPHEREA2(1, 1, 1)` returns 0.9553166181245

Range: Returns a real value.

See also: `SPHERETORECTX`, `SPHERETORECTY`, `SPHERETORECTZ`, `RECTTOSPHERER`, `RECTTOSPHEREA1`, `RECTTOPOLARA`, `RECTTOPOLARR`, `POLARTORECTX`, `POLARTORECTY`

RECTTOSPHERER(x, y, z)

Returns: Spherical radius from rectangular coordinates x , y , and z .

Constraints: x , y , and z must be real or integer values.

Examples: `RECTTOSPHERER(1, 2, 3)` returns 3.7416573867739
`RECTTOSPHERER(1, 1, 1)` returns 1.7320508075689

Range: Returns a real value.

See also: `SPHERETORECTX`, `SPHERETORECTY`, `SPHERETORECTZ`, `RECTTOSPHEREA1`, `RECTTOSPHEREA2`, `RECTTOPOLARA`, `RECTTOPOLARR`, `POLARTORECTX`, `POLARTORECTY`

REMAINDER(x, y)

Returns: The real remainder of x/y . Returns 0 if $y = 0$

Constraints: x and y must be real or integer values.

Examples: `REMAINDER(110.0, 25.0)` returns 10.0
`REMAINDER(-124, 25)` returns -24.0
`REMAINDER(-100, 0)` returns 0.0

Range: Returns a real value.

See also: `MODULO`, `IDIV`

RIGHTSTRING(*s*, *x*)

- Returns: The rightmost substring from *s* of up to *x* characters.
- Constraints: *s* must be a string or char argument, and *x* must be an integer value.
- Examples: `RIGHTSTRING("Probability theory", 6)` returns the string `theory`
- `RIGHTSTRING("Small", 4)` returns the string `mall`
- `RIGHTSTRING("Small", 20)` returns the string `small`
- Range: Returns a string value.
- See also: `LEFTSTRING`, `SUBSTRING`

ROOT(*x*, *y*)

- Returns: The *y*th root of *x*, $x^{1/y}$.
- Constraints: *x* and *y* must be integer, real, or complex arguments.
- Examples: `ROOT(100, 2)` returns `10.0`
- `ROOT(100, -2)` returns `0.10`
- `ROOT(-1, 2)` returns an error; but, `ROOT(-1+0i, 2)` returns `0.00000+1.00000i`
- Range: Returns a real value for real or integer arguments, and a complex value if any argument is complex.
- See also: `POWER`, `SQRT`

ROUND(*x*)

- Returns: *x* rounded and returned as the nearest integer.
- Constraints: *x* is a real or integer argument.
- Examples: `ROUND(1.9)` returns `2`
- `ROUND(2.0)` returns `2`
- `ROUND(1.5)` returns `2`
- `ROUND(-1.5)` returns `-2`
- Range: Returns an integer value.
- See also: `FRAC`, `CEIL`, `INT`, `FLOOR`, `TRUNC`

RRAND(*x*, *y*)

- Returns: A real random number from *x* to *y*.
- Constraints: *x* and *y* are real arguments.
- Examples: `RRAND(5.0, 10.0)` return a value uniformly drawn from 5 to 10.
- `RRAND(10.0, 5.0)` will return a value uniformly drawn from 5 to 10.
- Notes: Before `RRAND` or other random number functions can be used, the value of `RANDOMSEED` must be set to a positive constant.
- Range: Returns an integer from *x* to *y*.

See also:	RAND, IRAND, procedure SEED, RANDOMSEED
<i>RTOD</i> (x)	
Returns:	Radians from degrees, $180x/\pi$.
Constraints:	x is an integer or real expression.
Examples:	<p><code>RTOD(0.5)</code> returns 28.647889756541</p> <p><code>RTOD(PI)</code> returns 180.0</p>
Range:	Returns a real value
See also:	DTOR
<i>SEC</i> (x)	
Returns:	The secant of x .
Constraints:	x is an integer, real or complex argument.
Examples:	<p><code>SEC(0)</code> returns 1.0</p> <p><code>SEC(-oo)</code> returns 0.0</p> <p><code>SEC(-1)</code> returns 1.8508157177</p> <p><code>SEC(1 + 0.5i)</code> returns 1.08127+0.77819i</p>
Range:	Returns a real value for an integer or real argument. Returns a complex value for a complex argument.
See also:	CSC, SECH, SIN, COS
<i>SECH</i> (x)	
Returns:	The hyperbolic secant of x .
Constraints:	x is an integer, real or complex argument.
Examples:	<p><code>SECH(0)</code> returns 1.0</p> <p><code>SECH(-oo)</code> returns 0.0</p> <p><code>SECH(-1)</code> returns 0.6480542737</p> <p><code>SECH(1 + 0.5i)</code> returns 0.62949-0.26190i</p>
Range:	Returns a real value for an integer or real argument. Returns a complex value for a complex argument.
See also:	SEC, CSCH
<i>SETRANSFORM</i> (x)	
Returns:	The standard error of expression x given the current parameter estimates and the variance-covariance matrix.
Constraints:	x is an expression that includes free parameters within a MODEL statement.
Examples:	The following MODEL statement will estimate and then plot the result of a logistic regression with standard errors.

```

MODEL {returns standard errors for a logistic regression}
  DATA
    PDF BERNOULLITRIAL(outcome)
    PARAM p LOW = -100 HIGH = 100 START = 0 FORM = LOGISTIC
    COVAR age PARAM b_age LOW = -100 HIGH = 100 START = 0 END
    END {param}
  END {bernoullitrial}
END {data}
RUN
FULL THEN
  PLOT
    CURVE
      x = 1 TO 50 x, LOGISTIC(p + x * b_age)
    END {curve}
    CURVE WITH "errorbars"
      x = 1 TO 50 x, LOGISTIC(p + x * b_age),
        SETTRANSFORM(LOGISTIC(p + x * b_age))
    END {curve}
  END {plot}
END {full then}
END {model}

```

Range: Returns a real value.

Notes: This function must be called within a MODEL...END statement. Furthermore, the variance-covariance matrix must be computed and cannot be singular. The expression x must be a full expression that includes at least one of the free model parameters (or else the standard error will be zero).

This function works by first computing (numerically) an array of partial derivatives of the expression with respect of each of the N free parameters in the model, $\mathbf{d} = (\delta x / \delta p_1, \delta x / \delta p_2, \dots, \delta x / \delta p_N)$. Next, the standard error is computed as $\sqrt{\mathbf{d}' \mathbf{V} \mathbf{d}}$, where \mathbf{V} is the variance-covariance matrix.

See also: Statements MODEL and PLOT, procedure PTRANSFORM

SGN(x)

Returns: 1 if $x > 0$, 0 if $x = 0$, or -1, if $x < 0$ for integer and real arguments.

Returns the complex $\text{sgn}(x)$ function $\frac{a}{\sqrt{a^2 + b^2}} + \frac{b}{\sqrt{a^2 + b^2}}i$ for complex argument $x = a + bi$

Constraints: x is an integer, real, or complex argument.

Examples:

```

SGN(3) returns 1
SGN(0) returns 0
SGN(-2) returns -1
SGN(-3i) returns 0.00000-1.00000i
SGN(3i) returns 0.00000+1.00000i
SGN(-2 + 3i) returns -0.5547+0.83205i

```

Range: Returns an integer from -1 to 1 for integer and real values. Returns a complex value from $-1 \leq a \leq 1$, $-1 \leq b \leq 1$ for complex argument $x = a + bi$

See also:	SIGN, DELTA, HEAVISIDE
<i>SHIFTLEFT</i> (x, y)	
Returns:	Shifts bits of x to the left by y binary positions.
Constraints:	x and y are integers
Examples:	<code>SHIFTLEFT(1, 5)</code> returns 32 <code>SHIFTLEFT(2, 10)</code> returns 2048
Range:	Returns an integer value.
See also:	SHIFTRIGHT
<i>SHIFTRIGHT</i> (x, y)	
Returns:	Shifts bits of x to the right by y binary positions.
Constraints:	x and y are integers
Examples:	<code>SHIFTRIGHT(32, 5)</code> returns 1 <code>SHIFTRIGHT(2048, 10)</code> returns 2
Range:	Returns an integer value.
See also:	SHIFTLEFT
<i>SIGN</i> (x, y)	
Returns:	x with the same sign as y .
Constraints:	x and y are an integer or real arguments.
Examples:	<code>SIGN(-234, 1)</code> returns 234 <code>SIGN(234, 0)</code> returns 234 <code>SIGN(-234, 0)</code> returns -234 <code>SIGN(234, -1)</code> returns -234
Range:	Returns a value that is the same type as x .
See also:	SGN, NEGATE
<i>SIN</i> (x)	
Returns:	The sine of angle (in radians) x
Constraints:	x is an integer, real or complex argument.
Examples:	<code>SIN(PI)</code> returns 0.0 <code>SIN(PI/2)</code> returns 1.0 <code>SIN(PI/2+0i)</code> returns 1.0+0i
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.
See also:	COS, TAN, ARCSIN
<i>SINH</i> (x)	
Returns:	The hyperbolic sine of x .

Constraints: x is an integer, real or complex argument.

Examples: `SINH(1)` returns 1.1752011936438
`SINH(0)` returns 0.0
`SINH(-1)` returns -1.175201193644

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: `COSH`, `SIN`, `ARCSINH`

SPHERETORECTX($r, a1, a2$)

Returns: Rectangular x from 3 spherical coordinates, radius r , angles (in radians) $a1$ and $a2$.

Constraints: r , $a1$, and $a2$ are real or integer values.

Examples: `SPHERETORECTX(1, PI, PI)` returns 0
`SPHERETORECTX(1, PI/4, PI/4)` returns 0.5

Range: Returns a real value.

See also: `SPHERETORECTY`, `SPHERETORECTZ`, `RECTTOSPHERER`, `RECTTOSPHEREA1`, `RECTTOSPHEREA1`, `RECTTOPOLARA`, `RECTTOPOLARR`, `POLARTORECTX`, `POLARTORECTY`

SPHERETORECTY($r, a1, a2$)

Returns: Rectangular y from 3 spherical coordinates, radius r , angles (in radians) $a1$ and $a2$.

Constraints: r , $a1$, and $a2$ are real or integer values.

Examples: `SPHERETORECTY(1, PI, PI)` returns 0
`SPHERETORECTY(1, PI/4, PI/4)` returns 0.5

Range: Returns a real value.

See also: `SPHERETORECTX`, `SPHERETORECTZ`, `RECTTOSPHERER`, `RECTTOSPHEREA1`, `RECTTOSPHEREA1`, `RECTTOPOLARA`, `RECTTOPOLARR`, `POLARTORECTX`, `POLARTORECTY`

SPHERETORECTZ($r, a1, a2$)

Returns: Rectangular z from 3 spherical coordinates, radius r , angles (in radians) $a1$ and $a2$.

Constraints: r , $a1$, and $a2$ are real or integer values.

Examples: `SPHERETORECTZ(1, PI, PI)` returns -1
`SPHERETORECTZ(1, PI/4, PI/4)` returns 0.7071067811865

Range: Returns a real value.

See also: `SPHERETORECTX`, `SPHERETORECTY`, `RECTTOSPHERER`, `RECTTOSPHEREA1`, `RECTTOSPHEREA1`, `RECTTOPOLARA`, `RECTTOPOLARR`, `POLARTORECTX`, `POLARTORECTY`

SQR(x)

Returns: x squared, x^2

Constraints:	x is an integer, real, or complex value.
Examples:	<p><code>SQR(PI)</code> returns 9.8696044010894</p> <p><code>SQR(10)</code> returns 100.0</p> <p><code>SQR(-5)</code> returns 25.0</p> <p><code>SQR(-5 + 3i)</code> returns 16.0000-30.0000i</p>
Range:	Returns a real value for real or integer x and complex for complex x .
See also:	POWER
<i>SQRT</i> (x)	
Returns:	The square root of x , \sqrt{x}
Constraints:	$x \geq 0$.
Examples:	<p><code>SQRT(100)</code> returns 10.0</p> <p><code>SQRT(25)</code> returns 5.0</p>
Range:	Returns a real value for real or integer x and complex for complex x .
See also:	ROOT
<i>STANDARDIZE</i> (x , μ , σ)	
Returns:	A value for x standardized by location parameter μ and scale parameter σ , as $(x - \mu)/\sigma$.
Constraints:	$\sigma \geq 0$, all arguments are real or integer.
Examples:	<p><code>STANDARDIZE(1, 2, 1)</code> returns -1.0</p> <p><code>STANDARDIZE(1, 1, 1)</code> returns 0.0</p> <p><code>STANDARDIZE(2, 1, 1)</code> returns 1.0</p>
Range:	Returns a real value.
<i>STRINGLEN</i> (s)	
Returns:	The length of string s
Constraints:	s is a string or char argument.
Examples:	<p><code>STRINGLEN("1234")</code> returns the integer 4</p> <p><code>STRINGLEN("abcdefghij")</code> returns the integer 10</p>
Range:	Returns a non-negative integer value.
<i>STRING2INT</i> (s)	
Returns:	An integer number from string s
Constraints:	s is a string or char argument.
Examples:	<p><code>STRING2INT("-124")</code> returns the integer -124</p> <p><code>STRING2INT("0xMCMXIV")</code> returns the integer 1914</p> <p><code>STRING2INT("xyz")</code> results in an error</p>
Range:	Returns an integer value.

See also:	STRING2REAL, TOBASE
<i>STRING2REAL(s)</i>	
Returns:	A real number from string <i>s</i>
Constraints:	<i>s</i> is a string or char argument.
Examples:	<p>STRING2REAL("-124.73") returns the real value -124.73</p> <p>STRING2REAL("0xMCMXIV") returns the real value 1914.0</p>
Range:	Returns a real value.
See also:	STRING2INT, TOBASE
<i>STUDENTT(x, df)</i>	
Returns:	The upper tail area (beyond <i>x</i>) of a Student's <i>t</i> distribution with <i>df</i> degrees of freedom.
Constraints:	$x \geq 0$, $df > 0$
Examples:	<p>STUDENTT(3.078, 1) returns 0.0999903817</p> <p>STUDENTT(2.750, 30) returns 0.0049999472</p>
Range:	Returns a real value between 0.0 and 1.0.
See also:	FDIST, CHISQ, INVSTUDENTT
<i>SUBSTRING(s, x, y)</i>	
Returns:	A substring from string <i>s</i> , beginning in position <i>x</i> and length <i>y</i> .
Constraints:	<i>s</i> is a string, and <i>x</i> and <i>y</i> are integers.
Examples:	<p>SUBSTRING("Probability theory", 5, 7) returns the string ability</p> <p>SUBSTRING("Small", 10, 5) returns the empty string ("")</p> <p>SUBSTRING("Small", 2, 20) returns the string mall</p>
Range:	Returns a string. An empty string is returned if $x \leq 0$, $y \leq 0$ or if <i>x</i> is an empty string.
See also:	LEFTSTRING, RIGHTSTRING
<i>SUBTRACT(x, y)</i>	
Returns:	$x - y$. This is the functional form of the algebraic $x - y$.
Constraints:	<i>x</i> and <i>y</i> are integer, real or complex arguments.
Examples:	<p>SUBTRACT(10, 2) returns 8</p> <p>SUBTRACT(-10, 2) returns -12</p> <p>SUBTRACT(-10+4i, 2-3i) returns -12.000+7.000000i</p>
Range:	Returns an integer if both arguments are integers. A real value is returned if at least one argument is real and the other is real or integer. A complex number is returned if either argument is complex.
See also:	ADD

TAN(*x*)

Returns: The tangent of angle (in radians) *x*.

Constraints: *x* is an integer, real or complex argument.

Examples: `TAN(DTOR(45))` returns 1.0
`TAN(0)` returns 0
`TAN(pi/2)` returns +oo
`TAN(1/2 + 1i)` returns 0.19558+0.84297i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

See also: SIN, COS

TANH(*x*)

Returns: The hyperbolic tangent of *x*.

Constraints: *x* is an integer, real or complex argument.

Examples: `TANH(0)` returns 0
`TANH(1)` returns 0.7615941559558
`TANH(1 + 0.5i)` returns 0.84297+0.19558i

Range: Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

TOBASE(*x*, *b*)

Returns: A string with integer *x* in base *b*.

Constraints: *x* is an integer, *b* can be any base from 2 to 36

Examples: `TOBASE(39, 2)` returns the string 100111
`TOBASE(47, 16)` returns the string 2F

Range: Returns a string value.

See also: Number formats

TOLOWER(*s*)

Returns: A string in lower case.

Constraints: *s* is a string or char argument

Examples: `TOLOWER("A STRING")` returns the string a string
`TOLOWER('A' + 'b')` returns the string ab

Range: Returns a string value that is the same length as *s*.

See also: TOUPPER

TOUPPER(*s*)

Returns: A string in upper case.

Constraints: *s* is a string or char argument

Examples: `TOUPPER("a string")` returns the string A STRING

`TOUPPER('A' + 'b')` returns the string `AB`

Range: Returns a string value that is the same length as *s*.

See also: `TOLOWER`

TRIM(s)

Returns: A string with leading and trailing spaces removed.

Constraints: *s* is a string or char argument

Examples: `TRIM(" a string ")` returns "a string"

`TRIM(" ")` returns "" (empty string)

Range: Returns a string value that is not longer than *s*.

See also: `TRIML`, `TRIMR`

TRIML(x)

Returns: A string with leading spaces removed.

Constraints: *s* is a string or char argument

Examples: `TRIML(" a string ")` returns the string "a string "

`TRIML(" ")` returns the empty string (" ")

Range: Returns a string value that is not longer than *s*.

See also: `TRIM`, `TRIMR`

TRIMR(x)

Returns: A string with trailing spaces removed.

Constraints: *s* is a string or char argument

Examples: `TRIMR(" a string ")` returns the string " a string"

`TRIMR(" ")` returns the empty string (" ")

Range: Returns a string value that is not longer than *s*.

See also: `TRIM`, `TRIML`

TRUNC(x)

Returns: *x* truncated to an integer.

Constraints: *x* is a real or integer value

Examples: `TRUNC(1.0)` returns 1

`TRUNC(2.8)` returns 2

`TRUNC(-2.5)` returns -2

Range: Returns an integer value.

See also: `ROUND`, `FRAC`, `CEIL`, `INT`, `FLOOR`

WEEKDAY(x)

Returns: A numerical day of the week (Sun=1 Mon=2...) for Julian day *x*.

Constraints: *x* is an integer value

Examples:	<p>WEEKDAY(DMYTOJ(01, 01, 2000)) returns 7 (Saturday)</p> <p>WEEKDAY(DMYTOJ(15, 01, 2001)), M.L. King Jr.'s birthday, returns 2 (Monday).</p>
Range:	Returns an integer value from 1 to 7.
See also:	See also: JULIAND, JULIANM, JULIANY, YEARDAY, DMYTOJ
<i>XORF</i> (<i>x</i> , <i>y</i>)	
Returns:	Logical or boolean XOR function. This is the functional from of <i>x</i> XOR <i>y</i> .
Constraints:	<i>x</i> and <i>y</i> are integers or both boolean values.
Examples:	<p>XORF(TRUE, TRUE) returns FALSE</p> <p>XORF(FALSE, FALSE) returns FALSE</p> <p>XORF(TRUE, FALSE) returns TRUE</p> <p>XORF(2x010101, 2x000111) returns 18 (2x010010)</p>
Range:	If both <i>x</i> and <i>y</i> are integer variables or constants, ORF(<i>x</i> , <i>y</i>) returns the bitwise (logical) OR of the two numbers. If <i>x</i> and <i>y</i> are boolean variables or constants, ORF(<i>x</i> , <i>y</i>) returns the boolean AND of the two numbers.
See also	ORF, NOTF, ANDF
<i>YEARDAY</i> (<i>x</i>)	
Returns:	Day of the year (1-jan = 1...) for Julian day <i>x</i> .
Constraints:	<i>x</i> is an integer
Examples:	<p>YEARDAY(DMYTOJ(01, 01, 2000)) returns 1</p> <p>YEARDAY(DMYTOJ(30, 06, 2000)) returns 182</p>
See also:	See also: JULIAND, JULIANM, JULIANY, WEEKDAY, DMYTOJ
Range:	Returns an integer value from 1 to 366 for valid arguments.
Bugs:	The function crashes for some very old dates (B.C.)
<i>ZETA</i> (<i>x</i>)	
Returns:	The Riemann's Zeta function, $\zeta(x) = \sum_{k=1}^{\infty} \frac{1}{k^x}$, for real and complex arguments.
Constraints:	<i>x</i> is an integer, real, or complex value
Examples:	<p>ZETA(0) returns -0.50000</p> <p>ZETA(1) returns ∞</p> <p>ZETA(0.5) returns -1.460354509</p> <p>ZETA(1 + 0.5i) returns 0.57843-1.96355i</p>

Notes:	The function is evaluated using the second algorithm described by Borwein (1995). The algorithm uses a 30 term polynomial expansion that provides about 39 digits of precision.
Range:	Returns a real value for integer or real arguments. Returns a complex value for complex arguments.

Chapter 6

Special functions

This chapter documents the special functions available in *mle*. Special functions are called such because their syntax is more complicated than that of simple functions. Special functions also have optional arguments.

Special Functions Reference

DATA

The `DATA` function is used to loop through data variables. Usually the `DATA...END` function "feeds" observations, one at a time, to the likelihood, corresponding to the product (\prod) over all observations shown in likelihoods (or the Σ shown in loglikelihoods).

Note that the `DATA statement` is *not* the same as the `DATA function`. The two are used together. The `DATA statement` is used to read data in from a file, whereas the `DATA function` loops through the data observations, and tallies a sum or product.

The syntax of the `DATA` function is

```
DATA <optional_form>
  <expression>
END
```

The effect of the `DATA` statement is to loop through each observation read in by the `DATA` statement, and form a sum or product of the `<expression>`. When used in a likelihood model, the `DATA...END` function returns the *total logloglikelihood* or *total likelihood*, given a series of *observations* and an expression for an *individual likelihood* or *individual loglikelihood*.

`<expression>` is an integer or real expression that uses a data variable. A result is returned from `<expression>` for each data observation. The results over the entire data set are summed or taken as a product (see `<optional_form>`).

`<optional_form>` determines how the results are "tallied."

FORM = SUMLL: This takes the log after evaluating each *<expression>* and cumulatively sums over all observations. When used within a likelihood model, a likelihood (rather than a loglikelihood) is specified for *<expression>*. This is the default value if no *<formtype>* is specified. The DATA function using this form is equivalent to the function

```
SUMMATION D_IDX (1, N_OBS)
  LOG(<expression>)
END
```

The predefined variable D_IDX serves as a global index for data variables, and N_OBS is the count of observations read by the DATA statement.

FORM = SUM or FORM = SUMMATION: Cumulates sum after evaluating each *<expression>* over all observations. The natural log of *<expression>* is not taken. Hence, when used in likelihood models, a loglikelihood rather than a likelihood is specified for *<expression>*. The DATA function using this form is equivalent to the function

```
SUMMATION D_IDX (1, N_OBS)
  <expression>
END
```

FORM = PROD or FORM = PRODUCT: Takes the product after evaluating *<expression>* over all observations. The natural log of *<expression>* is *not* taken. Hence, when used in a likelihood model, a likelihood (rather than a loglikelihood) is specified for *<expression>*. The DATA function using this form is equivalent to the function

```
PRODUCT D_IDX (1, N_OBS)
  <expression>
END
```

See also: Additional details and examples can be found in the **Model** chapter of the *User's manual*. The LEVEL and LEVELDELTA special functions work with the DATA function to create nested likelihoods. Related special functions are PRODUCT and SUMMATION.

DERIVATIVE

The DERIVATIVE function computes the numerical value of a function's derivative at a particular point. The result is returned as a REAL number. Formats for the statement are:

```
DERIVATIVE <variable> = <expr1>, <expr2> END
DERIVATIVE <variable> = <expr1>, <expr2>, <expr3> END
DERIVATIVE (<expr4>) <variable> = <expr1>, <expr2> END
DERIVATIVE (<expr4>) <variable> = <expr1>, <expr2>, <expr3> END
```

<variable> is the variable of differentiation.
<expr1> is the point at which the derivative is evaluated.
<expr2> is the function that will be differentiated.

<expr3>

is an optional argument that is the largest value of dx to begin with. If <expr3> is not given, an initial value for dx of 0.001 is used, which is reasonable for a wide range of functions. (This default value can be changed by changing the value of the variable DIFF_DX). In finding the derivative, successively smaller values of dx are used until reasonable precision is reached.

For example, DERIVATIVE $x = 1$, SIN(x) END computes the value of the derivative at $x = 1$ for the function $\sin(x)$; it returns -0.841470984808. This derivative requires about 4 function evaluations with an initial $dx = 0.001$. If dx is changed to 0.1, as in DERIVATIVE $x = 1$, SIN(x), 0.1 END, the same answer is found after 14 function evaluations. Likewise, if dx is set to 1E-7, 6 function evaluations are required.

<expr4>

is an optional argument, enclosed within parentheses, that specifies the degree of differentiation. If this value is not given or is ≤ 1 , a first degree derivative will be found [i.e. $f'(x)$]. Higher degree derivatives are found if the fourth expression is specified and are greater than one. For example, SIN(x)' evaluated at $x = 1$ is equal to -SIN(1) which is about -0.841470984808. The expression DERIVATIVE (2) $x = 1$, SIN(x) END returns the same numerical result.

High order derivatives tend to lose numerical precision. This can be seen in the following series, which should all evaluate to 24.0:

```
DERIVATIVE (1) x = 1, 24*x    END returns 24.000000000000
DERIVATIVE (2) x = 1, 12*x^2  END returns 23.999999999998
DERIVATIVE (3) x = 1, 4*x^3   END returns 24.000000000455
DERIVATIVE (4) x = 1, x^4     END returns 24.000026132114
```

Reference:

The derivative function finds numerical estimates at a point using an adaptive algorithm similar to that described by Ridders (1982).

FINDMIN

The FINDMIN function iteratively finds the argument value that minimizes a bounded function, and returns the result as a REAL number. The function can be used to maximize a function by simply negating the expression to be maximized (<expr>). Formats are:

```
FINDMIN <variable> (<exp1>, <exp2>) <expr> END
FINDMIN <variable> (<exp1>, <exp2>, <exp3>) <expr> END
FINDMIN <variable> (<exp1>, <exp2>, <exp3>, <exp4>) <expr> END
FINDMIN <variable> (<exp1>, <exp2>, <exp3>, <exp4>, <exp5>) <expr> END
```

<variable>

is the argument that is changed to find the function minimum. This variable is required.

<expr>

is the function for which the minimum is to be found. This expression is required.

<exp1>

is the minimum boundary of the function. This expression is required.

<code><expr2></code>	is the maximum boundary of the function. This expression is required.
<code><expr3></code>	is a starting value of <code><variable></code> to try. This expression is optional. The midpoint between <code><expr1></code> and <code><expr2></code> will be used if <code><expr3></code> is not specified.
<code><expr4></code>	is an optional desired precision of the solution. This expression is optional. If the forth expression is not given, its value will taken from the global variable <code>FIND_EPS</code> .
<code><expr5></code>	is an optional maximum number of iterations allowed in finding the solution. If <code><expr5></code> is not given, the value will be taken from the global variable <code>FIND_ITERS</code> .
Example:	<p><code>FINDMIN x (0, 2*PI) COS(x) END</code> returns 3.1415925395570 (π is an exact solution).</p> <p>A more precise solution (but one that takes longer to find) is found by <code>FINDMIN x (0, 2*PI, 1, 1E-15) COS(x) END</code> which returns 3.1415926535713.</p>
Reference:	The <code>FINDMIN</code> function uses a one-dimensional bounded method by Brent (1973), and algorithms derived from Press et al. (1989).

FINDZERO

The `FINDZERO` function iteratively finds the argument for which a bounded function is zero. Formats are:

```
FINDZERO <variable> (<expr1>, <expr2>) <expr> END
FINDZERO <variable> (<expr1>, <expr2>, <expr3>) <expr> END
FINDZERO <variable> (<expr1>, <expr2>, <expr3>, <expr4>) <expr> END
```

<code><variable></code>	is the argument that is changed to find the zero value of the function.
<code><expr></code>	is the function to find zero for.
<code><expr1></code>	is the minimum boundary of the function.
<code><expr2></code>	is the maximum boundary of the function.
<code><expr3></code>	is an optional precision of the solution. If the third expression is not given, its value will taken from the global variable <code>FIND_EPS</code> .
<code><expr4></code>	is an optional maximum number of iterations allowed in finding the solution. If the forth expression is not given, the value will be taken from the variable <code>FIND_ITERS</code> .
Example:	<p><code>FINDZERO x (0, PI) COS(x) END</code> returns 1.5707963267949, which is the correct value of $\pi/2$.</p>

`FINDZERO` works for well-behaved functions that have a single continuous zero within the bounds. For example, `cos(x)` goes to zero for four different x values in the interval $[0, 4\pi]$. `FINDZERO x (0, 4*PI) COS(x) END` returns 10.995574287564 (and may return other solutions depending on hardware and some variable values).

Another pathological example is functions with no zero in the specified interval. For example, $\cos(x)$ has no value of zero in the interval $[2\pi/3, 4\pi/3]$. `FINDZERO x (2*PI/3, 4*PI/3) COS(x) END` returns the value $2\pi/3$; this is the closest value to zero found.

Reference: `FINDZERO` uses an algorithm developed by Brent (1973) to find the zero of a function. The algorithm will always find a zero of a function provided `<expr1>` and `<expr2>` encloses one zero. The algorithm is modified from Press et al. (1989).

IF ... THEN

The `IF...THEN...ELSE...END` function returns a value based on one or more conditions. The simplest format for the `IF` function is

```
IF <boolean_expr> THEN
  <expr1>
ELSE
  <expr2>
END
```

`<boolean_expr>` returns a boolean value (TRUE or FALSE). If the boolean function evaluates to TRUE then the first expression `<expr1>` will be evaluated, and returned as the value of the `IF` function. If the boolean function returns FALSE, the second expression `<expr2>` will be evaluated and returned as the result of the `IF` function.

The conditionally selected expression may be another `IF` function, so that multiple `IF ... THEN ... ELSE` functions may be nested:

```
IF <boolean_expr1> THEN
  <expr1>
ELSE
  IF <boolean_expr2> THEN
    <expr2>
  ELSE
    IF <boolean_expr3> THEN
      <expr3>
    ELSE
      <expr4>
    END      {3rd if...else}
  END      {2nd if...else}
END      {1st if...else}
```

An alternative to this is to use a series of `ELSEIF <boolean expression> THEN <expression>` with the basic `IF` function. The above example can be written

```
IF <boolean_expr1> THEN
  <expr1>
ELSEIF <boolean_expr2> THEN
  <expr2>
ELSEIF <boolean_expr3> THEN
  <expr3>
ELSE
  <expr4>
END
```

Examples: Here are some examples of `IF` functions used in assignment statements:

```

ind = IF (a^2 > 12) OR (a = 0) THEN a^2 ELSE 0 END

message = IF not valid THEN
    "The result is not valid"
ELSE
    "The result is valid"
END {if}

status = IF height < 48 THEN
    -1
ELSEIF (height >= 48) and (height <= 60) THEN
    0
ELSE
    1
END

```

Notes:

The *IF function* should not be confused with the *IF statement*, even though they look very similar. In the above example, the *IF function* returns a value as the right-hand side of an assignment statement. An *IF function* always returns a value, and can be used in the same context as, for example, a *SIN()* or an *ISEVEN()* function. The *IF statement* does not return a value. Rather, it controls whether or not one or more sets of statements will be executed. An *IF statement* can only be used in contexts where another statement can be used.

INTEGRATE

The *INTEGRATE* function does numerical integration in one dimension. The integration method can be changed, and the user is given control over the precision of the solution with some methods. Typically, *INTEGRATE* is used to integrate a likelihood over some distribution of unmeasured heterogeneity or to renormalize an improper density function. Two formats of *INTEGRATE* are

```

INTEGRATE <variable> ( <lower_limit> , <upper_limit> )
    <expression>
END

```

and

```

INTEGRATE <variable_name> ( <lower_limit> , <upper_limit>, <tolerance> )
    <expression>
END

```

<variable_name> is the name of a variable of integration (type *REAL*), which can be referenced within the *expression*.

<expression> is the integrand, and can be any legal real or integer expression (including, perhaps, more *INTEGRATE* functions).

<lower_limit> is the lower limit of integration. This expression is evaluated once, just prior to integration, and must return an integer or real result. The resulting value is then constant during the integration operations.

<upper_limit> is the upper limit of integration. This expression is evaluated once, just prior to integration, and must return an integer or real result. The resulting value is then constant during the integration operations.

<tolerance> is an optional argument that defines the precision to which the solution will be found. It must be a real expression.

Example:

Consider a model in which observed exact times (t) to failure are distributed according to a Weibull PDF, $f(t)$. In addition we model a distribution of unmeasured heterogeneity, $g(z) \sim N(0, \sigma_z^2)$. Assume that the effect of z on $f(t)$ is loglinear on a , so that the first parameter of the Gompertz distribution is $a' = ae^z$. The likelihood is:

$$L = \prod_{i=1}^N \int_{-25}^{25} g(z | 0, \sigma_z) f(t_i | ae^z, b) dz$$

In this model, we would like to estimate the parameters a , b , and σ_z from a series of observations. The *mle* MODEL statement for this model is

```
MODEL
  DATA
    INTEGRATE z (-25, 25)
      PDF NORMAL(z)      {g(z): heterogeneity}
      0, PARAM s LOW = 0.0001 HIGH = 5 START = 1 END
    END {pdf normal}
    *
      PDF GOMPERTZ(t)     {f(t): distribution of failures}
      PARAM a LOW=0.0 HIGH=0.9 START=0.073 FORM=LOGLIN
      COVAR z 1
      END {param a}
      PARAM b LOW = 0.0001 HIGH = 1.4 START = 0.5 END
      END {pdf Gompertz}
    END {integrate}
  END {data}
RUN
  FULL
END {model}
```

Notes:

The INTEGRATE function is particularly useful for

- (1) estimating distributions of unmeasured heterogeneity as shown in the example
- (2) estimating multilevel models for which one can integrate out the non-independence of observations
- (3) dealing with left or left-interval censoring as is described in the Examples chapter
- (4) computing survivorship in custom likelihood when a closed form is not available for the PDF
- (5) Numerical evaluation of the mean, variance, or other moments of a function.

It should be kept in mind that numerical integration is difficult and time consuming—particularly for nested integrals. One useful trick for successful integration is to set the limits of integration as narrow as possible without “shutting out” non-zero areas of the function to be integrated. In the above example, the limits of integration were set to ± 25 , because even with $s = 5$, the area under the distribution outside these limits is ignorable. An even better strategy is to use the PREASSIGN function to define the parameter s , and then compute an appropriate upper and lower limit for the integration:

```

MODEL
  DATA
    PREASSIGN
      BEGIN
        sigma = PARAM s LOW = 0.0001 HIGH = 5 START = 1 END
        upper_lim = 5*sigma
        lower_lim = -upper_lim
      END
      INTEGRATE z (upper_lim, lower_lim)
        PDF NORMAL(z) 0, sigma END
        * PDF GOMPERTZ(t)
          PARAM a LOW=0.0 HIGH=0.9 START=0.073 FORM=LOGLIN
          COVAR z 1
          END {param a}
          PARAM b LOW = 0.0001 HIGH = 1.4 START = 0.5 END
          END {pdf Gompertz}
        END {integrate}
      END {preassign}
    END {data}
  RUN
  FULL
END {model}

```

mle currently provides four different methods for performing numerical integration. Each method has its strengths and weaknesses.

Adaptive quadrature. This is the default method, and it can be defined by setting `INTEGRATE_METHOD = I_AQUAD`. The method is an eight point adaptive quadrature integration routine adapted from the routine QUANC8 (Forsythe et al. 1977). The method will recursively integrate the function until a specified precision is reached. Precision is defined by changing the `INTEGRATE_TOL` constant. By default `INTEGRATE_TOL = 0.000001`. This method works well for relatively smooth functions, and is probably the best general integration routine incorporated into *mle*.

Simpson. This method uses Simpson's rule to evaluate integrals to a predefined tolerance, and is defined by setting `INTEGRATE_METHOD = I_SIMPSON`. The method is adapted from the routine QSIMP (Press et al. 1986). The function will be integrated until a predefined precision is reached or a maximum number of iterations are reached. Precision is defined by changing the `INTEGRATE_TOL` constant. By default `INTEGRATE_TOL = 0.000001`. The maximum number of iterations is set with the constant `INTEGRATE_N` and is 100 by default. This method is useful for smooth functions.

Closed trapazoidal. This method uses a brute force extended trapazoidal function to evaluate integrals. The function to be integrated must be evaluable at the limits of integration; otherwise the opened trapazoidal should be used. The method is defined by setting `INTEGRATE_METHOD = I_TRAP_CLOSED`. The extended trapazoidal rule will be evaluated at a predefined number of equally spaced points defined by `INTEGRATE_N` (the default is 100). The minimum allowable steps is eight. The method does not provide an error tolerance. This brute-force method is useful for functions that are not smooth enough for adaptive quadrature or Simpson.

I_TRAP_CLOSE uses a closed extended Simpson's rule, so that the endpoints are evaluated. The formula is

$$\int_a^b f(t)dt \approx \frac{b-a}{N} \left\{ \frac{17}{48}[f(t_1) + f(t_N)] - \frac{59}{48}[f(t_2) + f(t_{N-1})] \right. \\ \left. + \frac{43}{48}[f(t_3) + f(t_{N-2})] + \frac{49}{48}[f(t_4) + f(t_{N-3})] + \sum_{i=5}^{N-4} f(t_i) \right\}$$

where N is the number of subdivisions (INTEGRATE_N), and $t_i = a + (i-1)(b-a)/N$. The error is on the order N^{-4} .

Open trapezoidal. This method is similar to the closed trapezoidal method, except that the function to be integrated is never evaluated at the limits of integration. The method is defined by setting INTEGRATE_METHOD = I_TRAP_OPENED. I_TRAP_OPEN uses an open extended Simpson's rule, so that the endpoints are never evaluated. The formula is

$$\int_a^b f(t)dt \approx \frac{b-a}{N} \left\{ \frac{109}{48}[f(t_2) + f(t_{N-1})] - \frac{5}{48}[f(t_3) + f(t_{N-2})] \right. \\ \left. + \frac{63}{48}[f(t_4) + f(t_{N-3})] + \frac{49}{48}[f(t_5) + f(t_{N-4})] + \sum_{i=6}^{N-5} f(t_i) \right\}$$

where N is the number of subdivisions (INTEGRATE_N), and $t_i = a + (i-1)(b-a)/N$. The error is on the order N^{-4} .

LEVEL

The LEVEL function works in conjunction with the DATA function. It provides a mechanism by which multilevel or hierarchical models can be constructed. The syntax of the LEVEL function is

```
LEVEL <boolean_expr> THEN <optional_form>
  <expression>
END
```

The effect of the level statement is to test <boolean_expr> for each observation and, while the condition is true, form a product of likelihoods out of the observations.

<boolean_expr> is an expression that returns true or false for each observation in the data set. If the result is true, <expression> is evaluated and multiplied by the current product (or summed—see <optional form>). If the result is false, the function exits and returns the current product or sum.

<expression> is an integer or real expression that uses a data variable. A result is returned from <expression> for each data observation. The results

over the entire data are taken as a product (or summed—see *<optional form>*) while *<boolean_expr>* is true.

<optional_form> are described for the DATA function on page 85. The default form for the LEVEL function is PRODUCT. Hence, by default a likelihood is being formed (not a log-likelihood, which is the default for the DATA function).

See also: More information on the LEVEL statement can be found in the **Model** chapter of the *User's manual*. The DATA and LEVELDELTA special functions are closely related functions used to create nested likelihoods. Other related special functions are PRODUCT and SUMMATION.

LEVELDELTA

The LEVELDELTA function works in conjunction with the DATA function. It provides a mechanism by which multilevel or hierarchical models can be constructed. The LEVELDELTA function is very similar to the LEVEL function. The syntax of the LEVELDELTA function is

```
LEVELDELTA <expr1> THEN <optional_form>
  <expression>
END
```

The effect of the LEVELDELTA function is to evaluate *<expression>* for each observation and, while the expression does not change, form a sum or product of likelihoods out of the observations.

<expr1> is a numeric expression. As the LEVELDELTA loops through the data set, a series of observations that return the same value for this expression are treated as belonging to the same level. If so, *<expression>* is evaluated and multiplied by the current product (or summed—see *<optional form>*). If not, the function exits and returns the current product or sum. LEVELDELTA can be used, for example, to cumulate a likelihood (or loglikelihood) for an individual with repeated observations, groups of individuals that are part of the same cluster, etc. *<expr1>* is frequently an individual's ID number or a group identifier.

<expression> is an integer or real expression that uses a data variable. A result is returned from *<expression>* for each data observation. The results over the entire data are taken as a product (or summed—see *<optional form>*) while *<expr>* evaluates to the same value.

<optional_form> are described for the DATA function on page 85. The default form for the LEVELDELTA function is PRODUCT. Hence, by default a likelihood is being formed (not a log-likelihood, which is the default for the DATA function).

See also: More information on the LEVELDELTA statement can be found in the **Model** chapter of the *User's manual*. The DATA and LEVEL special functions are closely related functions used to create nested

likelihoods. Other related special functions are `PRODUCT` and `SUMMATION`.

PARAM

mle has a general method for defining all parameters to be used in a likelihood model.¹ The `PARAM` function defines a parameter and its characteristics. When models are “solved”, *free parameters* are estimated by iteratively plugging new values in for those parameters until the values that maximize the likelihood are found. In other words, free parameters are values that are to be estimated by *mle*—they are the unknowns in likelihood models. If the parameter is not constrained to some fixed value in the `RUN` part of the model statement, *mle* will estimate the value of that parameter.

Covariate effects (and their associated parameters) can be modeled within the parameter statement, as well. Parameters are specified as

```
PARAM x HIGH = <expr> LOW = <expr> START = <expr> TEST = <expr>..END
```

or

```
PARAM x HIGH = <expr> LOW = <expr> START = <expr> TEST = <expr>
  COVAR <expr> PARAM z ... END
END {param}
```

Five characteristics can be set for each parameter. They are: 1) the highest possible value that can be tried for the parameter, 2) the lowest possible value that can be tried for the parameter, 3) the starting guess to help *mle* out from the start, and 4) a test value against which the parameter will be tested when standard errors are computed, and 5) the form for a parameter.

HIGH	This is the highest value of the parameter that is permitted. This value constrains the parameter and also determines the maximum limit for generating confidence intervals.
------	--

LOW	This is the lowest value of the parameter that is permitted. This value constrains the parameter and also determines the minimum limit for generating confidence intervals.
-----	---

Use care when setting the `HIGH` and `LOW` limits. Most importantly, limits must be constrained to valid ranges for the intrinsic parameter. Thus, for the `MIX` mixing proportion (the first of the three arguments; see function `MIX`) then, `HIGH = 1` and `LOW = 0`, should be defined as is appropriate for a probability—unless some `FORM` like `FORM = LOGISTIC` is used to constrain the resulting parameter to between 0 and 1. Sometimes it is useful to impose narrower limits, perhaps to avoid getting hung-up at a local maximum or to solve the model more quickly. Be careful, though. Limits that are too narrow may exclude the global maximum—after all, the best parameter estimates

¹ The word *parameter* is used in a very specific way, as defined in Chapter 1 of the *User's Manual*. Parameters are the quantities to be estimated in a likelihood model

for a set of data are presumably unknown. Excessively narrow limits may cause problems when numerical derivatives for the variance-covariance matrix are computed, as well. Also, likelihood confidence intervals will bump up and stop at the limits you set.

START	This is the initial value (or ‘guess’) used when estimating a parameter.
TEST	This is a value against which the parameter is tested in the output. The default value is 0.0. The <code>TEST = xxx</code> part of a <code>PARAM</code> function provides a value against which the parameter will be tested (in some reports). In a sense, the <code>TEST</code> value is a null hypothesis value (h_0). The test performed is $t = (\hat{p} - h_0) / SE(\hat{p})$, where \hat{p} is the maximum likelihood parameter estimate and $SE(\hat{p})$ is the standard error for the parameter estimate. The t -test is provided for convenience only. <i>mle</i> does not make use of the test in any way.
FORM	This parameter controls any mathematical transformation used for the parameter. Typically, this is used to take a parameter with a restricted valid range and transform it into a new parameter with an unrestricted range.

For some forms, the parameter itself is transformed. For example, when a parameter is a probability the parameter can be defined as:

```
PARAM p LOW = -999 HIGH = 999 START = 0 FORM = LOGISTIC END
```

The logistic transformation permits the parameter `p` to take on any value from negative infinity to infinity, but the resulting value passed used by the likelihood will be constrained to the range (0, 1). In other words, *mle* will estimate a parameter over the range -999 to 999, but before that parameter is used in computation, it will undergo a logistic transformation as $p = 1/[1 + \exp(p')]$, so that the value of p will be a probability. *mle* currently provides a limited number of specifications for how parameters and covariates are modeled. Even so, this mechanism for modeling covariates on any parameter is extremely general and provides the basis for building unique and highly mechanistic (Box et al. 1978) or etiologic (Wood 1994) models.

A description of all forms is given in Table 2.

Examples: Here is an example of a likelihood hand-coded for an exponential PDF for exact failure times. A `PARAM` and a built-in function are used in this likelihood:

```
MODEL
  DATA
    PARAM lambda LOW = 0 HIGH = 1 START = 0.23 END * EXP(-lambda * t)
  END
RUN
FULL
END
```


Notice that `lambda` is first defined as a parameter, and thereafter is used as an ordinary variable. As *mle* iteratively seeks a solution, the value of `lambda` will be updated. As the likelihood itself is being computed, the `PARAM` function will simply return the current estimate of `lambda`.

Notes:

Sometimes parameters are constrained for the purpose of hypothesis testing. They may be held constant, or fixed to the value of another parameter. These are called *fixed parameters*, and an estimate will not be found for them. *mle* provides the mechanism for fixed parameters primarily to reduce models from more complicated to simpler forms. For example, in a slope function, we may have reason to believe that the slope *m* is one. Perhaps this is because of the nature of the physical system we are modeling. We could first fit our collection of *x* values to the model with parameter *m* free, and secondly fit it with *m* held constant to 1. Statistical criteria can be used to determine whether *m* deviates from the value we expected it to be.

Typically, parameters are defined for the intrinsic parameters of a PDF function. For example, the normal PDF has two intrinsic parameters μ and σ . The first parameter specified in the parameter list will be treated as μ . The second will be treated as σ . How can you know the proper order for parameters? Generally location parameters appear first (and are usually denoted *a* in this manual), scale parameters are second and shape parameters are third. Even so, you can get a quick synopsis of each type of PDF by using the `-h` option from the command line, e.g.: `mle -h SHIFTWEIBULL`

Parameters are also used to model effects of covariates on other parameters. Here is an example in which two parameters, used in place of some fixed values of μ and σ for a normal distribution, are defined with two covariate parameters, each:

```
PDF NORMAL(topen tclose)
  PARAM mean LOW = 100 HIGH = 400 START = 270 TEST = 0 FORM = LOGLIN
    COVAR sex PARAM b_sex_mu LOW = -2 HIGH = 2 START = 0 END
    COVAR weight PARAM b_weight_mu LOW = -2 HIGH = 2 START = 0 END
  END
  PARAM stdev LOW = 0.1 HIGH = 100 START = 20 FORM = LOGLIN
    COVAR sex PARAM b_sex_sig LOW = -2 HIGH = 2 START = 0 END
    COVAR weight PARAM b_weight_sig LOW = -2 HIGH = 2 START = 0 END
  END
END
```

In this example, the first parameter of the normal distribution (μ) has two covariates and their corresponding parameters modeled on it. The exact specification of how covariates and their parameters are modeled depend on the `FORM` of the intrinsic parameter. In the example, the `FORM = LOGLIN` specifies that a log-linear specification is to be used. The log-linear specification is $\mu_i = \mu' \exp(\mathbf{x}_i \mathbf{b})$, where μ' is the estimated intrinsic parameter (mean in this case). Thus, for the *i*th observation, the μ parameter of the normal distribution will be constructed as: $\mu_i = \text{mean} \times \exp(\text{sex}_i \times \text{b_sex} + \text{weight}_i \times \text{b_weight})$.

Table 2. Forms and transformations for parameters.

Form	Parameter (p'), covariates (\mathbf{x}_i), covariate parameters (\mathbf{b}), and the value returned by the PARAM function (p_i)	Notes
NUMBER	$p_i = p'$	Default when no COVARs are modeled.
ADD	$p_i = p' + \mathbf{x}_i\mathbf{b}$	Must be used with care when the resulting parameter is constrained to positive values because p_i might take on negative values for some combinations of $\mathbf{x}_i\mathbf{b}$
INVERT	$p_i = 1/(p' + \mathbf{x}_i\mathbf{b})$	The denominator must not be zero.
INVADD	$p_i = 1/p' + \mathbf{x}_i\mathbf{b}$	p' must not be zero.
INVMULTIPLY	$p_i = \mathbf{x}_i\mathbf{b}/p'$	p' must not be zero.
INVLOGLIN	$p_i = \exp(\mathbf{x}_i\mathbf{b})/p'$	p' must not be zero.
DIVIDE	$p_i = p'/\mathbf{x}_i\mathbf{b}$	$\mathbf{x}_i\mathbf{b}$ must not be zero.
POWER	$p_i = p'^{\mathbf{x}_i\mathbf{b}}$	
POWEREXP	$p_i = p'^{\exp(\mathbf{x}_i\mathbf{b})}$	
EXPADD	$p_i = \exp(p' + \mathbf{x}_i\mathbf{b}) = \exp(p')\exp(\mathbf{x}_i\mathbf{b})$	Constrains p_i to positive values for all p' and $\mathbf{x}_i\mathbf{b}$.
FISHER	$p_i = \ln[(1 + p' + \mathbf{x}_i\mathbf{b})/(1 - p' + \mathbf{x}_i\mathbf{b})]/2$	This specification is useful when p_i can take on any value from $-\infty$ to ∞ and $-1 \leq p' + \mathbf{x}_i\mathbf{b} \leq 1$.
FISHERINV	$p_i = \frac{\exp[2(p' + \mathbf{x}_i\mathbf{b})] - 1}{\exp[2(p' + \mathbf{x}_i\mathbf{b})] + 1}$	Frequently used for parameters that are correlation coefficients because, for all values of $p' + \mathbf{x}_i\mathbf{b}$, p_i will be constrained from -1 to 1.
MULTIPLY	$p_i = p' \times \mathbf{x}_i\mathbf{b}$	A multiplicative specification.
EXCESS	$p_i = p' \exp(1 + \mathbf{x}_i\mathbf{b})$	
LOGLIN	$p_i = p' \exp(\mathbf{x}_i\mathbf{b})$	This is a common specification, especially for parameters that are interpreted as hazards. When p' is constrained positive, the p_i will also be positive. Like EXPADD but $p'_{expadd} = \exp(p'_{loglin})$. LOGLIN is the default specification whenever a COVAR is defined.
LOGISTIC	If ALTERNATE_LOGISTIC = FALSE, $p_i = 1/[1 + \exp(p' + \mathbf{x}_i\mathbf{b})]$. If ALTERNATE_LOGISTIC = TRUE, $p_i = \exp(p' + \mathbf{x}_i\mathbf{b})/[1 + \exp(p' + \mathbf{x}_i\mathbf{b})]$	Frequently used for parameters that are interpreted as probabilities because, for all values of $p' + \mathbf{x}_i\mathbf{b}$, p_i will be constrained from zero to one. The alternative forms are related to each other as $p'_{form1} = 1 - p'_{form2}$
LOGIT	$p_i = \ln[\exp(p' + \mathbf{x}_i\mathbf{b})/(1 + \exp(p' + \mathbf{x}_i\mathbf{b}))]$	This specification is useful when p_i can take on any value from $-\infty$ to ∞ and $p' + \mathbf{x}_i\mathbf{b}$ is a probability.

The second parameter, `stdev`, has the same two covariates modeled on it, but the parameter names are (and must be) different from the parameters modeled on `mean`.

In the previous example, the `mean` parameter was constrained to the range [100, 400] and the initial guess was 270.

PDF

The `PDF` function returns the value of, or area under, a pre-defined probability density or distribution functions. Although the name is `PDF`, the `PDF` function can return the probability density function, the cumulative density function, the survival density function, and the hazard function. In addition, the `PDF` function can return areas or densities that are left and right truncated. The structure of the `PDF` function call is:

```
PDF <PDF name> ( <time variable1>, <time variable2>,...)
  <intrinsic parameter 1>,
  <intrinsic parameter 2>,
  ...
  <optional HAZARD parameter>
END
```

The *name* following `PDF` is the name of the built-in distribution. A summary for each built-in distribution is given in later chapters.

Time variable list is a list of the time arguments passed to the `PDF`. Most univariate PDFs can take from one to four ‘time’ arguments.² In fact, these four times describe a single observation in such a way as to incorporate defects in the observation process (right censoring, left truncation, right truncation, cross-sectional). A description of how the four arguments combine to specify a probability are given in the section that follows. Note that the time arguments can be any expression, so that time shifts and transformations can be incorporated in this list.

Intrinsic parameter list provides specifications for the `PDF`’s intrinsic parameters. The order that the intrinsic parameters are specified is important; it corresponds to how the `PDF` is defined within *mle*. The `PDFs` chapter lists the order for intrinsic parameters; alternatively, the command line `mle -h` can be used to determine the proper argument order. Note that any expression can be used for an intrinsic parameter. That is, you do not need to use a `PARAM` function for the intrinsic parameters, although this is the most common use. Here is an example in which the location parameter is fixed to a constant for a shifted lognormal distribution:

```
PDF SHIFTLGNORMAL ( tooth_eruption_age )
  9, {shift the time back to conception}
  PARAM location LOW = 1 HIGH = 4 START = 2.5 END,
  PARAM scale LOW = 0.0001 HIGH = 3 START = 0.9 END
END
```

PDF Time Arguments

Most `PDFs` can have as few as one and as many as four time arguments specified. They are: t_u , the last observation time before an event; t_e , the first observed time after the event; t_a , the left truncation time for the observation or the `PDF`; and t_w , the right truncation time for the observation or the `PDF`. Understanding how

² These are called *time* variables in the context of survival analysis; however, they may represent other measurements (length, dose, height, etc.).

these four times act on the `PDF` statement is critical to creating the desired and proper likelihood.

PDFs contribute to likelihoods in a number of ways. In survival analysis, for example, the likelihood for an exact failure is given by the value of the PDF at the exact point of failure. For a right censored observation, the likelihood is given by summing up (integrating) all possible PDF values from the last observation time until the maximum possible time. The likelihood for a cross-sectional “responder” is the integral from zero to the time of first observation. Table 3 lists the values that result from the four time variables for different conditions. For example, when $t_u=t_e$ or when only one time variable is specified, *mle* returns the density at t_u . This is the desired likelihood for an exact failure. Likelihoods for right and interval censored observations, with and without left and right truncation are given in Table 3

The Hazard Parameter

For most parametric distributions (like the normal or lognormal distributions) the hazard function does not take on a simple or closed form. For this reason, most studies have modeled the covariates as acting on the failure time for these distributions. Nevertheless, there is no inherent reason why hazards models cannot be constructed using distributions without a closed form for the hazards functions. Most of the PDFs built into *mle* provide a general mechanism for covariates to be modeled as affecting the hazard of failure, rather than (or in

Table 3. Probabilities (or likelihoods) returned by PDF for one, two, three, and four time variables.

	Example	When	Class	Result
1	$\text{LNNORMAL}(t_e)$		Exact failure at t_e	$L = f(t_e)$
2	$\text{LNNORMAL}(t_u, t_e)$	$t_u = t_e$	Exact failure at $t_u = t_e$	$L = f(t_u) = f(t_e)$
3	$\text{LNNORMAL}(t_u, t_e)$	$t_e = \infty$ $t_e < t_u$	Right censored or cross-sectional non-responder at t_u	$L = \int_{t_u}^{\infty} f(z) dz = S(t_u)$
4	$\text{LNNORMAL}(t_u, t_e)$	$t_u = 0$	Cross-sectional responder at t_e	$L = \int_0^{t_e} f(z) dz = F(t_u)$
5	$\text{LNNORMAL}(t_u, t_e)$	$t_u \neq t_e$	Interval censored over the interval (t_u, t_e) . Includes, as a limiting case cross-sectional responder and right-censored.	$L = \int_{t_u}^{t_e} f(z) dz = S(t_u) - S(t_e)$
6	$\text{LNNORMAL}(t_u, t_e, t_a)$	$t_u = t_e$	Left-truncated exact failure	$L = \frac{f(t_u)}{\int_{t_a}^{\infty} f(z) dz} = \frac{f(t_u)}{S(t_a)}$
7	$\text{LNNORMAL}(t_u, t_e, t_a)$	$t_u \neq t_e$	Left-truncated, interval censored failure	$L = \frac{S(t_u) - S(t_e)}{\int_{t_a}^{\infty} f(z) dz} = \frac{S(t_u) - S(t_e)}{S(t_a)}$
8	$\text{LNNORMAL}(t_u, t_e, t_a, t_w)$	$t_u = t_e$	Left- and right-truncated, exact failure	$L = \frac{f(t_e)}{\int_{t_a}^{t_w} f(z) dz} = \frac{f(t_e)}{S(t_a) - S(t_w)}$
9	$\text{LNNORMAL}(t_u, t_e, t_a, t_w)$	$t_u < t_e$ $t_a \leq t_u$ $t_w \geq t_e$	Left- and right-truncated, interval censored failure	$L = \frac{S(t_u) - S(t_e)}{\int_{t_a}^{t_w} f(z) dz} = \frac{S(t_u) - S(t_e)}{S(t_a) - S(t_w)}$
10	$\text{LNNORMAL}(t_u, t_e, t_a)$	$t_u = t_e = t_a$	Hazard	$L = \frac{f(t_u)}{S(t_u)} = h(t_u)$
11	$\text{LNNORMAL}(t_u, t_e, t_a, t_w)$	$t_u = t_e = t_a$	Right-truncated hazard	$L = \frac{f(t_u)}{S(t_u) - S(t_w)} = h(t_u)$

addition to) affecting intrinsic parameters. Here is an example:

```
PDF NORMAL(topen tclose)
  PARAM mean  LOW = 100  HIGH = 400  START = 270  TEST = 0  FORM = LOGLIN END,
  PARAM stdev  LOW = 0.1  HIGH = 100  START = 20  END,
  HAZARD COVAR sex      PARAM b_sex      LOW = -2  HIGH = 2  START = 0  END
          COVAR weight  PARAM b_weight  LOW = -2  HIGH = 2  START = 0  END
  END {hazard}
END
```

The covariates `sex` and `weight` are modeled to effect on the hazard of failure. Parameters `b_sex` and `b_weight` provide estimates of the effect.

The `HAZARD` statement always provides a proportional hazards specification modeled directly on the hazard of the PDF. Usually, the specification is loglinear, so that the hazard for the i th observation including the covariate effects defined as $h_i(t_i|\mathbf{x}_i\mathbf{b}) = h(t_i)\exp(\mathbf{x}_i\mathbf{b})$, where $h(t)$ is the baseline hazard for the specified PDF. Then, the survival function becomes $S_i(t_i|\mathbf{x}_i\mathbf{b}) = S(t_i)^{\exp(\mathbf{x}_i\mathbf{b})}$, and the probability density function becomes $f_i(t_i|\mathbf{x}_i\mathbf{b}) = f(t_i)S(t_i)^{\exp(\mathbf{x}_i\mathbf{b})-1}\exp(\mathbf{x}_i\mathbf{b})$. The reason for exponentiating the $\mathbf{x}_i\mathbf{b}$ array is to prevent it from going negative (hazards are *always* be positive).

A multiplicative form for the proportional hazards specification can also be specified by setting the constant `EXP_HAZARD = FALSE` (it is `TRUE` by default). Then, the model is $h_i(t_i|\mathbf{x}_i\mathbf{b}) = h(t_i)\mathbf{x}_i\mathbf{b}$, $S_i(t_i|\mathbf{x}_i\mathbf{b}) = S(t_i)^{\mathbf{x}_i\mathbf{b}}$, and $f_i(t_i|\mathbf{x}_i\mathbf{b}) = f(t_i)S(t_i)^{\mathbf{x}_i\mathbf{b}-1}\mathbf{x}_i\mathbf{b}$. You must insure that $\mathbf{x}_i\mathbf{b}$ never goes negative.

PHAZARD

This is an experimental function that has not been fully checked. The `PHAZARD` function returns the p-hazard function for the specified probability density function. The p -hazard function is the hazard for a given p-probability density function (see **PPDF**, page 103).

```
PHAZARD <pdfname>(<quantile> [, <left_expr>[, <right_expr>]])
  <intrinsic parameter 1>,
  <intrinsic parameter 2>,
  ...
END
```

The `<pdfname>` following `PHAZARD` is the name of the built-in distribution. A summary for each built-in distribution is given in later chapters.

`<quantile>` is the quantile (range 0.0 to 1.0) for which to assess the p-hazard.

The optional `<left_expr>` and `<right_expr>` define the left and right truncation points for the underlying PDF. Two additional arguments `<eps>` and `<maxits>` can also be include after `<right_expr>`; see their descriptions for the **QUANTILE** function on page 106.

Intrinsic parameter list provides specifications for the underlying PDF's intrinsic parameters. The order that the intrinsic parameters are specified is important; it corresponds to how the PDF is defined within *mle*. The PDFs chapter lists the

order for intrinsic parameters; alternatively, the command line `mle -h` can be used to determine the proper argument order.

For example, `PHAZARD NORMAL(0.025) 4, 1 END` returns 0.0599436597

The p -hazard function is computed as $h_p(p) = [q(p)(1 - p)]^{-1} = f_p(p)/(1 - p)$, where $q(p)$ is the quantile density function and $f_p(p)$ is a p -PDF. Hence the above result is obtained with

```
1/(QDF NORMAL(0.025) 4, 1 END*(1 - 0.025)) → 0.0599436597
PPDF NORMAL(0.025) 4, 1 END/(1 - 0.025) → 0.0599436597
```

Some PDFs have no closed form. The quantile functions (QUANTILE, QDF, and PPDF) are computed iteratively.

See also: PDF, QUANTILE, QDF, PPDF

Details about quantile functions can be found in Gilchrist (2000).

PPDF

This is an experimental function that has not been fully checked. The `PPDF` function returns the p -PDF function for the specified probability density function. The p -PDF function is the probability density function specified in terms of quantile p .

```
PPDF <pdfname>(<quantile> [, <left_expr>[, <right_expr>]])
  <intrinsic parameter 1>,
  <intrinsic parameter 2>,
  ...
END
```

The `<pdfname>` following `PPDF` is the name of the built-in distribution. A summary for each built-in distribution is given in later chapters.

`<quantile>` is the quantile (range 0.0 to 1.0) for which to assess the p -PPDF.

The optional `<left_expr>` and `<right_expr>` define the left and right truncation points for the underlying PDF. Two additional arguments `<eps>` and `<maxits>` can also be include after `<right_expr>`; see their descriptions for the **QUANTILE** function on page 106.

Intrinsic parameter list provides specifications for the underlying PDF's intrinsic parameters. The order that the intrinsic parameters are specified is important; it corresponds to how the PDF is defined within *mle*. The PDFs chapter lists the order for intrinsic parameters; alternatively, the command line `mle -h` can be used to determine the proper argument order.

For example, `PPDF NORMAL(0.025) 4, 1 END` returns 0.0584450682.

The p -PDF function is computed as $f_p(p) = f[Q(p)] = 1/q(p)$, where $q(p)$ is the quantile density function, $Q(p)$ is the quantile function, and $f(x)$ is the probability density. Hence the above result is obtained with

```
PDF NORMAL( QUANTILE NORMAL(0.025) 4, 1 END ) 4, 1 END → 0.0584450682
1/QDF NORMAL(0.025) 4, 1 END → 0.0584450682
```

Some PDFs have no closed form. The quantile functions (QUANTILE, QDF, and PPDF) are computed iteratively.

See also: PDF, QUANTILE, QDF, PHAZARD

Details about quantile functions can be found in Gilchrist (2000).

PREASSIGN and POSTASSIGN

The likelihood is always specified as a single function. This means that within a likelihood, a special function must be used to compute intermediate results or perform other computations. The PREASSIGN...END function provides a mechanism to compute partial results of a likelihood outside of the main likelihood, or within part of the likelihood. The statement takes on this form

```
PREASSIGN
  <statement>
  <expression>
END
```

A single statement is defined immediately after the PREASSIGN. This statement will be executed prior to evaluation of the <expression>. After that, the <expression> is evaluated, and the result returned by the PREASSIGN function. For example, the following code would reparameterize the exponential PDF so that the parameter λ is replaced by the function $-b^{-1}$

```
MODEL
  PREASSIGN
    z = -1/PARAM b LOW = 0 HIGH = 1 START = 0.1 END
  ,
  DATA
    PDF EXPONENTIAL( t ) z END
  END {data}
  END {preassign}
RUN
  FULL
END
```

Notice that first the value z is assigned the value $-b^{-1}$. Next the likelihood $e^{\lambda t}$ is computed. But, lambda is constrained to z. The assignment in the first part of the PREASSIGN function will be executed for each observation.

The PREASSIGN and POSTASSIGN statements can be used with multiple statements by using an BEGIN...END block. It is sometimes helpful to define all parameters outside of the DATA statement and then use only variables in the likelihood expression. In other words, the parameters are defined “up front”, so that the likelihood function itself is easier to specify. Here is an example of re-coding a program so that all parameters are defined in advance.


```

MODEL          {mixture of two normal distributions}
PREASSIGN
BEGIN
  pr = PARAM  p      LOW = 0    HIGH = 1    START = 0.5 END
  u1 = PARAM  mu1    LOW = 5     HIGH = 14   START = 8    END
  s1 = PARAM  sigma1 LOW = 0.1   HIGH = 5    START = 1.2  END
  u2 = PARAM  mu2    LOW = 0     HIGH = 6    START = 2    END
  s2 = PARAM  sigma2 LOW = 0.01  HIGH = 5    START = 1.2  END
END
{now specify the likelihood expression }
DATA
  MIX(pr, PDF NORMAL(topen tclose) u1 s1 END, PDF NORMAL(topen tclose) u2 s2 END)
END {data}
END {preassign}
RUN
FULL
END {model}

```

The POSTASSIGN.<expr> <statement> END function is similar to the preassign function, except that the statement come *after* the function expression. The function defined by the first <expression> is evaluated first, and is the result returned by the POSTASSIGN function. Then the statements is evaluated with each result assigned to the corresponding <variable>. A BEGIN...END block allows for multiple statements.

PRODUCT

The PRODUCT function computes a finite product. The format is similar to the INTEGRATE or SUMMATION functions:

```

PRODUCT <variable name> (<lower_limit> , <upper_limit>)
  <expression>
END

```

or

```

PRODUCT <variable name> (<lower_limit> , <upper_limit>, <convergence>)
  <expression>
END

```

The expressions <lower_limit> and <upper_limit> define the lower and upper limits of the product. These expressions (as well as the optional <convergence> expression) are evaluated once. The optional <convergence> expression provides a second way to terminate the series. When used, the product will terminate when the difference between one product and the next is less than the value of the <convergence> expression

The PRODUCT function can be used, for example, to calculate likelihoods that incorporate geometric series. The inner <expression> will be repeatedly evaluated with the index variable incremented for each evaluation. Here is an example of a likelihood consisting of a Polya-Eggenberger distribution (Eggenberger and Pólya 1923) for exact failures at integer times t . The probability density function for the Polya-Eggenberger distribution is

$$f(t_i | p, n, c) = \frac{\binom{n}{t_i} \prod_{i=0}^{t_i-1} (p + ic) \prod_{j=0}^{n-t_i-1} (1 - p + jc)}{\prod_{k=0}^{n-1} (1 + kc)}$$

There are three products that must be computed as part of computing the density function. The following *mle* program fragment shows the code needed to implement the Polya-Eggenberger distribution. Parameters p and c are to be estimated for a set of observations \mathbf{t} .

```

MODEL
  DATA
    COMB(n, t)
    * PRODUCT i (0, t - 1)
      PARAM p LOW = 0 HIGH = 1 START = 0.5 END
      * i * PARAM c LOW = -1 HIGH = 25 START = 1 END
    END {product}
    *
      PRODUCT j (0, n - t - 1)
        1 - p + j*c
      END {product}
    /
      PRODUCT k (0, n - 1)
        1 - k*c
      END
    END {data}
  RUN
  FULL
END

```

QUANTILE

The QUANTILE function returns the inverse cumulative density for one of *mle*'s predefined probability density functions. For many of the built in pdfs, the QUANTILE function calculates the result from a closed-form solution. Some functions do not have simple solutions, so *mle* uses an iterative search for the solution. Formats of the QUANTILE function are:

```

QUANTILE <pdf> (<p>) <param1>, ... END
QUANTILE <pdf> (<p>, <lefttrunc> ), <param1>, ... END
QUANTILE <pdf> (<p>, <lefttrunc>, <righttrunc>) <param1>, ... END
QUANTILE <pdf> (<p>, <lefttrunc>, <righttrunc>, <eps>) <param1>, ... END
QUANTILE <pdf> (<p>, <lefttrunc>, <righttrunc>, <eps>, <maxits>) <param1>, ... END

```

<pdf> is one of *mle*'s built-in pdf names, like NORMAL, WEIBULL, etc. The expression <p> must evaluate to a probability. This is the area under the pdf for which the function will return a value of t .

<lefttrunc> is an optional left truncation time, so that quantiles can be returned for left-truncated densities.

<righttrunc> is optional and specifies a right-truncation time for the pdf, so that both left- and right-truncated.

<eps> is optional and is the precision to which the quantile is to be found. This argument has no effect unless an iterative method is used to find the solution. If

the third expression is not given, and an iterative solution is necessary, its value will be taken from the variable `FIND_EPS`.

`<maxits>`, the optional fifth expression, is the maximum number of iterations allowed in finding the solution. Again, this argument has no effect unless an iterative solution is used. If this expression is not given and an iterative solution is required, the value will be taken from the variable `FIND_ITERS`.

The function `QUANTILE NORMAL(0.975) 0, 1 END` will return the value of t that yields a probability of 0.975. The resulting value is 1.95996. The median of most pdfs can be returned by specifying a probability of 0.5. Thus, the median of a gamma function could be found with, for example, `QUANTILE GAMMA(0.5) 0.3, 2 END` returns 5.5945.

The quantile function can be used to draw random variates from the built-in probability density functions. This is done by calling function `RAND` for the argument to the `QUANTILE` function. For example, `QUANTILE NORMAL(RAND) 0, 1 END` will return a standard normal random variate.

QDF

This is an experimental function that has not been fully checked. The `QDF` function returns the quantile density function for the specified probability density function. The `QDF` function is the probability density function specified in terms of quantile p .

```
PPDF <pdfname>(<quantile> [, <left_expr>[, <right_expr>]])
  <intrinsic parameter 1>,
  <intrinsic parameter 2>,
  ...
END
```

The `<pdfname>` following `QPDF` is the name of the built-in distribution. A summary for each built-in distribution is given in later chapters.

`<quantile>` is the quantile (range 0.0 to 1.0) for which to assess the `QDF`.

The optional `<left_expr>` and `<right_expr>` define the left and right truncation points for the underlying PDF. Two additional arguments `<eps>` and `<maxits>` can also be include after `<right_expr>`; see their descriptions for the `QUANTILE` function on page 106.

Intrinsic parameter list provides specifications for the underlying PDF's intrinsic parameters. The order that the intrinsic parameters are specified is important; it corresponds to how the PDF is defined within *mle*. The PDFs chapter lists the order for intrinsic parameters; alternatively, the command line `mle -h` can be used to determine the proper argument order.

For example, `QDF NORMAL(0.025) 4, 1 END` returns 17.110083553.

The `QDF` function is computed as $q(p) = dQ(p)/dp$ where $Q(p)$ is the quantile function. Hence the above result is obtained (approximately) with

```
DERIVATIVE p=0.025 QUANTILE NORMAL(p) 4, 1 END END → 17.110082824
```

Some PDFs have no closed form. The quantile functions (QUANTILE, QDF, and PPDF) are computed iteratively.

See also: PDF, QUANTILE, PPDF, PHAZARD

Details about quantile functions can be found in Gilchrist (2000).

SUMMATION

The SUMMATION function computes a finite sum. The format is similar to the INTEGRATE or PRODUCT functions:

```
SUMMATION <variable name> (<lower_limit> , <upper_limit>)
  <expression>
END
```

or

```
SUMMATION <variable name> (<lower_limit> , <upper_limit>, <convergence>)
  <expression>
END
```

The expressions *<lower_limit>* and *<upper_limit>* define the lower and upper limits of the sum. These expressions (as well as the optional *<convergence>* expression) are evaluated once. The optional *<convergence>* expression provides a second way to terminate the series. When used, the summation will terminate when the difference between one sum and the next is less than the value of the *<convergence>* expression

Here is an example of a likelihood hand-coded for a Thomas distribution (Thomas 1949) for exact failures at integer times t . The probability density function for the Thomas distribution is

$$f(t_i | a, b) = e^{-a} \sum_{i=1}^{t_i} \frac{e^{-ib} a^i (ib)^{t_i-i}}{i!(t-i)!}$$

A single finite summation must be computed as part of computing the density function. The following *mle* program fragment shows the code needed to implement the Thomas distribution. Parameters a and b are to be estimated for observed times t .

```
MODEL
  DATA
    EXP(-PARAM a LOW = 0.0001 HIGH = 20 START = 5 END)
    * SUMMATION i (1, t)
      EXP(-i * PARAM b LOW=0.0001 HIGH=40 START=0.5 END)
      * a^i * (i*b)^(t - i) / (FACT(i)*FACT(t - iz))
    END {summation}
  END {data}
RUN
  FULL
END
```


Chapter 7

Statements

An *mle* program is, essentially, a collection of statements. This chapter describes all statements types available in *mle*.

Statements

Assignment statement

Assignment statements serve two purposes. The primary purpose is to assign values to variables. A secondary purpose is to define new variables.

A great number of pre-defined variables are available to change or fine-tune the behavior of *mle*, and the values of these variables can be changed with assignment statements. Some brief examples are:

MAXITER = 100	{Set the maximum number of iterations}
EPSILON = 0.0000001	{Set the criterion for convergence}
PRINT_OBS = TRUE	{prints all observations after transforms}

Assignment statements may be placed anywhere within the `MLE...END` statement around the `DATA...END` and `MODEL...END` statements.³ The generic forms for assignment statements are:

<code><var> = <expression></code>
<code><var>:<type> = <expression></code>

where `<var>` is the name of a variable. If the variable does not exist, it will be created. If so, and the first form of the assignment statement is used, the *type* (INTEGER, REAL, COMPLEX, etc.) returned by `<expression>` will define the type of the newly created variable.

³ Within the DATA statement, assignment statements are used for transformations. Within both the DATA and model statements, the PREASSIGN and POSTASSIGN functions allow a list of one or more assignments to be used. Finally, within the MODEL statement, there are several other uses for assignment statements, like to define start, highest, and lowest values of parameters.

Under the second form of assignment, the variable type is explicitly defined by `<type>`. A discussion of variable types is found later in this chapter. An important thing to understand about types and assignment statements is that the type returned by the `<expression>` must be compatible with `<type>` defined for a variable or an error will result. Some examples of assignment statements using expressions and types are:

```
MLE
y = Sqrt(44.5)           {evaluates to 6.6708...}
z = BETA(1.2, 9*3/10 + 1) {evaluates to 0.185...}
q = RAND                 {evaluates to a random number from 0 to 1}
r:REAL = 2               {defines r as real, assigns the value 2.0}
ru : REAL                {Defines ru, but does not initialize it}
s = "This is a string"   {s is created as type string}
b = IF RAND < 0.2 THEN TRUE ELSE FALSE END {b is created as type boolean}
ra : REAL[-180 TO 180]   {Defines an array of type REAL with elements from -180
to 180}
ia : INTEGER[0 TO 100, -1 TO 1] = 0 {Define an integer matrix and initialize
elements to 0}
bmi_max = weight_max/height_max^2
total = e1_count + e2_count + e3_count + e4_count
last_age = IF linear THEN max_age ELSE Sqrt(max_age) END
area = PDF NORMAL(-2, 2) 1, 3 END {gives area from -2 to 2 for N(1, 3)}
one = SIN(total)^2 + COS(total)^2
END
```

When a variable is first used in an assignment statement, its type is determined based on the type returned from the expression on the right-hand side of the assignment. Here are some examples:

```
large_data = N_OBS > 5000 {large_data is declared as type BOOLEAN}
subtitle   = "Analysis of " + INFILE {subtitle is declared as type STRING}
nine      = 3 * 3.0        {nine is type REAL}
five      = 2 + 3          {five is type INTEGER}
```

You can explicitly define a variable's type when the variable is first referenced in an assignment statement. Here are some examples:

```
c:STRING = 'x' {c would otherwise be CHAR}
nine:REAL = 3 * 3 {nine would otherwise be INTEGER}
t:BOOLEAN = TRUE {t is explicitly declared as boolean, although this is the
default}
ang:REAL = SIN(2*pi) {ang is explicitly declared as real, although this is the
default}
```

Array assignments

Multidimensional arrays of all types are supported by *mle*. Arrayed variables must be explicitly defined the first time the variable is mentioned in the program. The format is `<var> : <type>[<min1> TO <max1>, <min2> TO <max2>, ...]`. Some examples of declarations are:

```
s : STRING[1 TO 5]           {Defines a one-dimensional array of strings}
r : REAL[1 TO 10, 1 TO 10]   {Defines a 10 x 10 matrix}
b : BOOLEAN[0 TO 1, 0 TO 1, 0 TO 1] {Defines a 3 dimensional array}
```

An entire array can be initialized to a single value in an assignment statement. Some examples are:


```
s : STRING[1 TO 5] = ''      {Defines s and initializes all values to ''}
r : REAL[1 TO 10, 1 TO 10] = 0 {Defines 10 x 10 matrix and initializes elements to 0}
```

Individual elements within an array variable are accessed using brackets enclosing a subscripting. The following example creates an array of values for the sine function.

```
r : REAL[0 TO 359]
FOR i = 0 TO 359 DO
  r[i] = DTOR(i)
  writeln("Sin(" i ") = " SIN(r[i]) )
END
```

BEGIN statement

The BEGIN...END statement provides a means of providing multiple statements in contexts where only a single statement is allowed. The format is

```
BEGIN
  <statements>
END
```

The most important use for this statement is with the PREASSIGN...END and POSTASSIGN...END functions discussed in the **Special Functions** chapter.

BREAK statement

The BREAK statement works within the WHILE, REPEAT, and FOR statements. When a BREAK statement is encountered, the loop is immediately exited. The behavior of a BREAK statement outside of a loop causes the current "scope" to be exited. This means that within the main program (outside of a user-defined procedure or function) a BREAK acts like a HALT statement. Within a user-defined procedure or function, the procedure or function is exited. For example, the code

```
MLE

PROCEDURE testbreak
  WRITELN("Before BREAK in PROCEDURE testbreak")
  BREAK
  WRITELN("After BREAK - Shouldn't see this")
END

WHILE TRUE DO
  WRITELN("Before BREAK in WHILE")
  BREAK
  WRITELN("After BREAK in WHILE - Shouldn't see this")
END

WRITELN("Done with BREAK in WHILE")
testbreak
WRITELN("After testbreak")

END
```

produces the output

```

Before BREAK in WHILE
Done with BREAK in WHILE
Before BREAK in PROCEDURE testbreak
After testbreak

```

CONTINUE statement

The CONTINUE statement works within the WHILE, REPEAT, and FOR statements. When a CONTINUE statement is encountered, all further statements are skipped until the end of the loop. For example, the code

```

MLE

  x = 0
  WHILE x < 9 DO
    x = x + 1
    IF ISODD(x) THEN
      CONTINUE
    END {if}
    WRITELN('x is ' x)
  END {while}

  WRITELN("Done with WHILE")

END {mle}

```

produces the output

```

x is 2
x is 4
x is 6
x is 8
Done with WHILE

```

CURVE statement

The CURVE...END statement is a statement that occurs only within PLOT statements (see the PLOT statement, page 128). Each CURVE statement creates a single curve or surface. The CURVE statement is discussed in more detail in the chapter on plots.

Two-dimensional plots

The idea of the curve statement is to generate a series of points for a function. For simple curves two points must be defined: an x value and its corresponding y value. There are two forms for the CURVE statement (for producing two-dimensional plots). One form generates a series of REAL x values for use in computing y values. The second form generates an INTEGER series of points. The REAL version looks like this:

```

CURVE
  [KEY <keystring> | WITH <withstring> | AXES <axesstring> . . . ]
  <x_var> ( <x_min> <x_max> [ <x_points> ] )
  <x_expr> <y_expr> [ <expr> . . . ] [ <strings> . . . ]
END

```

The KEY, WITH, and AXES will be discussed later. This form of the CURVE statement creates a series of x points. It begins with the point $\langle x_{min} \rangle$ and ends

with the point $\langle x_{max} \rangle$; $\langle x_{points} \rangle$ points will be generated in total. Each point will be assigned to $\langle x_{var} \rangle$ in turn. The value of $\langle x_{var} \rangle$ will be used at each point to compute $\langle x_{expr} \rangle$ and $\langle y_{expr} \rangle$ (and perhaps other expressions as well). If the expression for $\langle x_{points} \rangle$ is missing, the value stored in PLOTPOINTS will be used instead (which is initially 100).

Here is an example that draws two curves on a plot:

```
MLE
  PLOTFILE(DEFAULTPLOTNAME)
  PLOT
    CURVE z ( 0, 15, 100 ) z, PDF NORMAL(z) 5, 2 END END
    CURVE z ( 0, 15, 100 ) z, PDF WEIBULL(z) 5.5, 2 END END
  END {plot}
END
```

The second form for the two-dimensional curve statement generates a series of INTEGER x values for use in computing y values. It looks like this:

```
CURVE
  [KEY <keystring> | WITH <withstring> | AXES <axesstring> . . . ]
  <x_var> = <x_min> TO <x_max>
  <x_expr> <y_expr> [ <expr> . . . ] [ <strings> . . . ]
END
```

This form of the CURVE statement creates a series of INTEGER x points. It begins with $\langle x_{var} \rangle$ set to $\langle x_{min} \rangle$ and ends with the point $\langle x_{max} \rangle$. The value of $\langle x_{var} \rangle$ will be incremented by 1 for each point and will be used to compute $\langle x_{expr} \rangle$ and $\langle y_{expr} \rangle$ (and perhaps other expressions as well). Here is an example that draws two curves on a plot:

```
MLE
  PLOTFILE(DEFAULTPLOTNAME)
  PLOT ("set data style boxes", "set xrange [-0.5:12.5])
    CURVE i = 0 TO 10 i, PDF BINOMIAL(i) 0.5, 10 END END
    CURVE i = 1 TO 12 i, PDF GEOMETRIC(i) 0.2 END END
  END {plot}
END
```

Each CURVE...END statement defines a single graph as a series of x and y points. The x and y values (and perhaps some values used for error bars and other things) are written to a data file. These data files (one per CURVE...END statement) are read by *Gnuplot* when creating the graphs.

KEY

There are three optional keywords that can be used in the CURVE...END statement. The first is KEY, followed by a string expression. This sets up a title for the plot key.

AXES

The AXES keyword defines the axis to which a curve will be plotted. A single string expression follows AXES. Valid values for this string are "x1y1", "x2y1", "x1y2", and "x2y2".

WITH

The `WITH` keyword defines the style of curve to be plotted, along with any options for that style. A single string expression follows `WITH`. The string begins with one of the *Gnuplot* plot styles, and is followed by options for that style. *mle* checks the first word of this string and makes sure there are enough `PLOT` expressions for the desired graph type. The information is also used to put together the *Gnuplot* `plot` or `splot` command. Valid values for the first word of this string are:

WITH style string	Number of expressions
"boxerrorbars"	4 to 6 CURVE expressions (2d only)
"boxes"	2 CURVE expressions (2d only)
"boxxyerrorbars"	4 to 7 CURVE expressions (2d only)
"candlesticks"	7 CURVE expressions (2d only)
"dots"	2 (2d) or 3 (3d) CURVE expressions
"errorbars"	3 to 4 CURVE expressions (2d only)
"financebars"	7 CURVE expressions (2d only)
"fsteps"	2 CURVE expressions (2d only)
"histeps"	2 CURVE expressions (2d only)
"impulses"	2 (2d) or 3 (3d) CURVE expressions
"lines"	2 (2d) or 3 (3d) CURVE expressions
"linespoints"	2 (2d) or 3 (3d) CURVE expressions
"points"	2 (2d) or 3 (3d) CURVE expressions
"steps"	2 CURVE expressions (2d only)
"vector"	4 (2d) or 5 (3d) CURVE expressions
"xerrorbars"	3 to 4 CURVE expressions (2d only)
"xyerrorbars"	4 to 6 CURVE expressions (2d only)
"yerrorbars"	3 to 4 CURVE expressions (2d only)

Options can follow each plot style in the `WITH` string. The options are `linetype <number>`, `linesize <number>`, `linewidth <number>`, `pointtype <number>` and `pointsize <number>` (the options can be abbreviated `lt`, `ls`, `lw`, `pt`, or `ps` respectively). The *Gnuplot* manual discusses these options in more detail.

Here is example of a simple plot that makes use of some of the `CURVE` options:

```

MLE
  PLOTFILE(DEFAULTPLOTNAME)
  PLOT("set key bottom left; set y2tics")
    CURVE KEY "sin(x)" AXES "xly1" WITH "lines linetype 3"
      x (0, 2*PI, 100)
      x, SIN(x)
    END
    CURVE KEY "cos(x)" AXES "xly1" WITH "lines linetype 3"
      x (0, 2*PI, 100)
      x, COS(x)
    END
    CURVE KEY "tan(x)" AXES "xly2" WITH "lines linetype 2"
      x (0, 2*PI, 100)
      x, TAN(x)
    END
  END {plot}
END {mle}

```

ERRORBARS

Additional expressions within CURVE...END define things like error bars. *Gnuplot* provides two standards for error bars. If only one additional (error bar) expression exist, that value is taken as a delta value to add and subtract from the y value. If two error bar expressions exist, the values are taken as the minimum and maximum (respectively) values for the error bars.

Here is an example of plotting error bars for a binomial experiment involving 40 observations:

```

MLE
{ -- Plots the probabilities of observing x boys in a families of exactly 5
  children.}
  n = 5      {bernoulli trials -- for families of size 5}
  p = 0.502  {probability of a male child per trial}

  { -- Also plots the standard errors for each outcome assuming that}
  fam = 40 {a sample of fam families are observed}

  PLOTFILE(DEFAULTPLOTNAME)
  PLOT("set yrange [0:]; set xrange [-0.25:" + REAL2STR(n + 0.25, 6, 2) + "]")
    CURVE WITH "errorbars"
      x = 0 TO n
      x                                     {x-axis value}
      PDF BINOMIAL(x) p, n END             {y-axis value}
      Sqrt(p*(1 - p)/fam)                  {errorbar delta}
    END {curve}
  END {plot}
END {mle}

```

Other strings

A series of one or more string expressions can follow the numeric expressions in the CURVE...END. These strings will be appended to the *Gnuplot* plot statement so that plot options or other functions can be plotted. The statements will be written to the plot file. The typical purpose is to re-plot curves in a different style.

Suppose we want to plot the normal distribution with $\mu=0$ and $\sigma=5$ over the range -10 to 10, and also show an 21-point histogram superimposed on the continuous curve. The *mle* code to do this is:

```

MLE
  PLOTFILE(DEFAULTPLOTNAME)    { open a plot file}

  PLOT("set ylabel 'normal pdf f(t)'; set xlabel 't' ")
    CURVE WITH "boxes"
      x (-10 10 21)
        x                      { the x value}
        PDF NORMAL(x) 0, 5 END  { the function to plot}
      ", '' with lines"
    END {do}
  END {plot}
END {mle}

```

Three-dimensional plots

Three-dimensional plots follow the same syntax as do two-dimensional plots, except that both an *<x_var>* and a *<y_var>* must be defined in the CURVE statement along with their ranges. Here is the formal definition for one form:

```

CURVE
  [KEY <keystring> | WITH <withstring> | AXES <axesstring> . . . ]
  <x_var> ( <x_min> <x_max> [ <x_points> ] )
  BY <y_var> (<y_min> <y_max> [ <y_points> ] )
    <x_expr>, <y_expr>, <z_expr> [ <expr> ... ]
  [<string>. . .]
END

```

Note that there is now a variable for both *x* and *y*. The specification for each variable is separated by the keyword BY. If the value of *<x_points>* or *<y_points>* is missing, it will be taken from the variable PLOTPOINTS (which is initially 100).

Alternatively, the INTEGER form of the CURVE statement can be used:

```

CURVE
  [KEY <keystring> | WITH <withstring> | AXES <axesstring> . . . ]
  <x_var> = <x_min> TO <x_max>
  BY <y_var> = <y_min> TO <y_max>
    <x_expr>, <y_expr>, <z_expr> [ <expr> ... ]
  [<string>. . .]
END

```

Additionally, the REAL and INTEGER forms can be combined:

```

CURVE
  [KEY <keystring> | WITH <withstring> | AXES <axesstring> . . . ]
  <x_var> = <x_min> TO <x_max>
  BY <y_var> (<y_min> <y_max> [ <y_points> ] )
    <x_expr>, <y_expr>, <z_expr> [ <expr> ... ]
  [<string>. . .]
END

```

or

```

CURVE
  [KEY <keystring> | WITH <withstring> | AXES <axesstring> . . . ]
  <x_var> ( <x_min> <x_max> [ <x_points> ] )
  BY <y_var> = <y_min> TO <y_max>
    <x_expr>, <y_expr>, <z_expr> [ <expr> ... ]
  [<string>. . .]
END

```

Gnuplot does not support error bars or boxes for three-dimensional plots. Thus, there are three required numeric expression ($\langle x_expr \rangle$, $\langle y_expr \rangle$, $\langle z_expr \rangle$) following the $\langle y_var \rangle$ definition (although additional numeric expressions can be written to the data file for other uses). These three required expressions gives the x , y , and z values to be plotted for each combination of x_var and y_var .

Here is an example of a simple three-dimensional plot. Suppose we want to plot the function $\sin(x)^2 + \cos(y)^2$ over the range 0 to 2π with 30 points in each dimension. The *mle* code to do this is:

```
MLE
  PLOTFILE(DEFAULTPLOTNAME)           { open plot file}

  PLOT("set contour base; set hidden3d" { plot a surface plot & a contour plot}
        "set view 50")                 { change the perspective a bit}
    CURVE x (0, 2*PI, 30) BY y (0, 2*PI, 30) { define the ranges }
      x, y, SIN(x)^2 + COS(y)^2           { the function to plot}
    END {curve}
  END {plot}
END {mle}
```

Three-dimensional plots can include multiple curves. For example, to the previous curve, we can add to the graph, a plane through $z = 1$, and another plane through $z = y/4$.

```
MLE
  PLOTFILE(DEFAULTPLOTNAME)           { open plot file}
  PLOT("set nocontour"                { no contours}
        "set hidden3d"                 { hide lines}
        "set view 50")                 { change the perspective a bit}
    CURVE x (0, 2*PI, 30) BY y (0, 2*PI, 30) { curve 1}
      x, y, SIN(x)^2 + COS(y)^2
    END {curve}
    CURVE x (0, 2*PI, 10) BY y (0, 2*PI, 10) {curve 2}
      x, y, 1
    END {curve}
    CURVE x (0, 2*PI, 10) BY y (0, 2*PI, 10) {curve 3}
      x, y, y/4
    END {curve}
  END {plot}
END {mle}
```

DATA statement

The purpose of a *DATA statement* is to create a series of observations from a file. The series of observations are usually used in the computation of a likelihood. The *DATA*...*END* statement defines the format of the data file, defines variables to be read in, provides a way of transforming variables, and dropping observations. Only an overview of the *DATA* statement is given here; details are given in a chapter devoted to the *DATA* statement.

The format for the *DATA* statement is:

```
DATA
  <variable> [FIELD x [LINE y]] [= <expr>] [DROPIF <expr> | KEEPIF <expr> ...]
  ...
END
```

A list of variables are given within the `DATA...END` block. This list can also specify a set of mathematical transformations on the data as it is being read. A description of each field follows:

- *<variable>* is the name of the variable to be defined. The variable must not already exist. All variables created by the `DATA` statement are defined to be type real. Integer values will be read in from the data file and converted to real number values. Text strings can exist within a fields of a text file, but must not be assigned to a variable.
- *Field*: The term *field* refers to which column within an input file a variable is found in. In the `hammes.dat` file used in Chapter 1, four fields (or columns) existed in the input file. The field specifier must be a positive integer constant.
- *Line*: Sometimes observations are spread across multiple lines. When the `LINE` keyword is used, e.g. `LINE 2`, *mle* keeps track of the maximum number of lines specified this way. Then, *all* observations are assumed to have that number of lines. If the observations each take but one line, the statement `LINE 1` may be dropped—one line per observation is assumed as a default. The line specifier must be a positive integer constant.
- *<=expr>* defines a transformation expression. The expression can refer to the variable being read, or any variables that are defined before the current variable. The line `newvar FIELD 3 = newvar^2` will read `newvar` from field three of the data file. The value of `newvar` is then squared and assigned back to `newvar`.
- *<DROPIF>* provides a mechanism to drop observations. The expression following `DROPIF` must evaluate to `TRUE` or `FALSE`. If `TRUE`, the observation is dropped. The line `newvar FIELD 3 DROPIF newvar <= 0` will drop all observations when the variable in field three is not positive.
- *<KEEPIF>* provides another mechanism to drop observations. The expression following `KEEPIF` must evaluate to `TRUE` or `FALSE`. If `FALSE`, the observation is dropped (that is, not kept). The line `newvar FIELD 3 KEEPIF newvar > 0` will drop all observations for which the variable in field three is not positive. `KEEPIF` and `DROPIF` expressions can be far more complex, but must return `TRUE` or `FALSE`.

Usually, data are read from a data file. The `DATAFILE()` procedure defines and opens this file. Here is an example:

```
DATAFILE("test.dat")
DATA
  o_time      FIELD 1  = o_time*365.25
                  DROPIF (o_time > 100)
  c_time      FIELD 3  = IF c_time = -1 THEN c_time ELSE c_time*365.25 END
  missing     FIELD 4  DROPIF missing_data <> 1
  frequency   FIELD 5  DROPIF frequency <= 0
END
```


The variable names `FREQUENCY` or `FREQ` are taken as frequencies for each observation. (If both variable names are used, `FREQUENCY` is taken as the frequency variable). The frequency of each observation is used to compute a proper likelihood as if multiple lines of identical observations were read. If the `FREQUENCY` or `FREQ` keywords are missing, a frequency of one is assumed for each observation.

The `DATA statement` is used in conjunction with the `DATA function`. Within a `MODEL statement`, you can use the `DATA function` to evaluate the likelihood, one observation at a time. Do not be confused by the fact that there is both a `DATA statement` and a `DATA function`. They complement each other. Simply remember that a `DATA statement` is used as a statement, and there is typically one such statement per program. The `DATA function` can only be used as part of an expression—typically only within the likelihood expression of a `MODEL statement`.

EXIT statement

The `EXIT` statement immediately exits the current procedure or function. When an `EXIT` statement is encountered outside of a procedure or function, the program exits. The statement has no arguments. For example, the code

```
MLE

PROCEDURE dosomething
  WRITELN('Enter f with i')
  z = 5
  WHILE z > 0 DO
    WRITELN('z is ' z)
    IF z = 2 THEN
      EXIT
    END {if}
    z = DEC(z)
  END {while}
  writeln('SHOULD NOT BE HERE')
END {f}

dosomething
writeln('Finished')
END {mle}
```

produces the output

```
Enter f with i
z is 5
z is 4
z is 3
z is 2
Finished
```

FOR statement

The `FOR` statement provides a means of looping through statements. The statement has a number of format. The most basic form of the statement is

```
FOR <var> = <expr> TO <expr> DO
  <statements>
END
```

The variable *<var>* must either not be previously defined or, if it already exists, it must be the same type as the first *<expr>*. The first *<expr>* is executed once and defines the starting value of *<var>*. The value of *<var>* changes as the FOR statement is executed. The second *<expr>* will be executed once and defines the last value of *var*. Here is an example that will print sine and cosine tables in one degree increments as well as creating a table of radians for each degree:

```
r : REAL[0 TO 359]
FOR x = 0 TO 359 DO
  r[x] = DTOR(x)
  WRITELN(x " degrees ( " r[x] " radians): SIN()=" SIN(r[x]) ", COS()=" COS(r[x]))
END
```

STEP

Another form for the FOR statement provides control over the step size. The STEP keyword provides this ability

```
FOR <var> = <expr> TO <expr> STEP <expr> DO
  <statements>
END
```

For example, the statement

```
FOR x = 0 TO 360 STEP 30 DO
  WRITE(x " ")
END
```

produces the output 0 30 60 90 120 150 180 210 240 270 300 330 360. The type of *<var>* with STEP can be either integer or real.

STEPS

Another form of the FOR statement loops between two numbers in a fixed number of steps. The STEPS keyword (which should not be confused with STEP) provides this ability.

```
FOR <var> = <expr> TO <expr> STEPS <expr> DO
  <statements>
END
```

For example, the statement

```
FOR x = 0 TO 360 STEPS 5 DO
  WRITE(x " ")
END
```

produces the output 0.0 90.0 180.0 270.0 360.0. Notice that *x* (and hence the returned values) are of type real.

Array FOR loops

Another form of the FOR statement will loop over each element of an array. This version of the FOR statement has this syntax:

```
FOR <var> = <array> DO
  <statements>
END
```

Notice that the `TO` keyword is not present in the form. For example, consider the following code:

```
myarray:STRING[1 TO 7] = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
FOR x = myarray DO
  WRITE(x " ")
END {for}
Writeln
FOR y = [3+2i, -4i, -2-4i, 3] DO
  WRITE(y " ")
END {for}
```

The result of this code is

```
Sun Mon Tue Wed Thu Fri Sat
3.0+2.0i 0.0-4.0i -2.0-4.0i 3.0+0.0i
```

FUNCTION declaration (user defined)

The function statement permits a user to define additional functions, in effect, allow you to extend the *mle* language.

Once a function has been defined, it can be called in the same way that any predefined function is called: the name and argument list is given in the same context as an expression. Arguments can be optionally be passed to the function.

The syntax for a procedure definition (without arguments) is

```
FUNCTION <name>:<type>
  <statements>
END
```

And a function with arguments is

```
FUNCTION <name>([var] <variable>:<type>...) : <type>
  <statements>
END
```

The *<name>* is an ordinary user-defined name, like those used for variables. Following the name and (optional) argument list comes a colon and a *<type>*. This type can be any type that defines a variable (REAL, INTEGER, BOOLEAN, etc.). Additionally, a function can return an array (of any dimension) of the basic types.

A list of *<statements>* comes between the end of the procedure definition and the final `END`. These statements can even include other “private” procedure definitions, only visible within the original procedure. Here is an example of a user-defined function.

```
FUNCTION CircleArea(radius:REAL):REAL
{ Returns the area of a circle}
IF radius < 0 THEN
  Writeln('Error: negative radius passed to CircleArea')
  RETURN = 0
ELSE
  RETURN = PI*radius^2
END {if}
END {CircleArea}
```

The expression `CircleArea(1)` returns the value 3.14159...

Argument list

The argument list, if any, follows the name of the function. In the above example, one argument is passed to the function. When the function is called, a single real or integer argument must be passed to the function.

The variables in the argument list is defined locally—that is, they are defined only within the body of the function. Any variable by the same name outside of the function is “masked” by the new, local version of the variable. Likewise, any variable declared within the body of the function is locally defined. The `VAR` keyword can change this procedure. See the **Argument list** section of **PROCEDURE declaration**, page 131.

The RETURN variable

Inside a function, the local variable `RETURN` is automatically created. The variable is created as the same type as defined for the function. The variable can be used as an ordinary variable. When the function exits, the value of `RETURN` is passed back to the expression from which the function was called.

Recursion

User-defined functions in *mle* support full recursion. That is, a function can call itself, invoking a new copy of local parameters and variables within each new nested call. Below is a simple example of a recursive function to compute the factorial function. (This is not the best way to compute $n!$, but the example is useful to demonstrate recursion).

```
FUNCTION MyFactorial(n:INTEGER):INTEGER
  IF n < 0 THEN
    WRITELN('Error: negative value passed to MyFactorial')
    RETURN = 0
  ELSEIF n <= 1 THEN
    RETURN = 1
  ELSE
    RETURN = n*MyFactorial(n - 1)
  END {if}
END {MyFactorial}

FOR x = 0 TO 10 DO
  WRITELN(x, '! = ', MyFactorial(x))
END {for}
```

The `FOR` loop following the function gives the result:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

IF statement

The IF statement provides a means of conditionally executing statements. The following types of IF statements are available:

```
IF <bexpr> THEN
  <statements>
END
```

This form will conditionally execute the <statements> only if <bexpr> evaluates to TRUE. An ELSE clause can be added to the statement so that one of two sets of statements will always be executed:

```
IF <bexpr> THEN
  <statements>
ELSE
  <statements>
END
```

In addition, one or more ELSEIF clauses can be added to the statement to allow multiple conditions to be tested:

```
IF <bexpr> THEN
  <statements>
ELSEIF <bexpr> THEN
  <statements>
ELSEIF <bexpr> THEN
  <statements>
ELSE
  <statements>
END
```

Here is an example of using the IF statement:

```
IF SYSTEM = "MS-DOS" THEN
  PRINTLN("Run from an MS-DOS system")
  SEP = '\ '
  DATAFILE("C:" + SEP + DIR + SEP + NAME)
ELSE
  PRINTLN("Run on a unix system")
  SEP = '/'
  DATAFILE(DIR + SEP + NAME)
END
```

INCLUDE statement

The INCLUDE statement is used to include statements from other files. The statement provides a mechanism for including procedure and functions from libraries. The syntax is

```
INCLUDE <name>
```

The <name> is a string constant. This is because <name> is immediately opened and included as code in the current program. Files can be recursively included, and is only limited by the operating system's limitations on the number of files that can be simultaneously opened

mle uses the following process to find *<name>*. First, it checks to see if *<name>* exists without modification. Next, the set of paths specified by the *-I* command line option are searched for *<name>*. Finally, the set of paths specified by the operating system environment variable *MLELIBS*. Here are some examples.

The statement

```
INCLUDE "C:\mymlplib\fft.mle"
```

will continue parsing the file *C:\mymlplib\fft.mle*, and then continue parsing the current file. The statement

```
INCLUDE "fft.mle"
```

will first try opening *fft.mle* in the current directory.

If this file does not exist, the *mle* will check through directories specified by the command line option (if any) *-I*. Suppose the command line was specified as *-I C:\include\mle;C:\mymlplib*. *mle* will check each of these two directories for the file *fft.mle*.

Finally, if *fft.mle* is still not found, *mle* will check all directories in the operating system environment variable *MLELIBS*. Suppose this variable is set as "set *MLELIBS*=*C:\include\mle;C:\proj\math*". Then, *mle* will check both directories for the file *fft.mle*. An error results if the file is still not found.

Be aware that different operating systems have different ways of separating lists of directories. The above examples apply to DOS or Windows operating systems. In Unix, a colon is used to separate a list of directories. Hence, the previous command line would be given as *-I \include\mle:~\mymlplib*.

The *INCLUDE* statement helps create reusable software modules (procedures and functions). One helpful hint when developing modules is to always begin the included file with a *BEGIN ... END* statement. That way, any unclosed statements will result in an error message before the parser leaves the include file. For example,

```
BEGIN
{This include file has a procedure for computing a random seed based on the
 system time and date}

PROCEDURE randomseed
  a1 : INTEGER
  a2 : INTEGER
  a3 : INTEGER
  a4 : INTEGER
  GETTIME(a1, a2, a3, a4)
  a4 = a1 + a2 + a3 + a4
  GETDATE(a1, a2, a3)
  seed(a1 + a2 + a3 + a4)
END {randomseed}

END
```

MODEL statement

The `MODEL...RUN...END` statement defines the underlying probability model used by *mle*, defines the parameters to be found for the model, and defines constraints under which parameters are to be estimated. Only an overview of the `MODEL` statement is given here. The **MODEL** chapter in the *User's Manual* contains more details and examples.

The basic structure of the `MODEL` statement looks like this:

```
MODEL
  <expression>
RUN  [THEN <statements> END]
  <run specifications>
END
```

Between `MODEL` and `RUN` is a single expression that is the likelihood. Within the likelihood is one or more `PARAM...END` functions. These define the parameters, whose values will be found for maximizing the likelihood. One of the most important aspects of learning *mle* is the design and construction of the expression for the likelihood.

Run specification

The `<run specifications>` is given between the `RUN` and the `END` part of the `MODEL` statement, this provides a way of evaluating the full model as well as a series of nested or reduced models. Parameters can be constrained for the purpose of hypothesis testing or otherwise modifying the model. Parameters may be held constant, or fixed to the value of another parameter. These are called *fixed parameters*, and an estimate will not be found for them when the likelihood is maximized. The runlist in *mle* provides the mechanism for fixed parameters primarily to reduce models from more complicated to simpler forms. Essentially, this `<run specification>` list gives instructions on how to run the model and the various sub-parameterizations of the model. There are three keywords:

- **FULL**: If all of the parameters (defined by `PARAM...END` functions in the model) are to be found, a simple `FULL` command is placed between the `RUN` and its matching `END`.
- **REDUCE**: Reduced models, where one or more parameters are constrained to a constant or another parameter, are specified as `REDUCE` followed with a list of one or more "reductions". Any number of `REDUCE` commands (along with one `FULL`) can be used in a single model. For example, you might wish to find all parameters, then constrain a parameter called `beta1` to be zero, and then `beta2` to zero, and then constrain both to zero. Then you would put the following specify

```
MODEL
...
RUN
  FULL                                {Run the full model}
  REDUCE beta1=0                      {Constrain beta1 to 0}
  REDUCE beta2=0                      {Constrain beta2 to 0}
  REDUCE beta1=0 beta2=0              {Constrain betas to 0}
END
```

- **WITH**: Specifies a list of parameters to contain in the model. The `WITH` keyword is followed by a list of model parameters. Each parameter included in the list will be

included in the model. Variables *not* included will be constrained to their TEST value (this value is set in the PARAM function, or is zero if unspecified). Here is an example of using a WITH runlist

```
MODEL
...
RUN
  WITH mu, sigma, betal, beta2      {This is the full model}
  WITH mu, sigma, beta2             {Constrain betal to 0}
  WITH mu, sigma, betal             {Constrain beta2 to 0}
  WITH mu, sigma                   {Constrain betas to 0}
END
```

The WITH keyword has an additional “trick”. You can include a list of parameters in parentheses. All models will be formed from the set of variables in parenthesis as well as the variables NOT in parenthesis that are included in all models. Hence the above MODEL statement could be more economically written

```
MODEL
...
RUN
  WITH mu, sigma (betal, beta2)
END
```

The same four versions of the model will be created and run. The above WITH is equivalently written WITH mu (betal) sigma (beta2) or WITH (betal) (beta2) mu sigma.

It is important to realize that a *single* WITH list with N parameters will generate a total of 2^N models. To put that in perspective, a list of two variables will result in four models being estimated. For ten parameters, 1024 models will be estimated. For 15 variables 32,768 models will be estimated!

Results

Results of model estimation are written to the output file, specified by the OUTFILE function. A number of variables control the format of the output.

A report with estimated standard errors is printed when PRINT_SE = TRUE. *mle* uses two different estimates for the variance-covariance matrix. One or both methods may be used by setting INFO_METHOD1=TRUE or INFO_METHOD2=TRUE. The default method (INFO_METHOD1) computes the variance and covariance matrix by inverting Nelson's (1983) approximation to the Fisher's information matrix. The second estimate of the variance-covariance matrix (INFO_METHOD2) is computed by estimating the second partial derivative by numeric perturbation. This method is less accurate than the first method in some cases. Nevertheless, when hierarchical likelihoods are being computed, this method will produce better estimates.

Whenever standard error are reported, a variance-covariance matrix will be estimated. The matrix is printed when PRINT_VCV = TRUE. By default PRINT_VCV = FALSE.

An approximate confidence region for each parameter can be estimated by *mle*. For multivariate models, the regions are only valid if the parameters are

uncorrelated, or if there is only a single parameter. The report is printed when `PRINT_CI = TRUE`.

When the variable `PRINT_SHORT = TRUE`, the report formats are modified so that all parameters estimates are printed on a single line. This is useful when results are to be subsequently processed by a computer program or read into a spreadsheet.

The SDF, PDF, and hazard function can be tabulated in the output for all predefined PDFs used in a model by setting `PRINT_DISTS = TRUE`. The tabulation starts at value `DIST_T_START`, ends at the value `DIST_T_END`, and is tabulated for `DIST_T_N` equally spaced points. The distributions are tabulated at the mean value of covariates. Likewise, plots of distributions and likelihood surfaces can be generated. See the **PLOT** chapter of the *User's Manual* for details.

mle provides a mechanism for accessing results from previous runs either within or outside of the `MODEL` statement. Each `MODEL` statement is numbered (beginning with 1) in the order in which they are found in the program. Furthermore, each run of the model, defined by the `FULL` or `REDUCE` statement, is numbered beginning with 1 for each `MODEL`. The following variables are created:

```
<param>.<m>.<r>
<param>.LOW.<m>.<r>
<param>.HIGH.<m>.<r>
<param>.START.<m>.<r>
<param>.UCI.<m>.<r>
<param>.LCI.<m>.<r>
<param>.SE.<m>.<r>
LOGLIKELIHOOD.<m>.<r>
FREE_PARAMS.<m>.<r>
DELTA_LL.<m>.<r>
ITERATIONS.<m>.<r>
EVALS.<m>.<r>
VCV_EVALS.<m>.<r>
CI_EVALS.<m>.<r>
INVERTFLAG.<m>.<r>
CONVERGENCE.<m>.<r>
VCV.<m>.<r>
```

where `<m>` is the model number and `<r>` is the run number for the model, and `<param>` is the name for a free parameter in the model. Each `VCV.<m>.<r>` is an $n \times n$ matrix where n is the number of free parameters, which is available in `FREE_PARAMS.<m>.<r>`. The variable `INVERTFLAG.<m>.<r>` is a boolean variable that specifies whether or not the variance-covariance matrix was inverted without error.

Each `CONVERGENCE.<m>.<r>` variable has an integer value that takes on a value given in Table 4.

Table 4. Meaning of the CONVERGENCE variable.

Value	Meaning
0	Not done
1	Stopped after maximum function evaluations
2	Stopped after maximum number of iterations
3	Stopped after maximum time
4	Stopped by termination file
5	Converged normally
6	Trouble converging in one dimension
7	Starting value is not within min and max bounds
8	Starting temperature is not positive
9	Did not converge

PLOT statement

The `PLOT...END` statement initiates a single graph or chart. The statement does not itself define the particular curves to be plotted on the plot; instead, each `CURVE...END` statement executed within the `PLOT...END` statement will add a single curve to the plot (see the `CURVE` statement, page 114).

The format of the statement is

```
PLOT [( <string_expr> . . . )]
<statements>
END
```

When a `PLOT` statement is executed, a few statements may be written to the plot file. Then the `<statements>` are executed. All `CURVE` statements executed before the `END` is reached will result in one curve being added to the current plot.

The optional series of string expressions (enclosed within parentheses) can immediately following the `PLOT` statement. These strings will be written to the plot file. The purpose of these strings is to provide additional information to the *Gnuplot* program, such as titles, ranges, and borders. They are simply written verbatim to the plot file. In fact, plots can be written in the *Gnuplot* language with these strings. Here is an example:

```
MLE
PLOTFILE("gploteg.plt")
PLOT ( "plot [0:2*pi][-5:5] sin(x), cos(x), tan(x)" ) END
END
```

The `PLOT` statement writes the `PLOTINIT` string to the plot file. You can assign a string to the `PLOTINIT` variable, and it will be written for each `PLOT`.

PROCEDURE declaration

The procedure statement permits a user to define “subroutines.” These procedures, in effect, allow you to extend the *mle* language.

Once a procedure is initially defined, the procedure is then called in the same way that predefined procedures are called: the name and argument list is given in the same context as a statement. Arguments can be optionally be passed to the procedure.

The syntax for a procedure definition (without arguments) is

```
PROCEDURE <name>
  <statements>
END
```

And a procedure with arguments is

```
PROCEDURE <name>([var]<variable>:<type>...)
  <statements>
END
```

The *<name>* is an ordinary user-defined name, like those used for variables. A list of *<statements>* comes between the end of the procedure definition and the final *END*. These statements can even include other “private” procedure definitions, only visible within the original procedure. Here is an example of a simple procedure.

```
PROCEDURE GiveErrorMessage(numb:INTEGER mess:STRING)
{ Reports an error number and message, offers user a chance to stop}
ans:STRING {variable defined only within procedure}
WRITELN('Error number: ', numb, ' has occurred')
WRITELN(mess)
REPEAT
  WRITE('Continue [yes/no]? ')
  READLN(ans)
  ans = TOLOWER(ans)
  IF ans == 'no' THEN
    WRITELN('Bye')
    HALT {halt the program}
  ELSEIF ans == 'yes' THEN
    EXIT {exit this procedure}
  END {if}
UNTIL FALSE {loop forever}
END {GiveErrorMessage}
```

Argument list

The argument list, if any, follows the name of the procedure. In the above example, two arguments are passed to the procedure when it is called. The first argument must be an integer and the second a string. For example, suppose the procedure is called with the following actual arguments: `GiveErrorMessage(14, 'Time is up!')`. Within the procedure, the value 14 would be assigned to the variable `numb`, and `'Time is up!'` would be assigned to `mess`.

The variables in the argument list are defined locally—that is, they are defined only within the body of the procedure. Any variable by the same name outside of the procedure is “masked” by the new, local version of the variable. Likewise,

any variable declared within the body of the procedure (like the string `ans`) is locally defined.

Another important aspect of arguments is that changing values of the argument will only affect the variables within the procedure. That is, the arguments are passed to the procedure as, in effect, new variables. This behavior can be changed for each variable by preceding the name of the variable by the keyword `VAR`.

For example, the variable declaration `VAR response:integer` added to an argument list effectively “links” the variable used to call the procedure to the variable within the procedure. In fact, the name of the variable in the procedure call need not be the same as the variable within the procedure. Here is a simple example:

```
PROCEDURE DoSomething(VAR numb1:INTEGER, numb2:INTEGER)
  a:STRING = "something"
  b:REAL = 7.2
  INC(numb1)
  INC(numb2)
END {DoSomething}

a:INTEGER = 5
b:INTEGER = 10

DoSomething(a, b)
WRITELN('a is ', a, ' and b is ', b)
```

This code fragment results in the output “a is 6 and b is 10”. The value of `b` did not change because the `VAR` keyword was missing. Notice that within `DoSomething`, new variables `a` and `b` are declared.

Recursion

User-defined procedures (and functions) in *mle* support full recursion. That is, a procedure can call itself, invoking a new copy of local parameters and variables within each new nested call. Below is a simple example of a recursive function

```
PROCEDURE MyRecursive(r:REAL)
  IF r < 10 THEN
    MyRecursive(r + 1)
  ELSE
    WRITELN('r is ', r)
  END {if}
END {MyRecursive}

MyRecursive(1)
```

The call on the last line will result in a cascade of 10 calls to `MyRecursive` before finally printing the value 10.0.

Procedure call

A call to a procedure is a type of statement. Procedures are single word commands that include zero or more arguments. These procedures perform some specific task given a list of arguments. A number of intrinsic procedures are

available in *mle*. Additionally users can define new procedures (see PROCEDURE declaration, page 131).

An example of a procedure is the statement `DATAFILE("hammes.dat")` found in the example in Chapter 1 of the *User's Manual* defines and opens up the datafile used by the `DATA` statement. A list of all procedures, with examples, can be found in the **PROCEDURES** chapter. Here are some examples:

```
DATAFILE("hammes.dat")
OUTFILE("hammes.out")
SEED(9734) {Set a random number seed}
```

REPEAT statement

The `REPEAT` statement provides a means of looping through statements until some condition is met. The format is

```
REPEAT
  <statements>
UNTIL <bexpr>
```

The `<statements>` are executed and then the boolean expression `<bexpr>` is evaluated. If the result is `FALSE`, the loop repeats and `<statements>` are executed again. When `<bexpr>` evaluates to `TRUE`, the loop terminates. A `REPEAT` statement always executes the `<statements>` at least once.

WHILE statement

The `WHILE` statement provides a means of looping through statements while some condition is met. The format is

```
WHILE <bexpr> DO
  <statements>
END
```

The boolean expression `<bexpr>` is executed first. If the value is `TRUE`, the `<statements>` are executed once and `<bexpr>` is evaluated again. The sequence continues until `<bexpr>` evaluates to `FALSE`. That is, when `<bexpr>` is `FALSE`, the loop terminates.

Other looping statements are the `FOR` statement (page 121) and the `REPEAT` (page 133) statement

Chapter 8

Procedures

There are a few intrinsic procedures available in *mle*. Procedures are single word commands that include zero or more arguments. Procedures perform some task given the list of arguments. Procedures do not return a value the way a function does. The following sections describes the procedures available in *mle*.

Built-in procedures

CHDIR(s)

Purpose: Changes into the directory *s*.

Arguments: *s* is a string that contains the name of a directory.

Examples: `CHDIR("C:\TEMP")` `WRITELN(GETDIR)` writes C:\TEMP

Notes: File and directory names are case sensitive on Unix systems. They are not case sensitive on DOS and Windows systems.

See also: `MKDIR`, `RMDIR`, function `GETDIR`, function `DIREXISTS`

CLOSE(f)

Purpose: Closes file *f*.

Arguments: *f* is a previously opened file. *f* must be defined as a `FILE` variable. Additionally, there are several predefined file variables: `FOUTPUT` is the standard output file. `FOUTFILE` is the file opened with the `OUTFILE()` procedure. `FINPUT` is the standard input (the terminal by default). `FDATA` is the data file usually opened with `DATAFILE()`.

Examples: `CLOSE(FOUTFILE)`
`CLOSE(myfile)`

See also: `OPENREAD`, `OPENWRITE`, `FLUSH`, `OUTFILE`, `DATAFILE`, function `EOF`

DATAFILE(s)

Purpose: Opens up a data file for the `DATA` statement. Closes any previous data files.

Arguments: A single string expression

- Examples: DATAFILE("mydata.dat")
DATAFILE(filename + '.' + datextension)
- See also: OUTFILE
- DATATOARRAY(d1, d2,...)*
- Purpose: Converts a variable array created with a DATA statement to an ordinary array.
- Arguments: A list of one or more variables that were created by a DATA statement
- Examples: DATATOARRAY(id, missing)
- See also: DATA statement
- DEC(i)*
- Purpose: Decrements variable i.
- Arguments: *i* is an INTEGER variable. *i* must be a variable name, not a function, as *i* will be updated with the new value.
- Examples: DEC(x)
- See also: INC, function INC
- DUMPSYMBOL(s)*
- Purpose: Gives symbol table information about symbol *s*.
- Arguments: *s* is the name of a variable, user-defined procedure or function.
- See also: DUMPTABLE
- DUMPTABLE*
- Purpose: Dumps the entire symbol table.
- Arguments: none.
- See also: DUMPSYMBOL
- ERASE(s)*
- Purpose: Erases the file named *s*.
- Arguments: *s* is a string variable.
- Examples: ERASE("FNAME.TMP")
ERASE("C:\TEMP\SAV.TMP")
- Notes: The file names are not case sensitive in DOS and Windows systems. File names are case sensitive in Unix systems
- See also: RMDIR, function EXISTS
- EXEC(s1, s2)*
- Purpose: Executes via the operating system command *s1* with arguments *s2*
- Examples: EXEC('command.com', '/c dir') executes the DOS dir command.
EXEC('sort.exe', '< f.dat > fs.dat') executes the DOS sort command.

`EXEC('ls', '-l')` executes the Unix `ls` command.

Notes: The `EXEC` function works like the `EXEC` procedure except that it returns an error code.

See also: Function `EXEC`

FINISHPLOT(b)

Purpose: Closes and tries to display the current plot with *Gnuplot*.

Arguments: *b* is a boolean expression. If *b* is true, a *Gnuplot* `pause -1` statement is written to the plotfile. This causes the graph to be displayed until some action is taken (key hit or click, depending on the device). If the argument is `FALSE`, the pause statement is not written to the plotfile.

Examples: `FINISHPLOT(TRUE) {will pause}`
`FINISHPLOT(FALSE) {will not pause}`

Notes: This procedure is useful for displaying a plot from within *mle*, or automatically creating output for different graphics devices from within *mle*. After closing the plotfile, the *Gnuplot* program is executed with plotfile as its argument. This causes the plot to be written to whatever terminal is defined. For example, if the command `set terminal windows` (Windows) or `set terminal x11` (Unix) is specified in the plotfile, the graph will be displayed on the screen. Other drivers will cause the plot to be written to the file defined by a *Gnuplot* `set output` command.

In order to execute the *Gnuplot* program, *mle* first uses the variable `GNUPLOT` as the name of the executable program. By default `GNUPLOT="gnuplot.exe"` on DOS and Windows operating systems and `GNUPLOT="gnuplot"` on Unix systems. You can change this variable to specify the full pathname and executable name for the *Gnuplot* command, e.g. `GNUPLOT="C:\gnuplot\gnuplot.exe"`. If the `GNUPLOT` variable does not point to a valid name, *mle* next checks for an environment variable called `GNUPLOT`, and tries to find the corresponding program. If that fails, *mle* uses the environment variable `PATH` to find the filename specified by the *mle* variable `GNUPLOT`. Finally, if that fails, *mle* uses the environment variable `PATH` to find the filename specified by the `GNUPLOT` environment variable.

See also: `OPENPLOTFILE`, `WRITEPLOT`, `WRITEPLOTLN`, `PLOT` statement

FLUSH(f)

Purpose: Flushes any buffered output for file *f*.

Arguments: *f* is a previously opened file. *f* must be defined as a `FILE` variable. Additionally, there are several predefined file variables: `FOUTFILE` is the standard output file. `FOUTFILE` is the file opened with the `OUTFILE()` procedure.

Examples: `FLUSH(FOUTFILE)`
`FLUSH(myfile)`

See also:	OPENREAD, OPENWRITE, CLOSE, OUTFILE, DATAFILE
<i>GETDATE</i> (<i>y, m, d</i>)	
Purpose:	Returns the current date.
Arguments:	Requires from 1 to 3 integer variable arguments. The first will take on the current year, the second the month and the third the day of the month.
Examples:	<i>GETDATE</i> (<i>year</i>) Returns the year <i>GETDATE</i> (<i>year, month</i>) Returns the month and the year <i>GETDATE</i> (<i>year, month, day</i>) Returns the year, month and day
See also:	GETTIME
<i>GETTIME</i> (<i>h, m, s, s100</i>)	
Purpose:	Returns the current time.
Arguments:	Requires from 1 to 4 integer variable arguments; (1) current hour, (2) the current minute, (3) the current second day, and (4) the current hundreds of a second.
Examples:	<i>GETTIME</i> (<i>hour, min, sec, s100</i>) Returns the current time down to the 100 th of a second. <i>GETTIME</i> (<i>hour, min</i>) Returns the current hour and minute
See also:	GETDATE
<i>HALT</i>	
Purpose:	Terminates execution of the program.
Arguments:	None
<i>INC</i> (<i>i</i>)	
Purpose:	Increments variable <i>i</i> .
Arguments:	<i>i</i> is an <code>INTEGER</code> variable. <i>i</i> must be a variable name, not a function, as <i>i</i> will be updated with the new value.
Examples:	<i>INC</i> (<i>x</i>)
See also:	DEC, function <code>INC</code>
<i>MKDIR</i> (<i>s</i>)	
Purpose:	Creates the directory <i>s</i> .
Arguments:	<i>s</i> is a string that contains the name of a directory.
Examples:	<i>MKDIR</i> (" <code>\TEMP</code> ") creates the directory <code>\TEMP</code> on the current drive
Notes:	File and directory names are case sensitive on Unix systems. They are not case sensitive on DOS and Windows systems.
See also:	CHDIR, RMDIR, function <code>GETDIR</code> , function <code>DIREXISTS</code>

OPENAPPEND(f, s)

- Purpose: Opens text file *f*, named *s*, for appending. The file will be opened and set for writing just after the last character in the file. The file is appended with `WRITELN(f, <var1>, ...)`.
- Arguments: *f* is a FILE variable. *s* is a string that is the name of the file to append.
- Examples: `OPENAPPEND(fnames, "names.dat")`
- See also: `OPENREAD`, `OPENWRITE`, `CLOSE`, `WRITE`, `WRITELN`

OPENREAD(f, s)

- Purpose: Opens text file *f*, name *s*, for reading. The file is read with `READLN(f, <var1>, ...)`.
- Arguments: *f* is a FILE variable. *s* is a string that is the name of the file to open.
- Examples: `OPENREAD(fnames, "names.dat")`
- See also: `OPENWRITE`, `OPENAPPEND`, `CLOSE`, `READ`, `READLN`

OPENWRITE(f, s)

- Purpose: Opens text file *f*, named *s*, for writing. The file will be opened and set to the beginning of the file (if it exists) or is newly created for writing. The file is written with `WRITELN(f, <var1>, ...)`.
- Arguments: *f* is a FILE variable. *s* is a string that is the name of the file to create or overwrite.
- Examples: `OPENWRITE(fnames, "names.dat")`
- See also: `OPENREAD`, `OPENAPPEND`, `CLOSE`, `WRITE`, `WRITELN`

OUTFILE(s)

- Purpose: Opens up an outfile to which analytical results are printed. Closes any previous outfile. The DATA and MODEL statements report results to the outfile. The sequence of operations corresponds to: `CLOSE(FOUTFILE) OPENWRITE(FOUTFILE, s)`
- Arguments: *s* is a string that is the name of the outfile. The function `DEFAULTOUTNAME` is useful for picking a reasonable name for the outfile.
- Examples: `OUTFILE("mydata.out")`
`OUTFILE(DEFAULTOUTNAME)`
- Notes: File and directory names are case sensitive on Unix systems. They are not case sensitive on DOS and Windows systems.
- See also: `DATAFILE`, Function `DEFAULTOUTNAME`

PLOTFILE(s)

- Purpose: Opens up a file to which a Gnuplot program is written. Also closes any previous plotfile. This procedure must be called before a `PLOT` statement can be executed, a `SURFACE` can be executed, a `WRITEPLOT`, `WRITEPLOTLN` or `FINISHPLOT` procedure can be called.

Arguments: *s* is a string that is the name of the plotfile. The function DEFAULTPLOTNAME is useful for picking a reasonable name for the plotfile. Names should end in .plt for consistency with Gnuplot

Examples:
 PLOTFILE("mygraph.plt")
 PLOTFILE(DEFAULTPLOTNAME)

Notes: File and directory names are case sensitive on Unix systems. They are not case sensitive on DOS and Windows systems.

See also: Function DEFAULTPLOTNAME, procedures WRITEPLOT, WRITEPLOTLN and FINISHPLOT, statements PLOT, MODEL (SURFACE option).

PRINT(*a1, a2,...*)

Purpose: Prints variables to the outfile without including a carriage return at the end.

Arguments: Any number of arguments of any type except FILE.

Examples:
 PRINT("The value of x is ", x)
 PRINT("Sin(x) squared is ", SIN(x)^2)

See also: PRINTLN, OUTFILE, WRITE, WRITELN

PRINTLN(*a1, a2,...*)

Purpose: Prints variables to the outfile file and includes a carriage return at the end.

Arguments: Any number of arguments of any type except FILE.

Examples:
 PRINTLN("The value of x is ", x)
 PRINTLN("Sin(x) squared is ", SIN(x)^2)

See also: PRINT, OUTFILE, WRITE, WRITELN

PTRANSFORM(*v1, v2, e*)

Purpose: Transforms one or more MODEL parameters into a new value and its standard error. The value of *e* is evaluated and assigned to *v1*. The standard error of *v1* is assigned to *v2*.

Examples: The following MODEL statement will estimate a logistic regression and then print a table of probabilities with standard errors.

```
MODEL {returns standard errors for a logistic regression}
DATA
  PDF BERNOULLITRIAL(outcome)
  PARAM p LOW = -100 HIGH = 100 START = 0 FORM = LOGISTIC
  COVAR age PARAM b_age LOW = -100 HIGH = 100 START = 0 END
  END {param}
  END {bernoullitrial}
END {data}
RUN
FULL THEN
  prob : REAL
  SEprob:REAL
  FOR x = 1 to 50 DO
    PTRANSFORM(prob, SEprob, LOGISTIC(p + x * b_age))
    WRITELN(x, ' ', prob, ' ', SEprob)
  END {for}
END {full then}
END {model}
```

Notes:	<p>This procedure must be called within a <code>MODEL . . .END</code> statement. Furthermore, the variance-covariance matrix must be computed and cannot be singular. The expression e must be a full expression that includes at least one of the free model parameters (or else the standard error will be zero).</p> <p>This procedure computes the standard error by first computing (numerically) an array of partial derivatives of the expression with respect of each of the N free parameters in the model, $\mathbf{d} = (\delta x/\delta p_1, \delta x/\delta p_2, \dots \delta x/\delta p_N)$. Next, the standard error is computed as $\sqrt{\mathbf{d}'\mathbf{V}\mathbf{d}}$, where \mathbf{V} is the variance-covariance matrix.</p>
See also:	Statements <code>MODEL</code> and <code>PLOT</code> , function <code>SETRANSFORM</code>
<i>READ(a1, a2,...)</i>	
Purpose:	Reads variables from standard input, without reading a carriage return at the end of line.
Arguments:	Any number of <code>REAL</code> , <code>INTEGER</code> , <code>STRING</code> and <code>CHAR</code> arguments. <code>FILE</code> arguments will result in the remaining variables being read to the specified file.
Examples:	<pre>READ(x, y, z) READ(foutput, x, y, z) { reads from the keyboard}</pre>
See also:	<code>READLN</code> , <code>WRITE</code> , <code>WRITELN</code> , <code>EOLN</code> , <code>EOF</code>
<i>READLN(a1, a2,...)</i>	
Purpose:	Reads variables from standard input through the end of the current line.
Arguments:	Any number of <code>REAL</code> , <code>INTEGER</code> , <code>STRING</code> and <code>CHAR</code> arguments. <code>FILE</code> arguments will result in the remaining variables being read to the specified file.
Examples:	<pre>READLN(x, y, z) READLN(foutput, x, y, z) { reads a line from the keyboard}</pre>
See also:	<code>READ</code> , <code>WRITE</code> , <code>WRITELN</code>
<i>RENAME(s1, s2)</i>	
Purpose:	Renames the file named $s1$ to $s2$.
Arguments:	$s1$ and $s2$ are both string variables.
Examples:	<code>RENAME("FNAME.TMP" , "FNAME.SAV")</code>
Notes:	The file names are not case sensitive in DOS and Windows systems. File names are case sensitive in Unix systems
See also:	<code>ERASE</code> , function <code>EXISTS</code>
<i>RMDIR(s)</i>	
Purpose:	Deletes the directory named s .
Arguments:	s is a string that contains the name of a directory.

Examples:	<code>RMDIR("\TEMP")</code> deletes the directory \TEMP on the current drive
Notes:	File and directory names are case sensitive on Unix systems. They are not case sensitive on DOS and Windows systems.
See also:	<code>MKDIR</code> , <code>CHDIR</code> , function <code>GETDIR</code> , function <code>DIREXISTS</code>
<i>SEED(i)</i>	
Purpose:	Seeds the random number generator.
Arguments:	A single positive integer argument.
Examples:	<code>SEED(13234)</code> <code>SEED(x)</code>
Notes:	Procedure <code>GETTIME</code> can be used to generate pseudorandom seeds.
See also:	<code>GETTIME</code> , functions <code>RAND</code> , <code>IRAND</code> , <code>RRAND</code> , variable <code>RANDOMSEED</code>
<i>WRITE(a1, a2,...)</i>	
Purpose:	Writes a message to the terminal without including a carriage return at the end.
Arguments:	Any number of arguments of any type. <code>FILE</code> arguments will result in the remaining variables being written to the specified file.
Examples:	<code>WRITE("The value of x is ", x)</code> <code>WRITE("Sin(x) squared is ", SIN(x)^2)</code> <code>WRITE(foutput, "Hello there")</code> {writes to the screen}
See also:	<code>WRITELN</code> , <code>OUTFILE</code> , <code>PRINT</code> , <code>PRINTLN</code> , function <code>PUT</code>
<i>WRITELN(a1, a2,...)</i>	
Purpose:	Writes a message to the terminal and includes a carriage return at the end.
Arguments:	Any number of arguments of any type.
Examples:	<code>WRITELN("The value of x is ", x)</code> <code>WRITELN("Sin(x) squared is ", SIN(x)^2)</code> <code>WRITELN(foutput, "Hello there")</code> {writes the line to the screen}
See also:	<code>WRITE</code> , <code>OUTFILE</code> , <code>PRINT</code> , <code>PRINTLN</code> , function <code>PUT</code>
<i>WRITEPLOT(a1, a2,...)</i>	
Purpose:	Writes a message to the plotfile without including a carriage return at the end.
Arguments:	Any number of arguments of any type except <code>FILE</code> .
Examples:	<code>WRITEPLOT("set data style ", x)</code>
See also:	<code>WRITEPLOTLN</code> , <code>PLOTFILE</code> , <code>WRITE</code> , <code>WRITELN</code> , <code>PRINT</code> , <code>PRINTLN</code>
<i>WRITEPLOTLN(a1, a2,...)</i>	
Purpose:	Writes a message to the terminal and includes a carriage return at the end.

Arguments: Any number of arguments of any type.

Examples: `WRITEPLOTLN("set data style ", x)`

See also: `WRITEPLOT`, `PLOTFILE`, `WRITE`, `WRITELN`, `PRINT`, `PRINTLN`

Chapter 9

Continuous probability density functions

This chapter describes the continuous probability density functions that are defined in *mle*. Continuous PDFs are defined for real values of “time” (or t).

Notation and format

The following type of information is typically given:

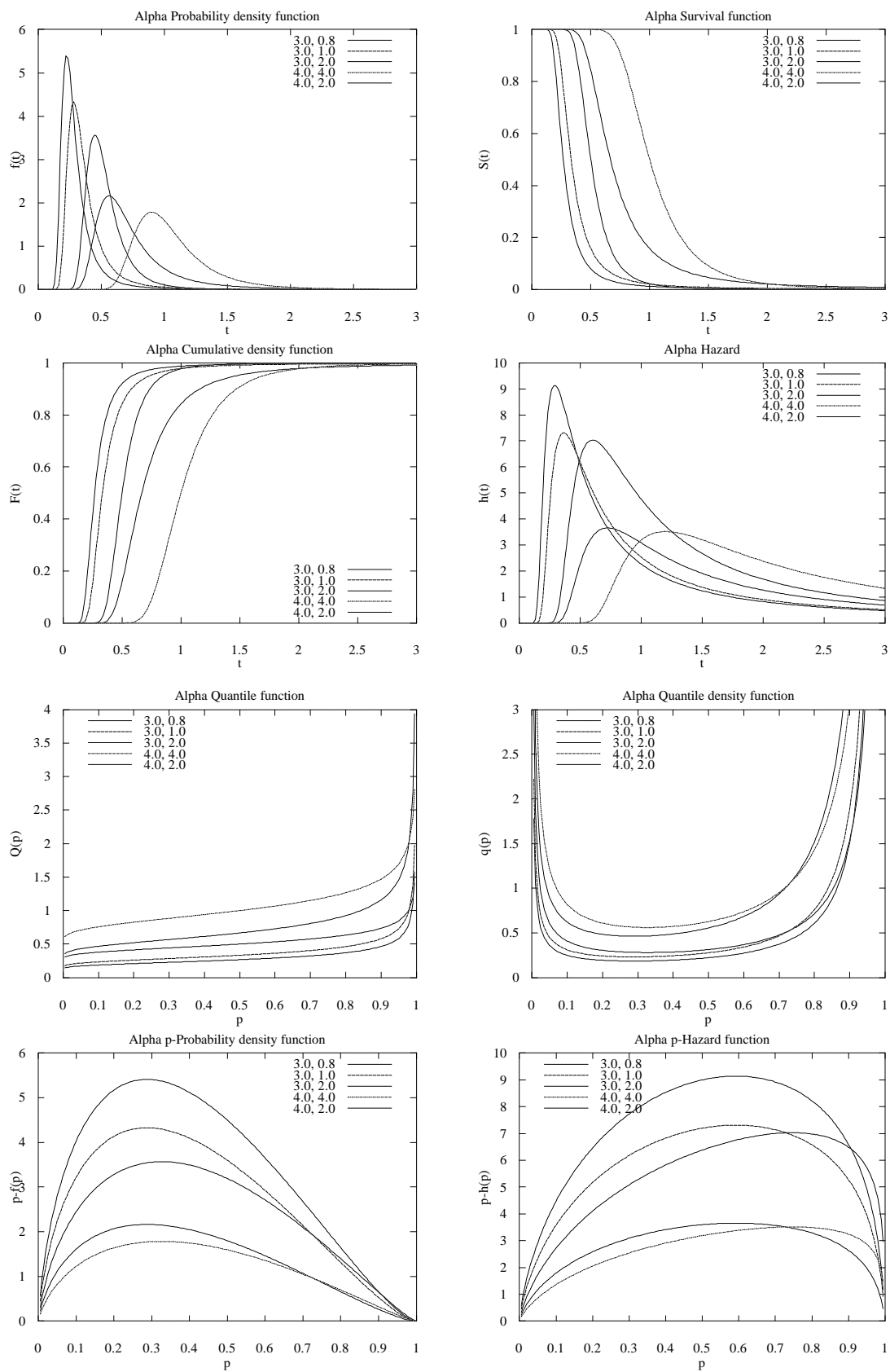
- A brief description of the PDF. For some PDFs, historical notes or mechanism by which the distribution arises are given.
- The intrinsic parameters and any constraints on the intrinsic parameters.
- The “time” variables, and the range of these variables, if any.
- The probability density function
- The survival function
- The hazard function
- The quantile function.
- One or more measures of central tendency (mean, median, mode) and sometimes the variance given in terms of the intrinsic parameters.
- Reduced forms of the model, other names commonly used for the PDF, and references to related distributions and functions in How to read a data set into *mle*.

- References to the primary source, or to a source that further describes the characteristics of the distribution.
- Graphs of the PDF, SDF, CDF, HF, quantile function, quantile density function, p-PDF, and p-HF.

ALPHA

This is the alpha distribution. This distribution has been used to model lifetimes and tool wear.

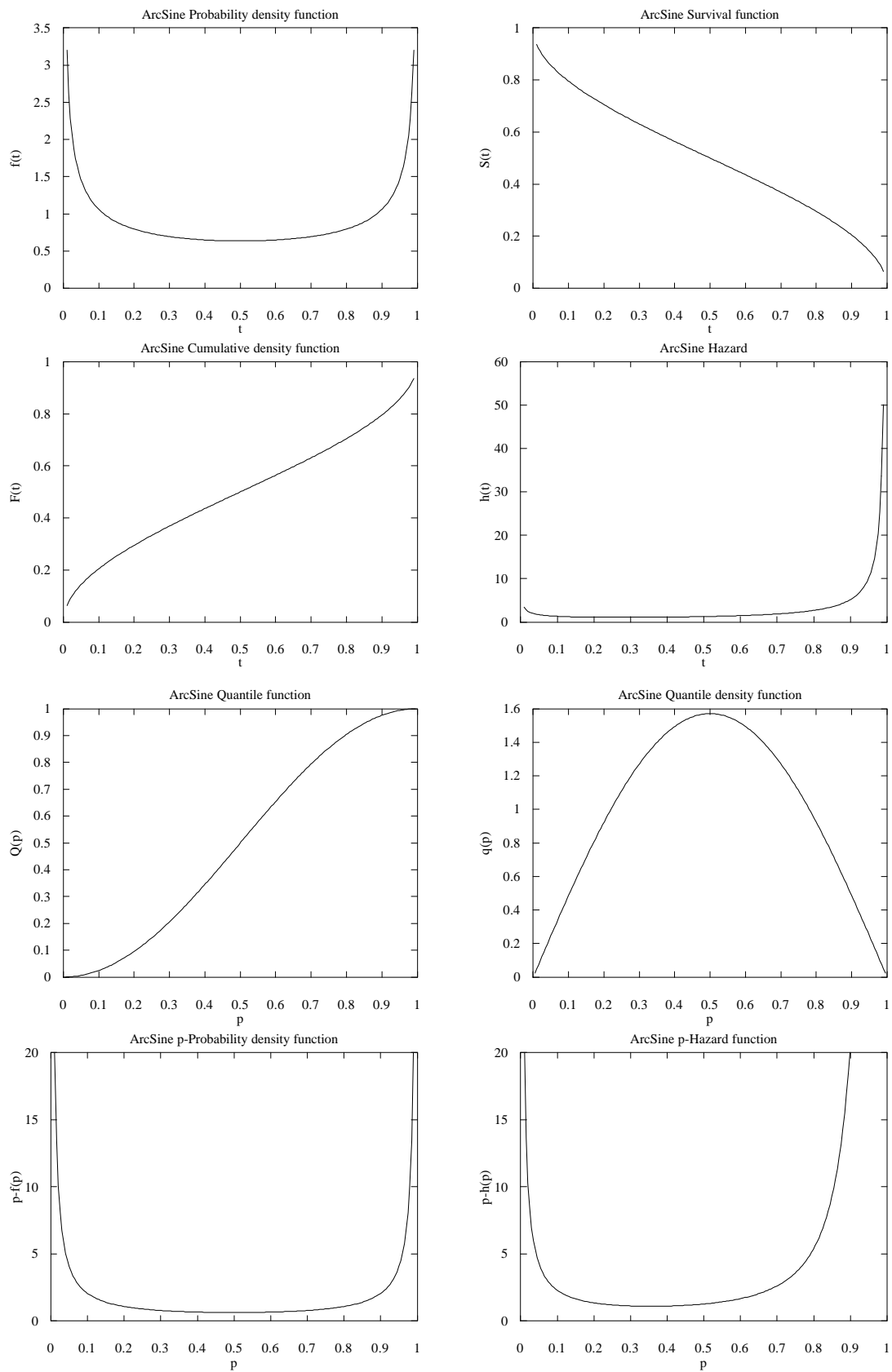
Parameters:	a and b
Constraints:	$a > 0, b > 0$
Time variables:	t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t > 0$
PDF:	$f(t) = \frac{b \exp\left[-\frac{1}{2}\left(a - \frac{b}{t}\right)^2\right]}{t^2 \Phi(a) \sqrt{2\pi}}$
SDF:	$S(t) = \frac{\Phi\left(a - \frac{t}{b}\right)}{\Phi(a)}$
Quantile:	$t_q = \frac{b}{a - \Phi_q[q\Phi(a)]}$
Median:	$\frac{b}{a - \Phi_q[\Phi(a)/2]}$
Mode:	0 and 1
References:	Johnson et al. (1994); Salvia (1985)
See also:	INVGAUSSIAN



ARCSINE

This is the parameterless arcsine distribution. The arc sine distribution arises as a special case of the Beta distribution when $\nu = \omega$. The PDF goes to infinity at $t=0$ and $t = 1$.

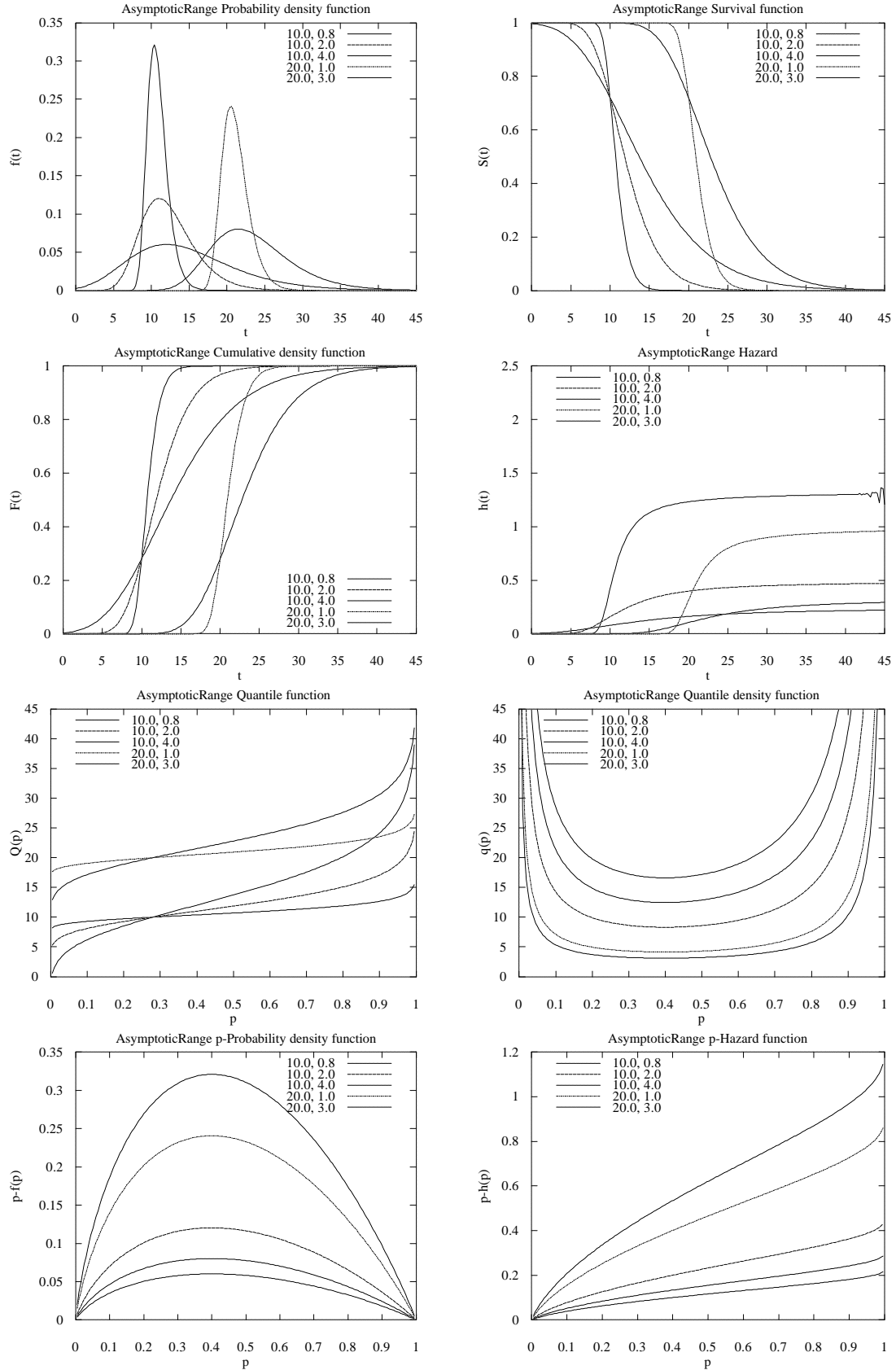
Parameters:	none.
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$0 \leq t \leq 1$
PDF:	$f(t) = \frac{1}{\pi\sqrt{t(1-t)}}$
SDF:	$S(t) = \frac{2}{\pi} \arcsin(\sqrt{t}) = \frac{2}{\pi} \arccos(2t-1)$
Quantile:	$t_q = \sin\left(\frac{q\pi}{2}\right)^2$
Mean:	1/2
Median:	1/2
Mode:	0 and 1
Variance:	1/8
References:	Johnson et al. (1995); Christensen (1984); Johnson et al. (1995); Lévy (1939); Rao (1973)
See also:	BETA



ASYMPTOTICRANGE

This is the asymptotic range distribution.

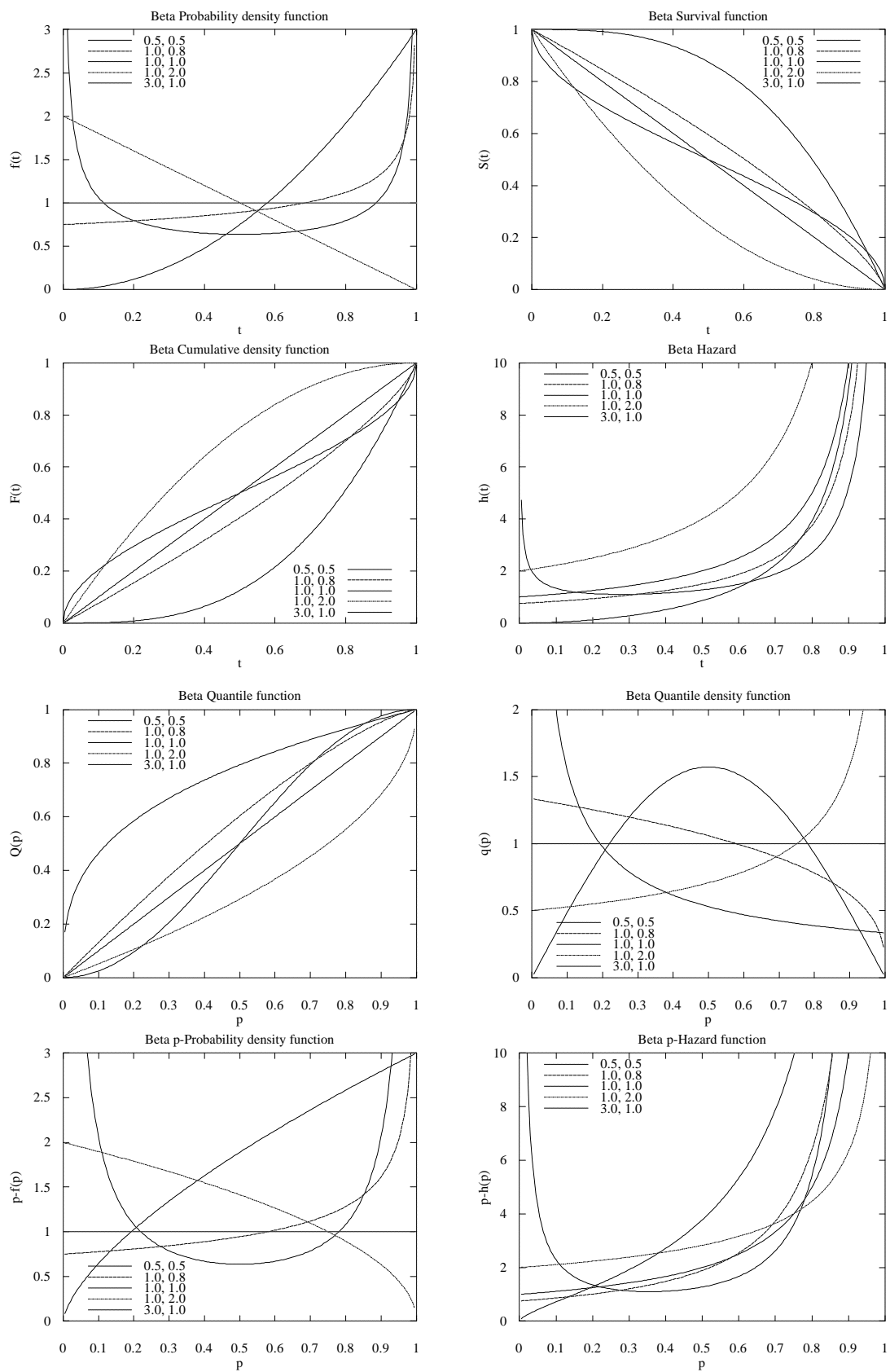
Parameters:	a (location), b (scale)
Constraints:	$b \geq 0$
Time variables:	t_u, t_e, t_α, t_0 . An exact failure is defined when $t_u = t_e$.
Range:	$-\infty < t < \infty$
PDF:	$f(t) = \frac{2}{b} e^{-\frac{t-a}{b}} K_0 \left(2e^{-\frac{t-a}{2b}} \right)$
SDF:	$S(t) = 1 - 2e^{-\frac{t-a}{2b}} K_1 \left(2e^{-\frac{t-a}{2b}} \right)$
Mean:	$a + 2\gamma b$
Median:	$\approx a + 0.92860 b$
Mode:	$\approx 0.50637 b$
Variance:	$b^2 \pi^2 / 3$
References:	Christensen (1984); Gumbel (1947)
Bugs:	The hazard function can be imprecise for $b < 1$ (see graph). This error is probably due to instability in large values of K_0 or K_1 .

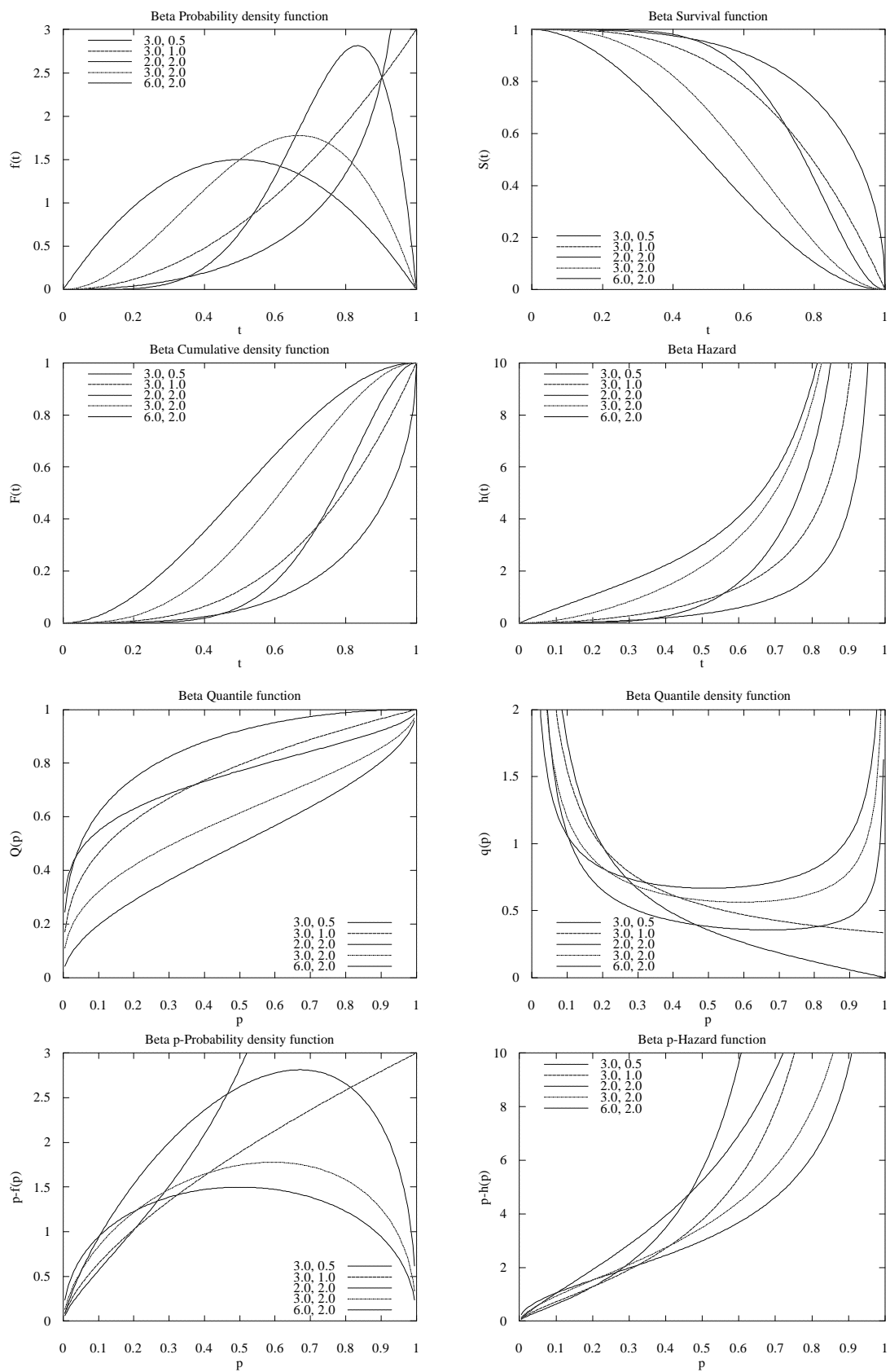


BETA

The Beta distribution is also called a Pearson Type I distribution. This distribution takes on values from 0 to 1. The distribution is J-shaped when $(\omega - 1)(\nu - 1) < 0$ and is U-shaped for $\omega < 1$ and $\nu < 1$. For other ν and ω the distribution is unimodal.

Parameters:	ν (shape 1) and ω (shape 2)
Constraints:	$\nu \geq 0, \omega \geq 0, \nu + \omega > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$0 \leq t \leq 1$
PDF:	$f(t) = \frac{t^{\nu-1}(1-t)^{\omega-1}}{B(\nu, \omega)}$
SDF:	$S(t) = \beta_p(\nu, \omega)$
Quantile:	$t_q = I_q^{-1}(\nu, \omega)$
Mean:	$\nu/(\nu + \omega)$
Mode:	$\begin{cases} (\nu - 1)/(\nu + \omega - 2), & \nu > 1, \omega > 1 \\ 0 & \nu < \omega, \nu \leq 1 \text{ or } \omega \leq 1 \\ 1 & \nu > \omega, \nu \leq 1 \text{ or } \omega \leq 1 \\ 0 \text{ and } 1 & \nu < 1, \omega < 1, \nu = \omega \\ 1/2 & \nu = 1, \omega = 1 \end{cases}$
Variance:	$\nu\omega/[(\nu + \omega)^2(\nu + \omega + 1)]$
Reduced models:	Reduces to the ARCSINE distribution when $\nu = \omega = 1/2$, reduces to the power function distribution when $\omega = 1$, reduces to the uniform distribution when $\nu = \omega = 1$.
Other names:	Pearson Type I or II
References:	Bayes (1763); Christensen (1984); Evans et al. (2000); Johnson et al. (1995); Rao (1973)





BIRNBAUMSAUNDERS

This is the Birnbaum-Saunders distribution. The distribution is nearly symmetric for small b , and becomes highly skewed for large b .

Parameters: a (location), b (scale).

Constraints: $a \geq 0, b \geq 0$

Time variables: $t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.

Range: $t > 0$

PDF:
$$f(t) = \frac{1 + \frac{a}{t}}{2b\sqrt{2\pi t}} \exp\left[\frac{-1}{2t}\left(\frac{t-a}{b}\right)^2\right]$$

SDF:
$$S(t) = 1 - \Phi\left(\frac{t-a}{b\sqrt{t}}\right)$$

Quantile:
$$t_q = \left(\frac{b}{2}\Phi_q + \sqrt{a + \frac{b^2}{2}\Phi_q}\right)^2$$

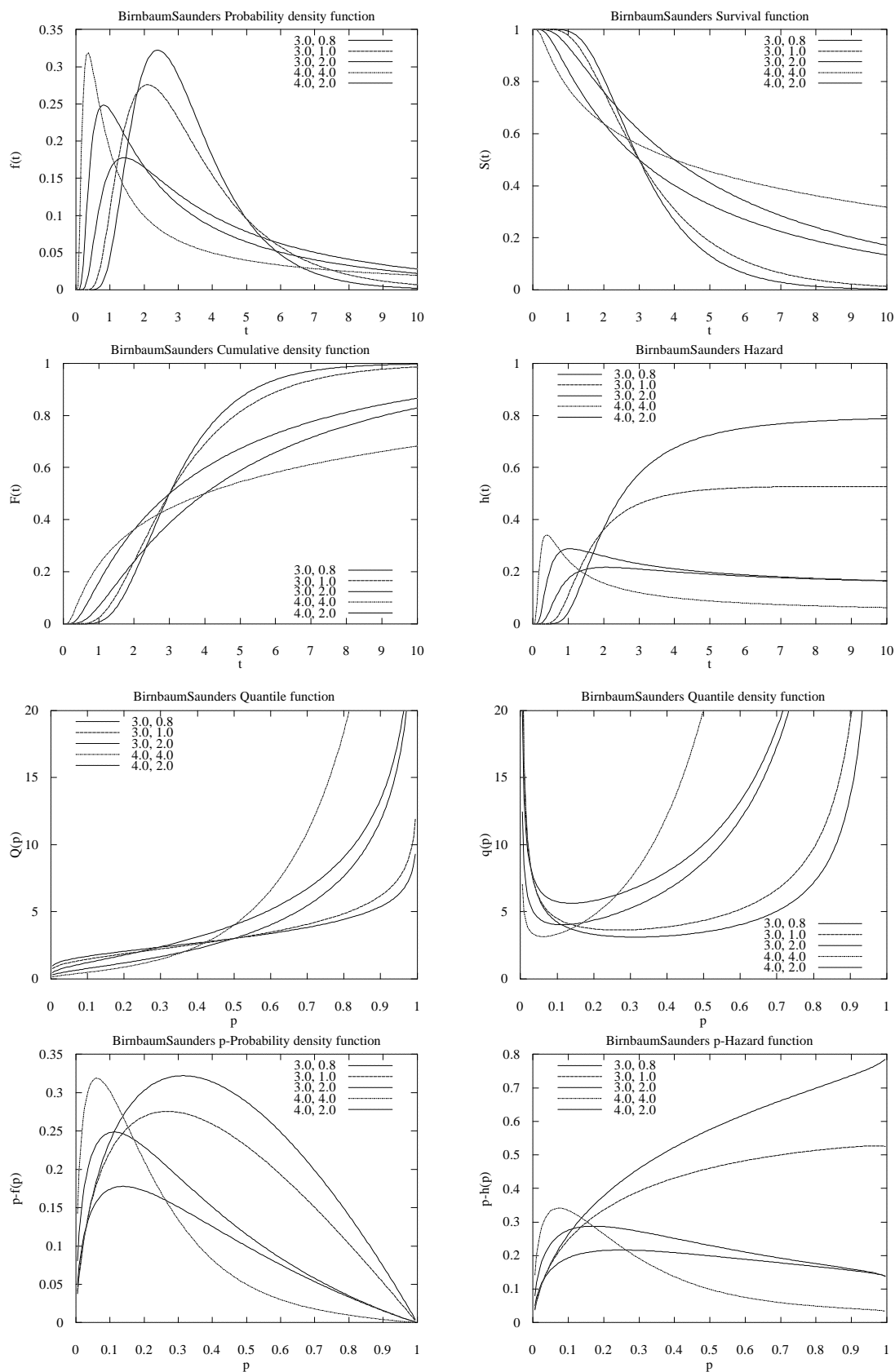
Mean: $a + b^2/2$

Median: a

Variance: $ab^2 + 5b^4/4$

Other names: Fatigue-life

References: Birnbaum and Saunders (1969); Johnson et al. (1994); Christensen (1984)



BIVNORMAL

This is the bivariate normal (or Gaussian) distribution with five intrinsic parameters.

Parameters: μ_x, σ_x , are the mean and standard deviation in the x dimension; μ_y, σ_y are the mean and standard deviations in the y dimension, and ρ is the correlation between X and Y .

Constraints: $\sigma_1 > 0, \sigma_2 > 0, 0 \leq \rho \leq 1$

Time variables: $t_{ux}, t_{uy}, t_{ex}, t_{ey}, t_{\alpha x}, t_{\alpha y}, t_{\omega x}, t_{\omega y}$.

Range: $-\infty < t < \infty$

PDF:

$$f(t_x, t_y) = \frac{\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(t-\mu_x)^2}{\sigma_x^2} - \frac{2\rho(t-\mu_x)(t-\mu_y)}{\sigma_x\sigma_y} + \frac{(t-\mu_y)^2}{\sigma_y^2}\right)\right]}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}}$$

Mean: μ_x, μ_y

Median: μ_x, μ_y

Mode: μ_x, μ_y

Variance: σ_x^2, σ_y^2

Covariance(X, Y): $\rho\sigma_x\sigma_y$

Notes: Covariate effects cannot be modeled on the hazard.

See also: NORMAL

CAUCHY

This is the Cauchy distribution. The distribution is unimodal and symmetric and with tails that extend to infinity. The quartiles are found as $a - b$ and $a + b$.

Parameters: a (location) and b (scale)

Constraints: $b > 0$

Time variables: t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.

Range: $-\infty < t < \infty$

PDF:
$$f(t) = \frac{1}{\pi b \left[1 + \left(\frac{t-a}{b} \right)^2 \right]}$$

SDF:
$$S(t) = \frac{2 \arctan \left(\frac{t-a}{b} \right) + \pi}{2\pi}$$

Quantile:
$$t_q = a + b \tan \left[\pi \left(q - \frac{1}{2} \right) \right]$$

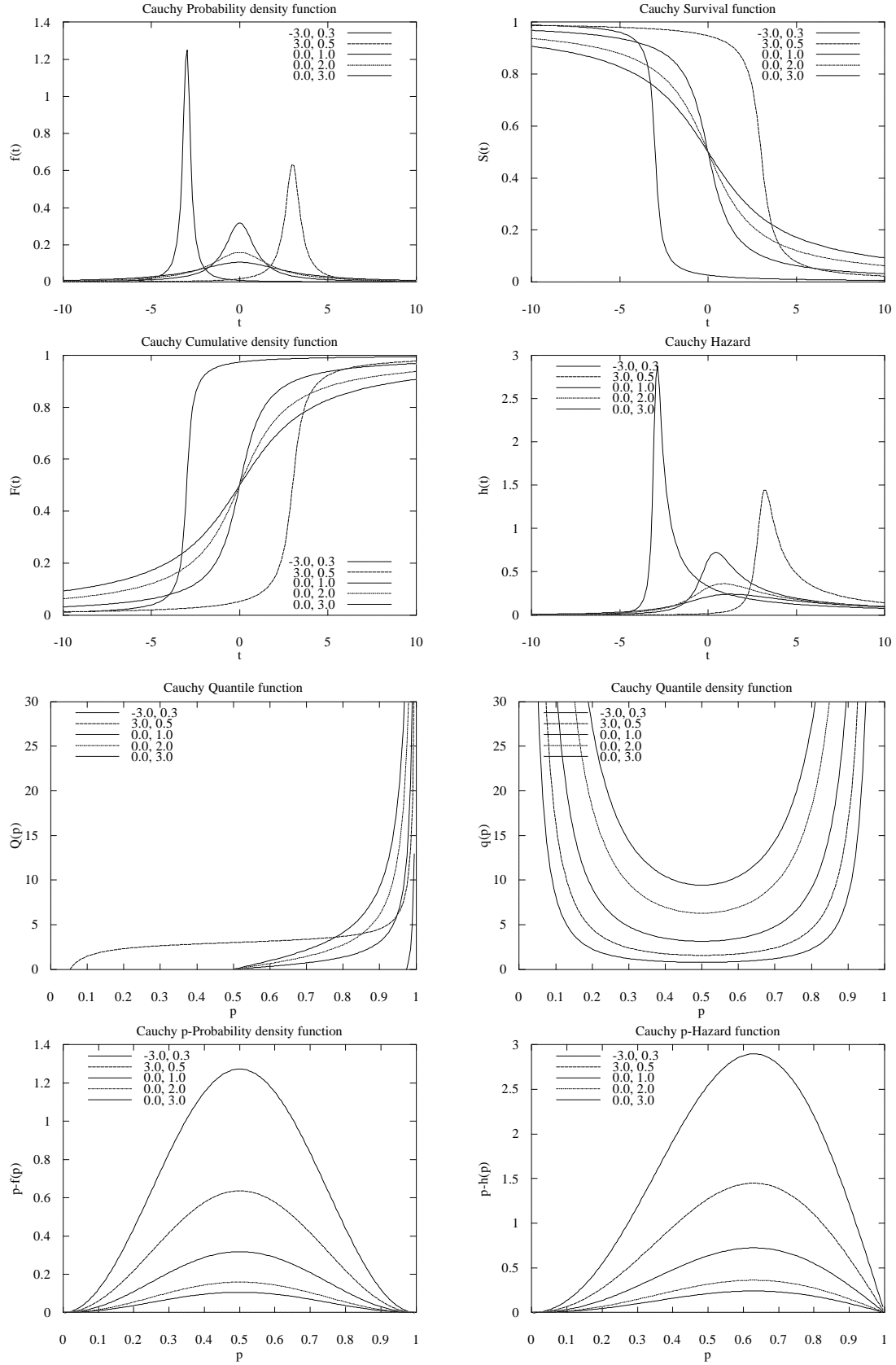
Mean: Doesn't exist

Median: a

Mode: a

Variance: ∞

References: Johnson et al. (1994); Christensen (1984); Evans et al. (2000); Rao (1973).



CHI

This is the three-parameter chi distribution.

Parameters: a (location), b (scale), c (shape).

Constraints: $a \geq 0, b > 0, c \geq 0$

Time variables: $t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.

Range: $t \geq 0$

PDF:
$$f(t) = \frac{2^{1-\frac{c}{2}}}{b\Gamma(\frac{c}{2})} \left(\frac{t-a}{b}\right)^{c-1} e^{-\frac{1}{2}\left(\frac{t-a}{b}\right)^2}$$

SDF:
$$S(t) = \frac{\Gamma\left[\frac{c}{2}, \frac{1}{2}\left(\frac{t-a}{b}\right)^2\right]}{\Gamma(\frac{c}{2})}$$

Hazard:
$$h(t) = \frac{2^{1-\frac{c}{2}} \left(\frac{t-a}{b}\right)^{c-1} e^{-\frac{1}{2}\left(\frac{t-a}{b}\right)^2}}{b\Gamma(\frac{c}{2}) - b\gamma\left[\frac{c}{2}, \frac{1}{2}\left(\frac{t-a}{b}\right)^2\right]}$$

Quantile: $q_t = a + b\sqrt{\chi_q^2(c)}$

Mean: $a + b\sqrt{2}\Gamma\left(\frac{c+1}{2}\right)\Gamma\left(\frac{c}{2}\right)^{-1}$

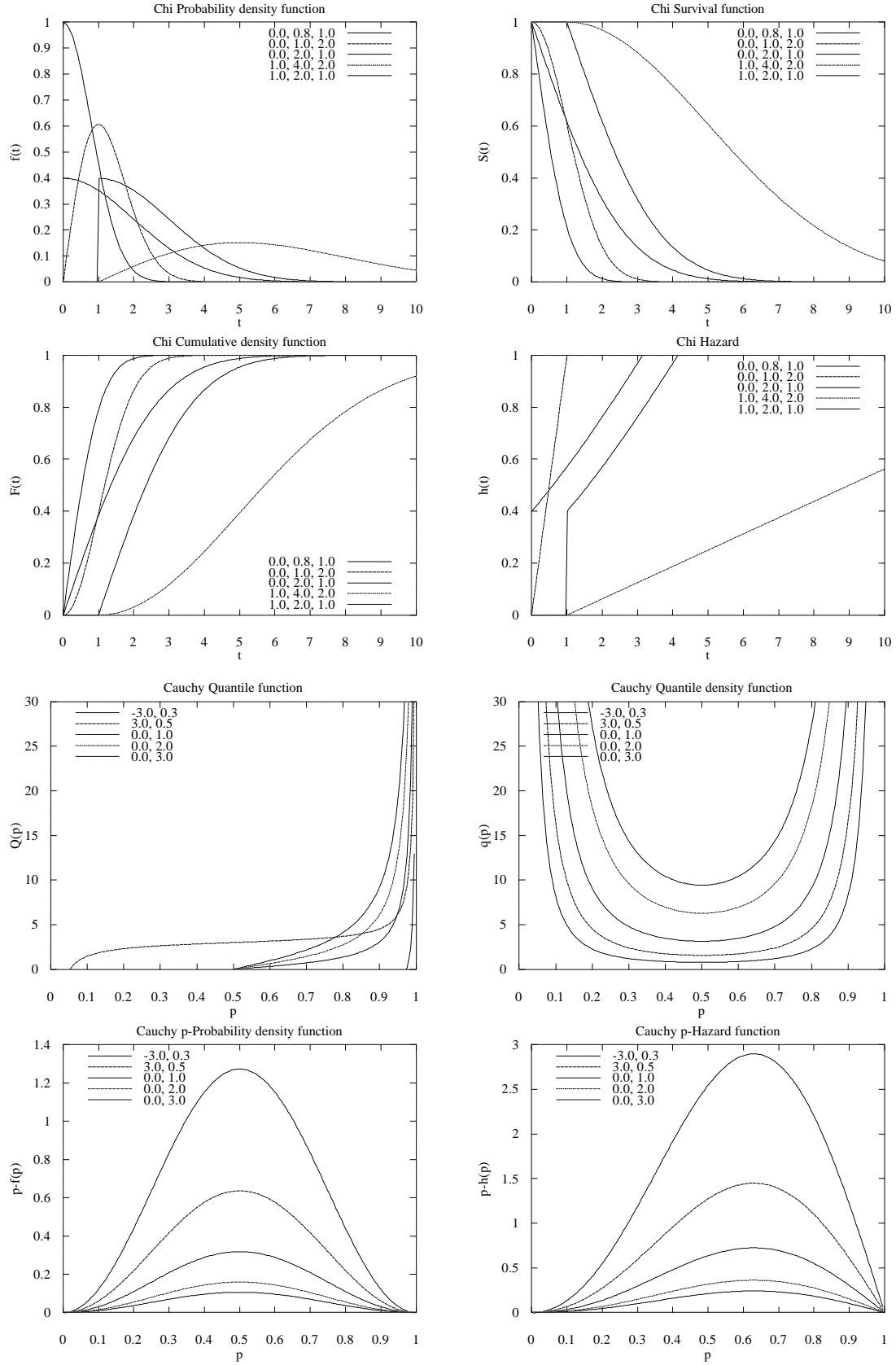
Mode:
$$\begin{cases} a + b\sqrt{c-1}, & c > 1 \\ a, & c \leq 1 \end{cases}$$

Variance: $cb^2 - 2b^2\left[\Gamma\left(\frac{c+1}{2}\right)\Gamma\left(\frac{c}{2}\right)^{-1}\right]^2$

Reduced models: Reduces to a Rayleigh distribution with $c = 2$, and a type of Maxwell distribution with $c = 3$.

References: Christensen (1984); Evans et al. (2000)

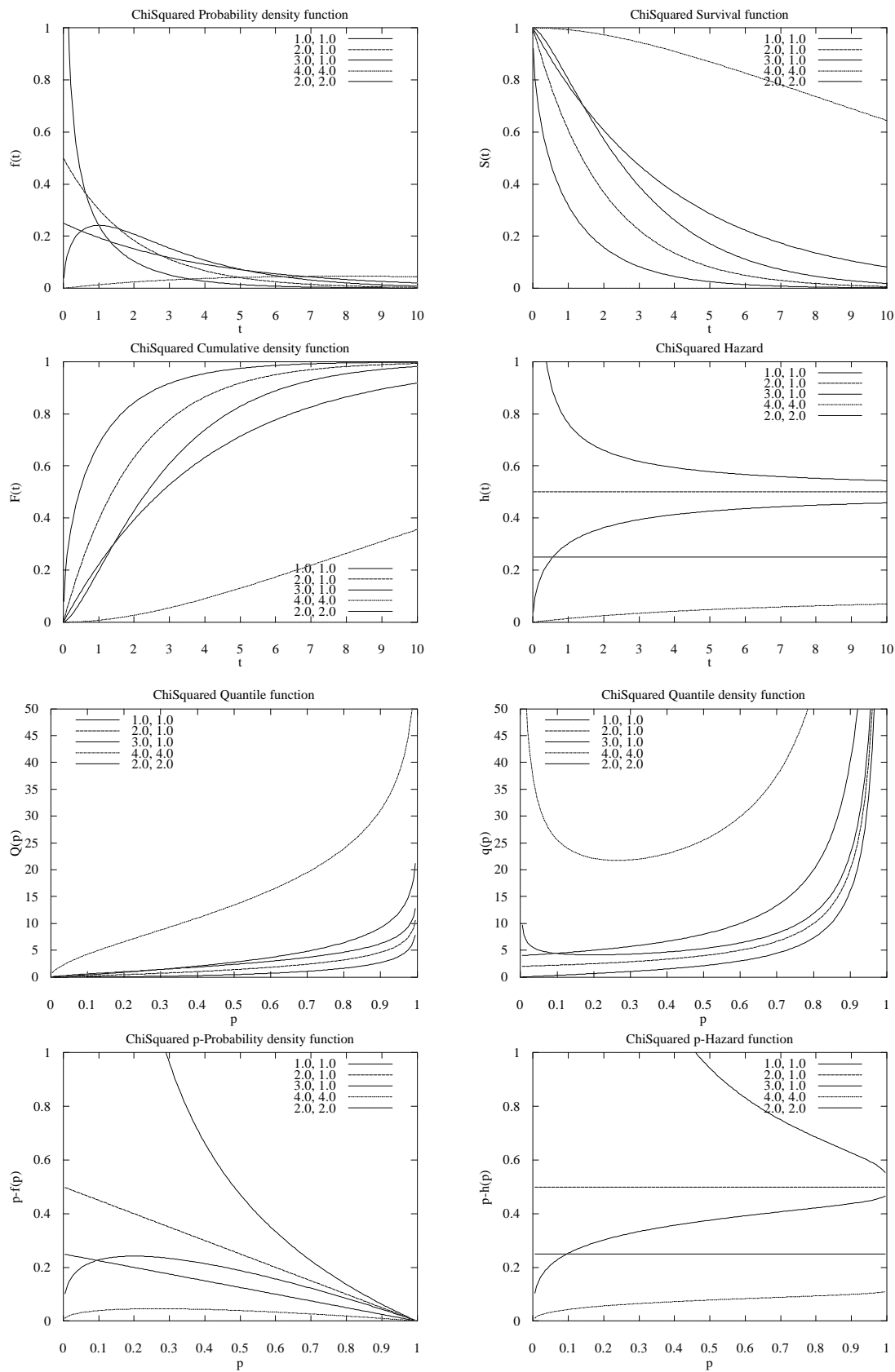
See also: RAYLEIGH, CHISQUARED, MAXWELL



CHISQUARED

This is the central Chi-squared distribution.

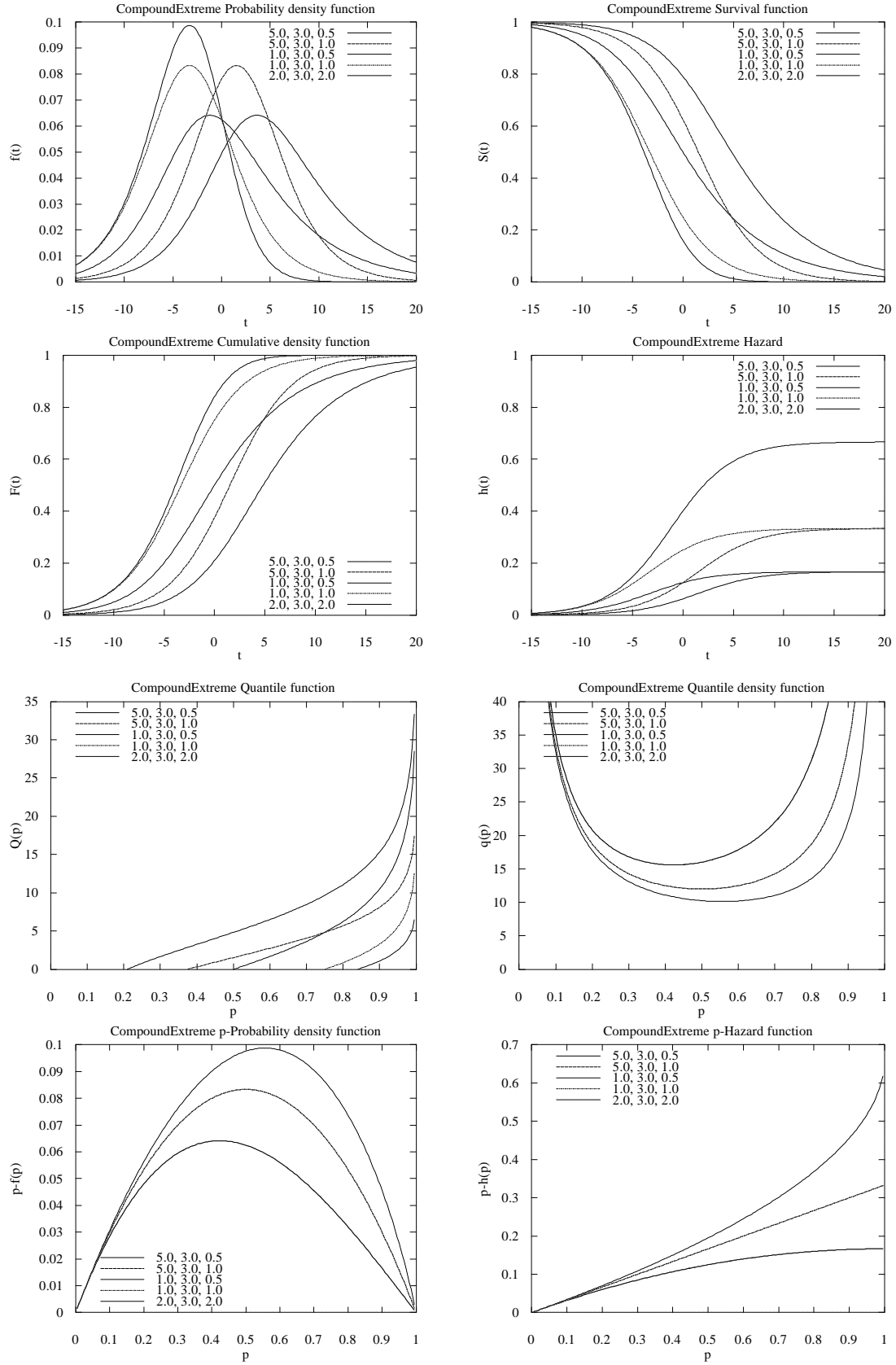
Parameters:	a (location), b (scale)
Constraints:	$a \geq 0, b > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{t^{\frac{a}{2}-1} e^{-\frac{t}{2b}}}{(2b)^{\frac{a}{2}} \Gamma\left(\frac{a}{2}\right)}$
SDF:	$S(t) = \frac{\gamma\left(\frac{a}{2}, \frac{t}{2b}\right)}{\Gamma\left(\frac{a}{2}\right)}$
Hazard:	$h(t) = \frac{t^{\frac{a}{2}-1} e^{-\frac{t}{2b}}}{\gamma\left(\frac{a}{2}, \frac{t}{2b}\right) (2b)^{\frac{a}{2}}}$
Quantile:	$q_t = b\chi_q^2(a)$
Mean:	ab
Median:	$\approx ab - 2/3$, when ab is large.
Mode:	$\begin{cases} b(a-2), & a > 2 \\ 0, & a \leq 2 \end{cases}$
Variance:	$2ab^2$
Reduced models:	Reduces to an exponential distribution with $\lambda=(2b)^{-1}$ when $a = 2$.
References:	Johnson et al. (1994); Christensen (1984); Evans et al. (2000); Pearson (1900)
See also:	GAMMA, CHI



COMPOUNDEXTREME

This is the compound extreme value distribution.

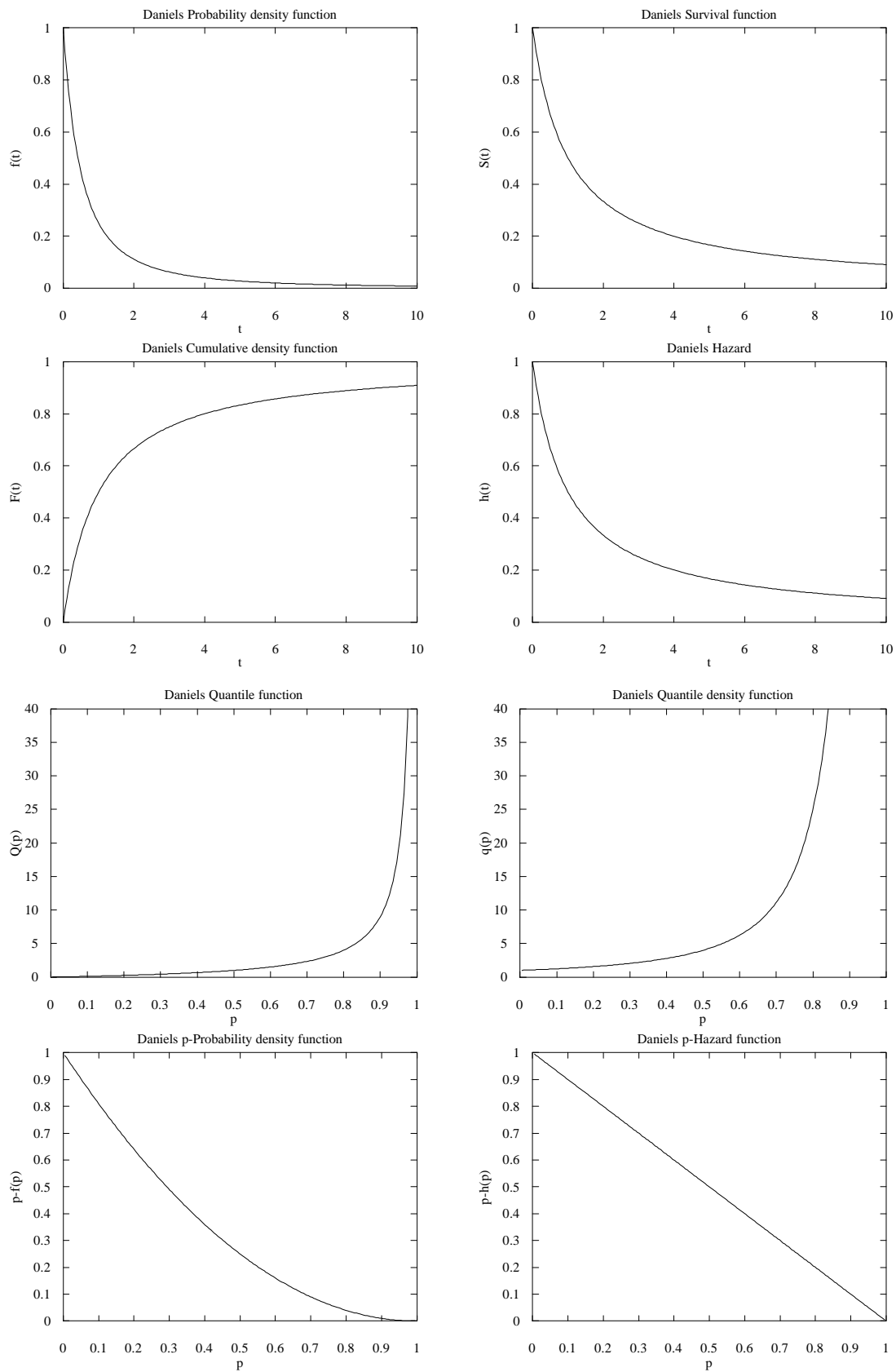
Parameters:	a (location), b (scale), c (shape).
Constraints:	$a > 0, b > 0, c > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$-\infty < t < \infty$
PDF:	$f(t) = \frac{ca^c e^{\frac{t}{b}}}{\left(a + be^{\frac{t}{b}}\right)^{c+1}}$
SDF:	$S(t) = \left(\frac{a}{a + be^{\frac{t}{b}}}\right)^c$
Hazard:	$h(t) = \frac{c}{ae^{-\frac{t}{b}} + b}$
Quantile	$t_q = b \ln \left\{ \frac{a}{b} \left[(1-q)^{-c-1} - 1 \right] \right\}$
Mean:	$b \left[\ln \left(\frac{a}{b} \right) + \gamma - \psi(c) \right]$
Median:	$\ln \left[\frac{a}{b} \left(2^{\frac{1}{c}} - 1 \right) \right] b$
Mode:	$b \ln \left(\frac{a}{bc} \right)$
Variance:	$b^2 [\psi'(1) - \psi'(c)]$
References:	Christensen (1984); Johnson et al. (1995)
See also:	LARGEEXTREME



DANIELS

This is the parameterless Daniel's distribution.

Parameters:	none
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = (1+t)^{-2}$
SDF:	$S(t) = (1+t)^{-1}$
Hazard:	$h(t) = (1+t)^{-1}$
Quantile:	$t_q = \frac{q}{1-q}$
Mean:	∞
Median:	1
Mode:	0
Variance:	∞
References:	Christensen (1984); Daniels (1945)



DISK

The disk distribution begins at mode a and monotonically approaches zero at $a + 4b$, somewhat akin to a shifted negative exponential distribution.

Parameters: a (location), b (scale)

Constraints: $b > 0$

Time variables: $t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.

Range: $a \leq t \leq a + 4b$

PDF:
$$f(t) = \frac{1}{b} - \frac{1}{\pi b} \sqrt{\frac{t-a}{b} \left(1 - \frac{t-a}{4b}\right)} - \frac{2}{\pi b} \arcsin\left(\frac{1}{2} \sqrt{\frac{t-a}{b}}\right)$$

SDF:
$$S(t) = \frac{3}{2} - \frac{t-a}{b} + \frac{t-a+2b}{4\pi b} \sqrt{\frac{t-a}{b} \left(4 - \frac{t-a}{b}\right)} + \frac{1}{\pi} \arcsin\left(\frac{t-a-2b}{2b}\right) + 2 \frac{t-a-2b}{\pi b} \arcsin\left(\frac{1}{2} \sqrt{\frac{t-a}{b}}\right)$$

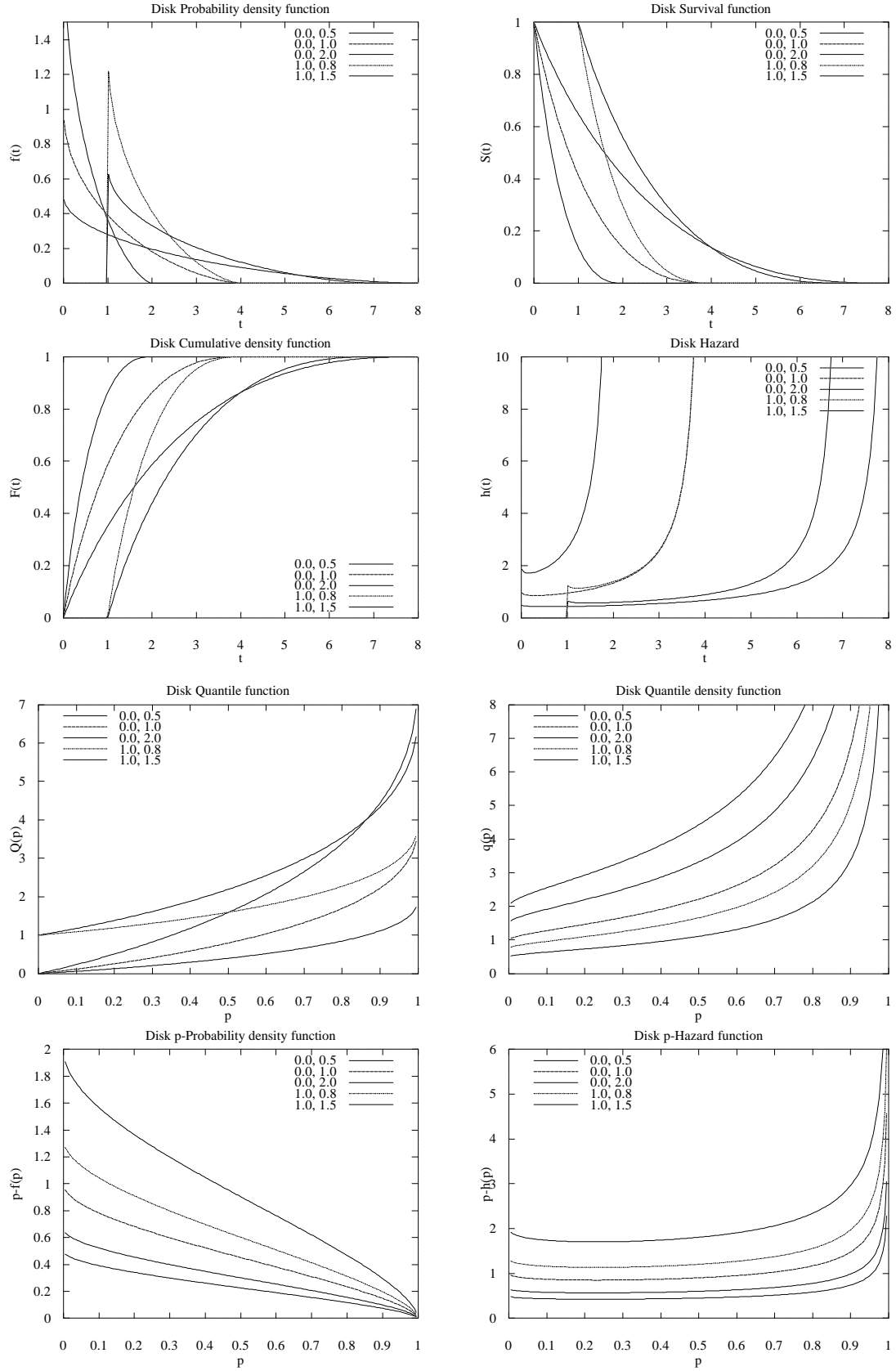
Mean: $a + b$

Median: $\approx a + 0.7944 b$

Mode: a

Variance: $2b^2/3$

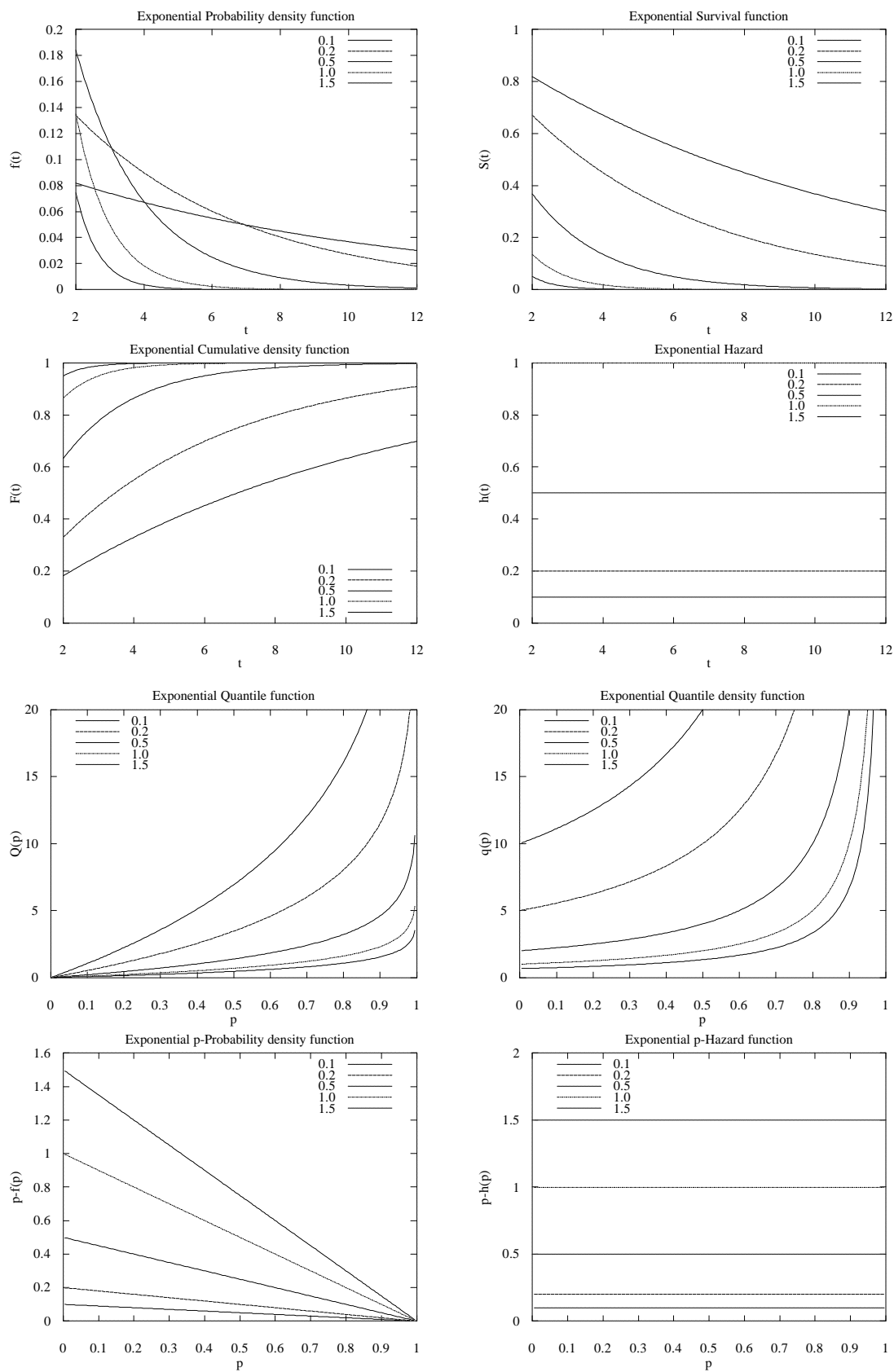
References: Borel (1925); Christensen (1984)



EXPONENTIAL

The exponential is commonly used in reliability engineering, queuing theory and biology. The 'memoryless' property of the exponential distribution is an important characteristic. It says, in effect, that for a survivor, future times to failure are completely independent of the past. Another way to express the property is that the hazard of failure is constant.

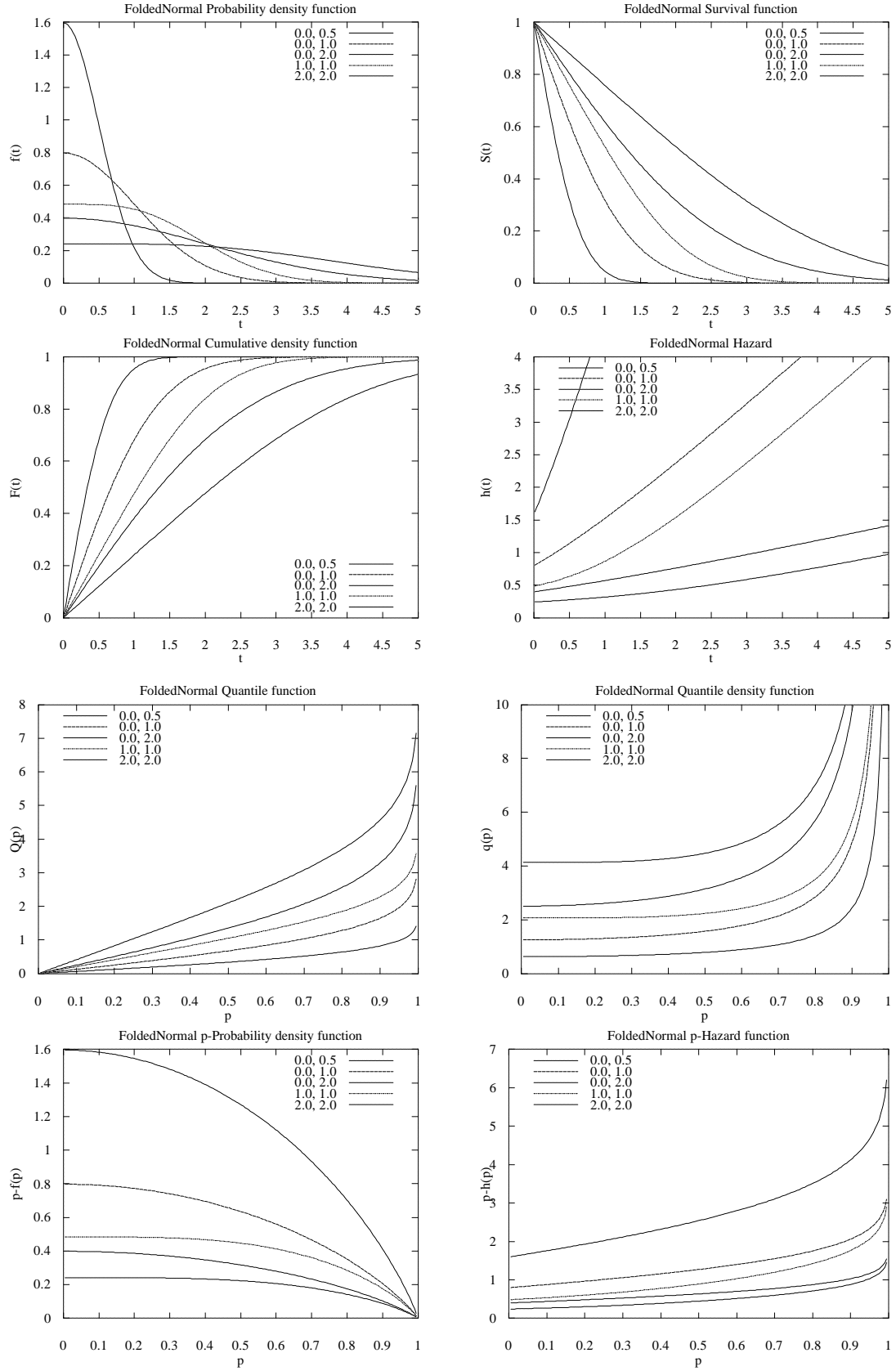
Parameter:	λ (hazard and 1/scale).
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$
Range:	$t \geq 0$
PDF:	$f(t) = \lambda \exp(-\lambda t)$
SDF:	$S(t) = \exp(-\lambda t)$
Hazard:	$h(t) = \lambda$
Quantile:	$t_q = -\ln(1 - q)/\lambda$
Mean:	$1/\lambda$
Median:	$\ln(2)/\lambda$
Mode:	0
Variance:	$1/\lambda^2$
References:	Johnson et al. (1994); Christensen (1984); Evans et al. (2000); Nelson (1982)
See also:	The SHIFTEXPONENTIAL distribution is a 2 parameter (location-scale) version of this distribution.



FOLDEDNORMAL

This distribution describes the absolute value of a normally distributed random variable.

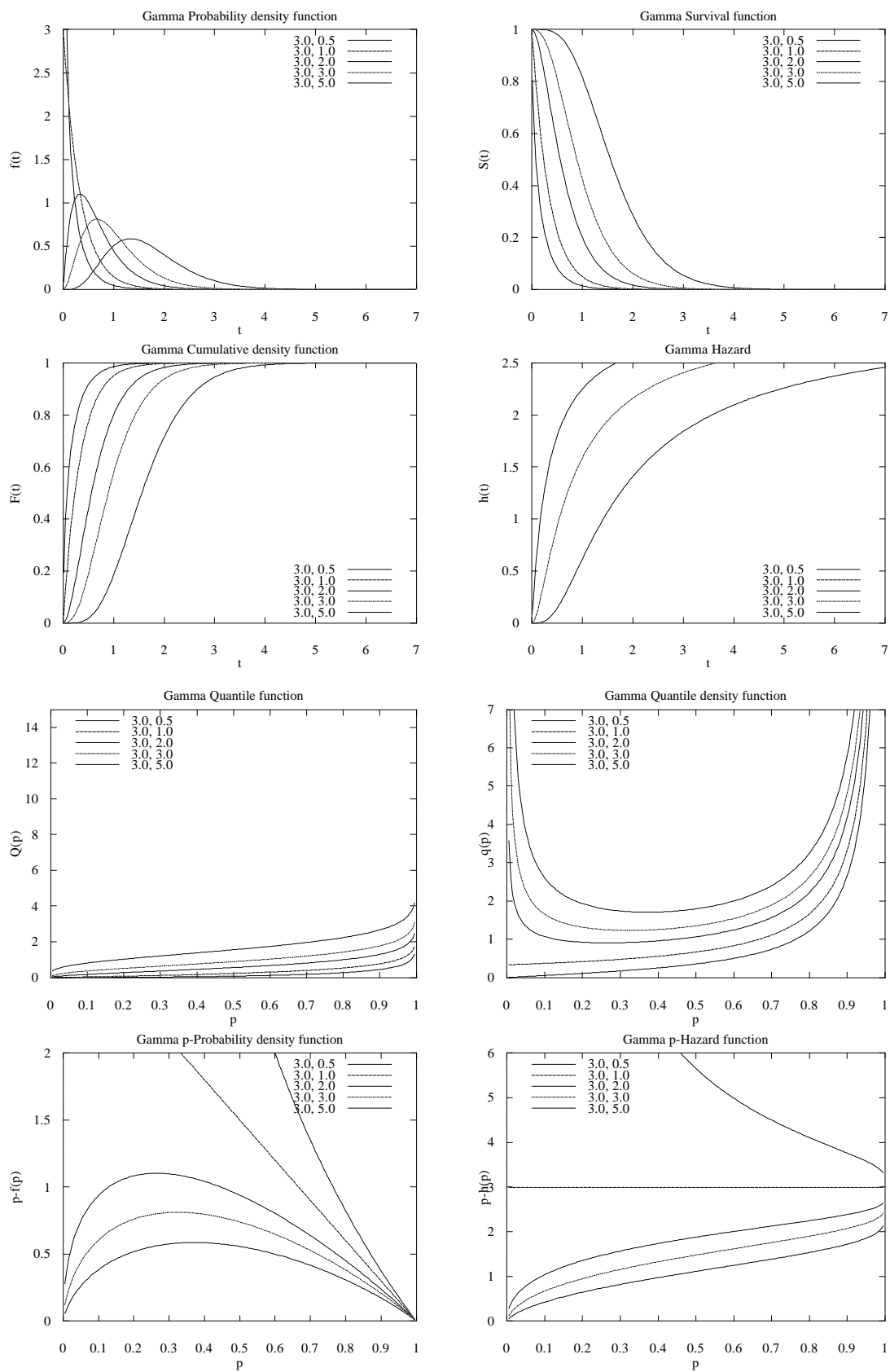
Parameters:	μ is a location parameter, σ is the scale parameter that correspond to the unfolded normal distribution.
Constraints:	$\sigma > 0, \mu \geq 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$. For censored and truncated times, positive values of t_u, t_e, t_α , and t_ω should be used.
Range:	$-\infty < t < \infty$
PDF:	$f(t) = \frac{1}{\sigma} \phi\left(\frac{t-\mu}{\sigma}\right) - \frac{1}{\sigma} \phi\left(\frac{t+\mu}{\sigma}\right)$
SDF:	$S(t) = 1 - \Phi\left(\frac{t-\mu}{\sigma}\right) + \Phi\left(\frac{-t-\mu}{\sigma}\right),$
Mean:	$\sigma \sqrt{\frac{2}{\pi}} \exp\left(-\frac{\mu^2}{2\sigma^2}\right) + \mu \Phi\left(1 - \frac{\mu}{\sigma}\right) - \mu$
Mode:	0 for $\mu \leq \sigma$, otherwise > 0
Variance:	$\sigma^2 + \mu^2 - \left[\sigma \sqrt{\frac{2}{\pi}} \exp\left(-\frac{\mu^2}{2\sigma^2}\right) + \mu \operatorname{erf}\left(\frac{\mu}{\sqrt{2}\sigma}\right) \right]^2$
Reduced models:	Reduces to the half-normal distribution when $\mu = 0$.
References:	Johnson et al. (1994).
See also:	NORMAL

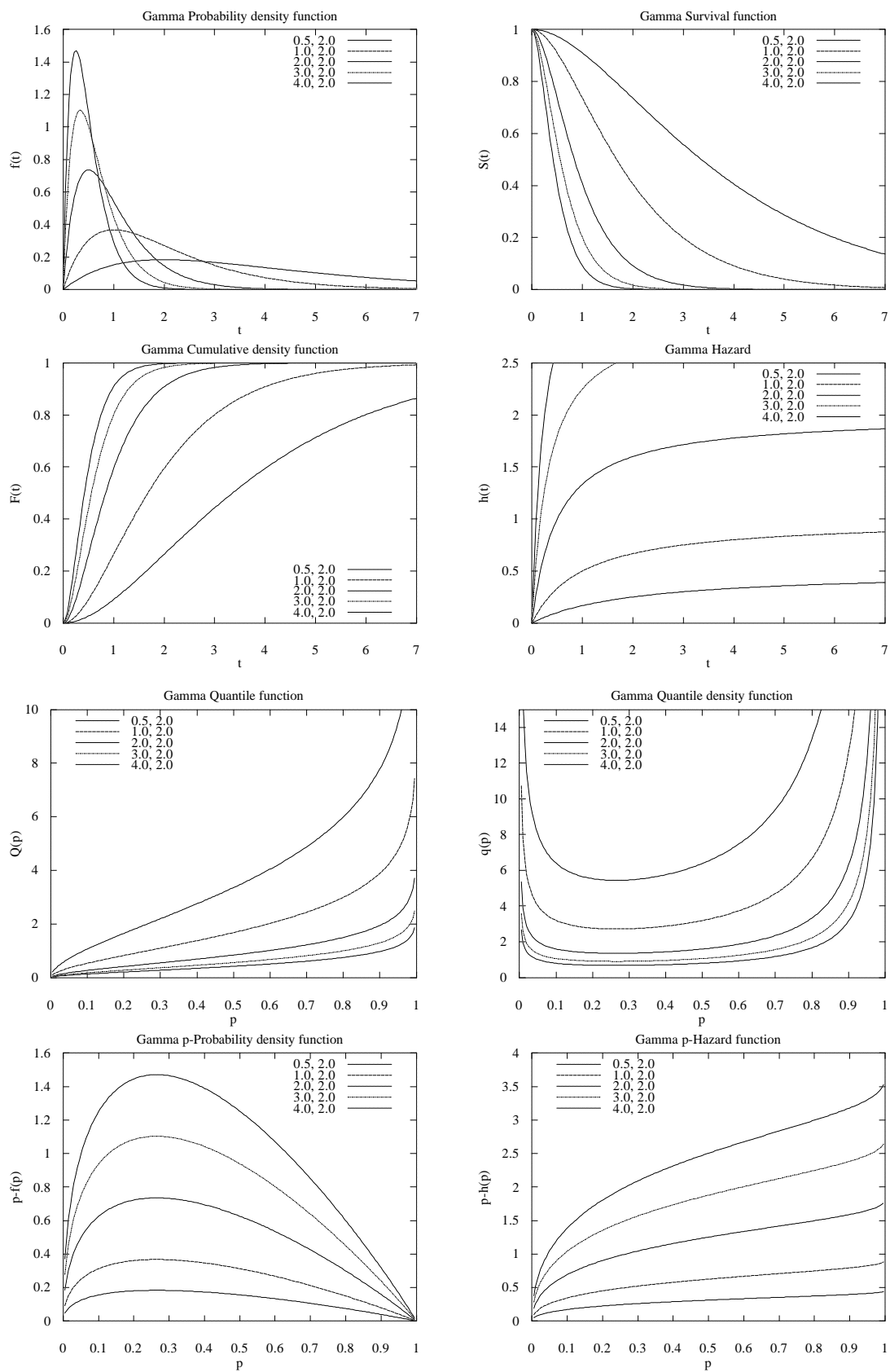


GAMMA

This is the gamma distribution, also known as the Pearson Type III distribution.

Parameters:	λ (hazard and 1/scale), c (shape)
Constraints:	$\lambda > 0, c > 0$
Time variables:	t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{\lambda^c t^{c-1}}{\Gamma(c) e^{\lambda t}}$
SDF:	$S(t) = \frac{\Gamma(c, \lambda t)}{\Gamma(c)}$
Hazard:	$h(t) = \frac{\lambda^c t^{c-1}}{\Gamma(c, \lambda t) e^{\lambda t}}$
Quantile:	$t_q = \frac{\chi_q^2(2c)}{2\lambda}$
Mean:	c/λ
Mode:	$\begin{cases} (c-1)/\lambda, & c > 1 \\ 0 & c \leq 1 \end{cases}$
Variance:	c/λ^2
Reduced models:	Reduces to an exponential distribution when $c = 1$. Reduces to an Erland distribution when parameter c is an integer. Reduces to a Chi-squared distribution with $c = v/2$. This distribution is the SHIFTGAMMA distribution with $a = 0$ and $1/b = \lambda$.
Other names:	Pearson Type III
References:	Johnson et al. (1994); Christensen (1984), Elandt-Johnson and Johnson (1980), Evans et al. (2000), Kalbfleisch and Prentice (1980).
See also:	EXPONENTIAL, SHIFTGAMMA, GENGAMMA

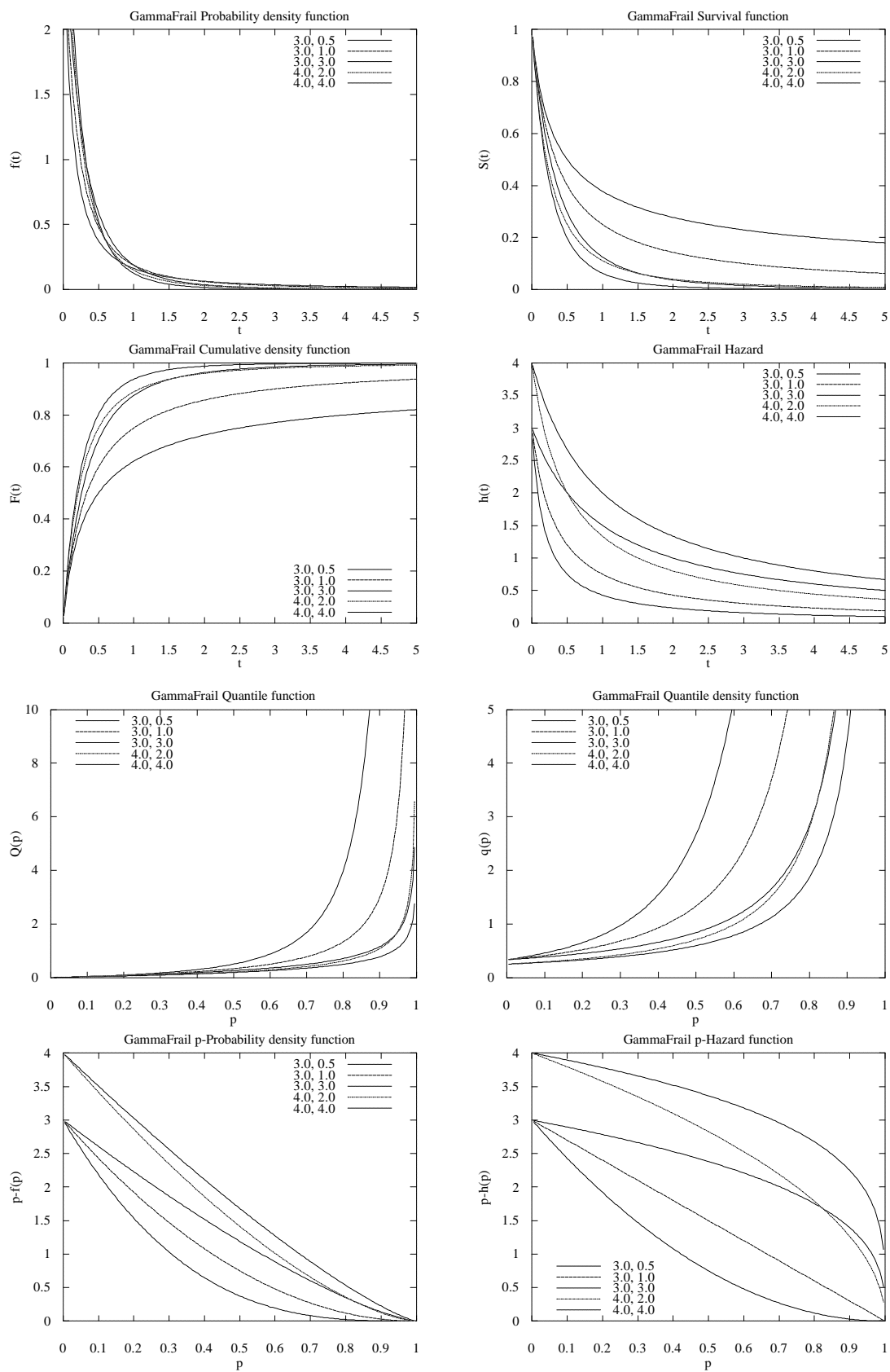




GAMMAFRAIL

This distribution serves as a model of constant hazard (negative exponential density of failure times) for each individual, but gamma-distributed frailty (heterogeneity) among individuals.

Parameters:	λ (hazard and 1/scale), c (shape)
Constraints:	$\lambda > 0, c \geq 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{\lambda c^{c+1}}{(\lambda t + c)^{c+1}}$
SDF:	$S(t) = \left(\frac{c}{c + \lambda t} \right)^c$
Hazard:	$h(t) = \frac{\lambda c}{ct + c}$
Quantile:	$t_q = \frac{c}{\lambda} \left(\frac{1}{\sqrt[c]{q}} - 1 \right)$
Mean:	$\frac{c}{\lambda(c-1)}$
Median:	$t_q = \frac{c}{\lambda} \left(\frac{1}{\sqrt[c]{\frac{1}{2}}} - 1 \right)$
Variance:	$\frac{c^3}{\lambda^2 (c-1)^2 (c-2)}$
References:	
See also:	EXPONENTIAL, GAMMA



GENGAMMA

This is the four parameter (shifted) gamma distribution, also known as the Pearson Type III distribution.

Parameters: a (location), b (scale, inverse of the hazard), c (1st shape), d (2nd shape).

Constraints: $b > 0, c > 0, d > 0$

Time variables: t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.

Range: $t \geq a$

PDF:
$$f(t) = \frac{d(t-a)^{cd-1} e^{-\left(\frac{t-a}{b}\right)^d}}{b^{cd} \Gamma(c)}$$

SDF:
$$S(t) = \frac{\Gamma\left[c, \left(\frac{t-a}{b}\right)^d\right]}{\Gamma(c)}$$

Hazard:
$$h(t) = \frac{d(t-a)^{cd-1} e^{-\left(\frac{t-a}{b}\right)^d}}{b^{cd} \Gamma\left[c, \left(\frac{t-a}{b}\right)^d\right]}$$

Quantile:
$$t_q = a + b \left[\frac{1}{2} \chi_q^2(2c) \right]^{-1}$$

Mean:
$$a + b \frac{\Gamma(c + d^{-1})}{\Gamma(c)}$$

Mode:
$$\begin{cases} a + \frac{b}{d} \left(\frac{cd-1}{d} \right)^{d-1} & cd > 1 \\ a & cd \leq 1 \end{cases}$$

Variance: $b^2 c$

Reduced models: Reduces to the shifted gamma distribution when $d = 1$. Reduces to the shifted exponential when $c = 1$ and $d = 1$. Reduces to the shifted Weibull distribution when $c = 1$. Reduces to the Chi-squared distribution with v degrees of freedom when $a = 0, b = 2, c = v/2$, and $d = 1$.

References: Christensen (1984); Evans et al. (2000); Kalbfleisch and Prentice (1980).

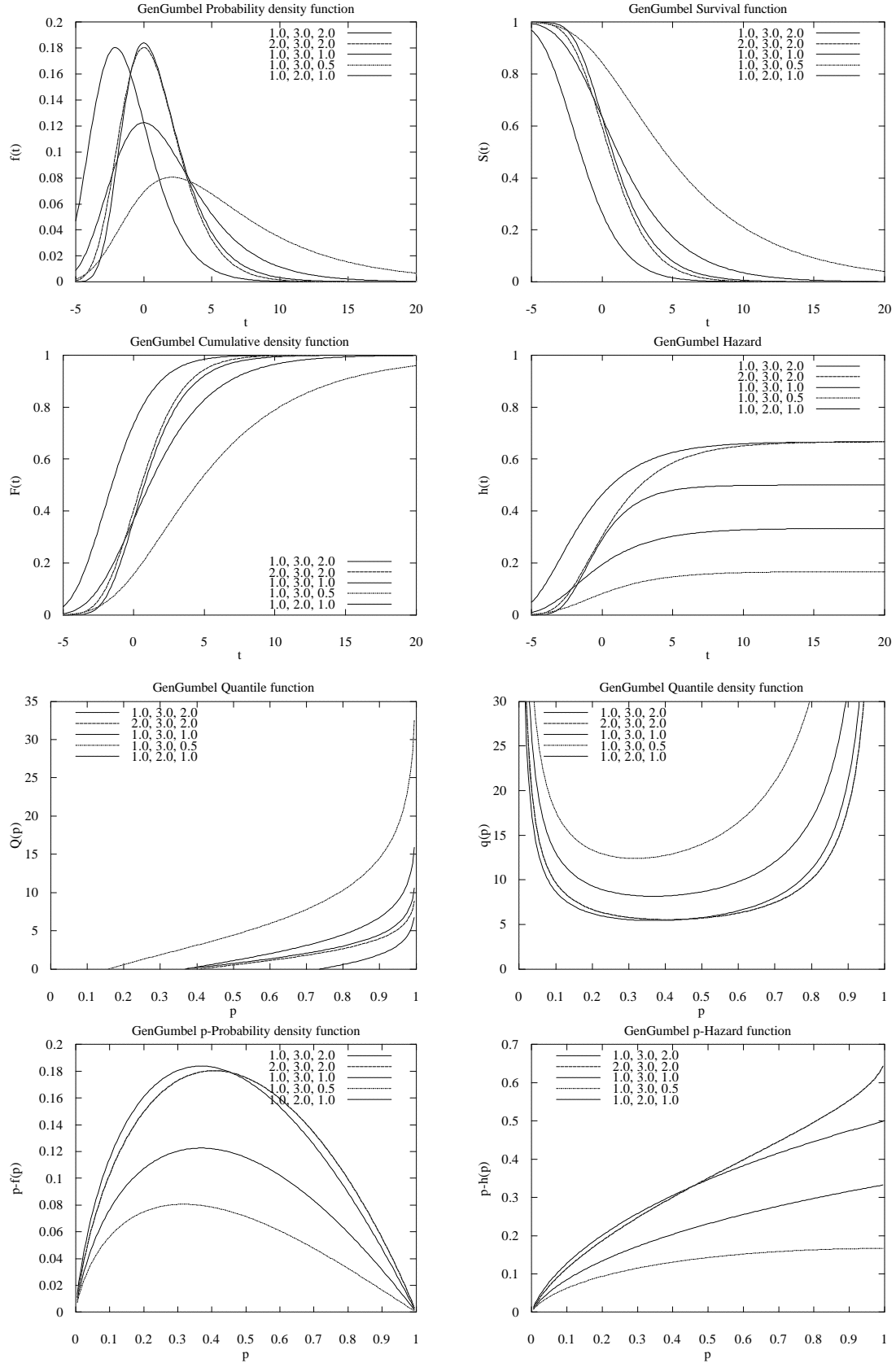
See also: SHIFTEXPONENTIAL, GAMMA, SHIFTGAMMA,
CHISQUARED, SHIFTWEIBULL

Bugs: Currently is only implemented with three intrinsic parameters!

GENGUMBEL

This is the three-parameter generalized Gumbel distribution.

Parameters:	a (location), b (scale), c (shape)
Constraints:	$a > 0, b > 0, c > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$-\infty < t < \infty$
PDF:	$f(t) = \frac{a}{b\Gamma(c)} \exp\left[-\frac{ct}{b} - ae^{-\frac{t}{b}}\right]$
SDF:	$S(t) = \frac{\gamma\left(c, ae^{-\frac{t}{b}}\right)}{\Gamma(c)}$
Hazard:	$f(t) = \frac{a}{b\gamma\left(c, ae^{-\frac{t}{b}}\right)} \exp\left[-\frac{ct}{b} - ae^{-\frac{t}{b}}\right]$
Mean:	$b[\ln(a) - \psi(c)]$
Median:	$b\ln(a/c)$
Variance:	$b^2\psi'(c)$
References:	Ahuja and Nash (1967); Christensen (1984)
See also:	GUMBEL



GOMPERTZ

This is the Gompertz distribution, which is sometimes used as a model of senescent mortality (with $b > 0$) and infant mortality (with $b < 0$).

Parameters: a (scale), b (shape)

Constraints: $a \geq 0$

Time variables: $t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.

Range: $t \geq 0$

PDF:

$$f(t) = a \exp\left[bt + \frac{a}{b}(1 - e^{bt})\right] N \quad N = \begin{cases} 1, & b \geq 0 \\ \left(1 - e^{\frac{a}{b}}\right)^{-1} & b < 0 \end{cases}$$

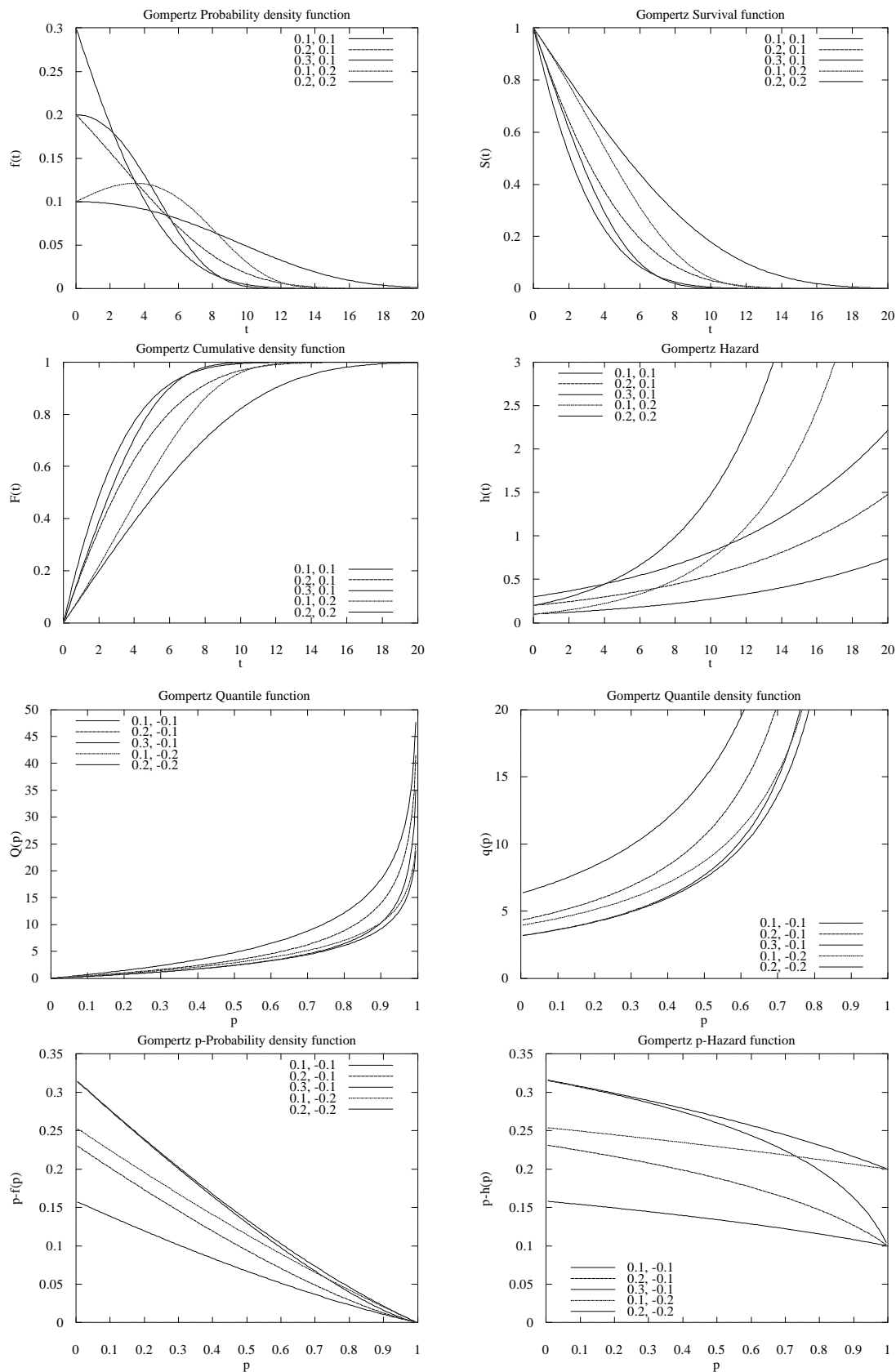
SDF: $S(t) = \exp\left[\frac{a}{b}(1 - e^{bt})\right] N$

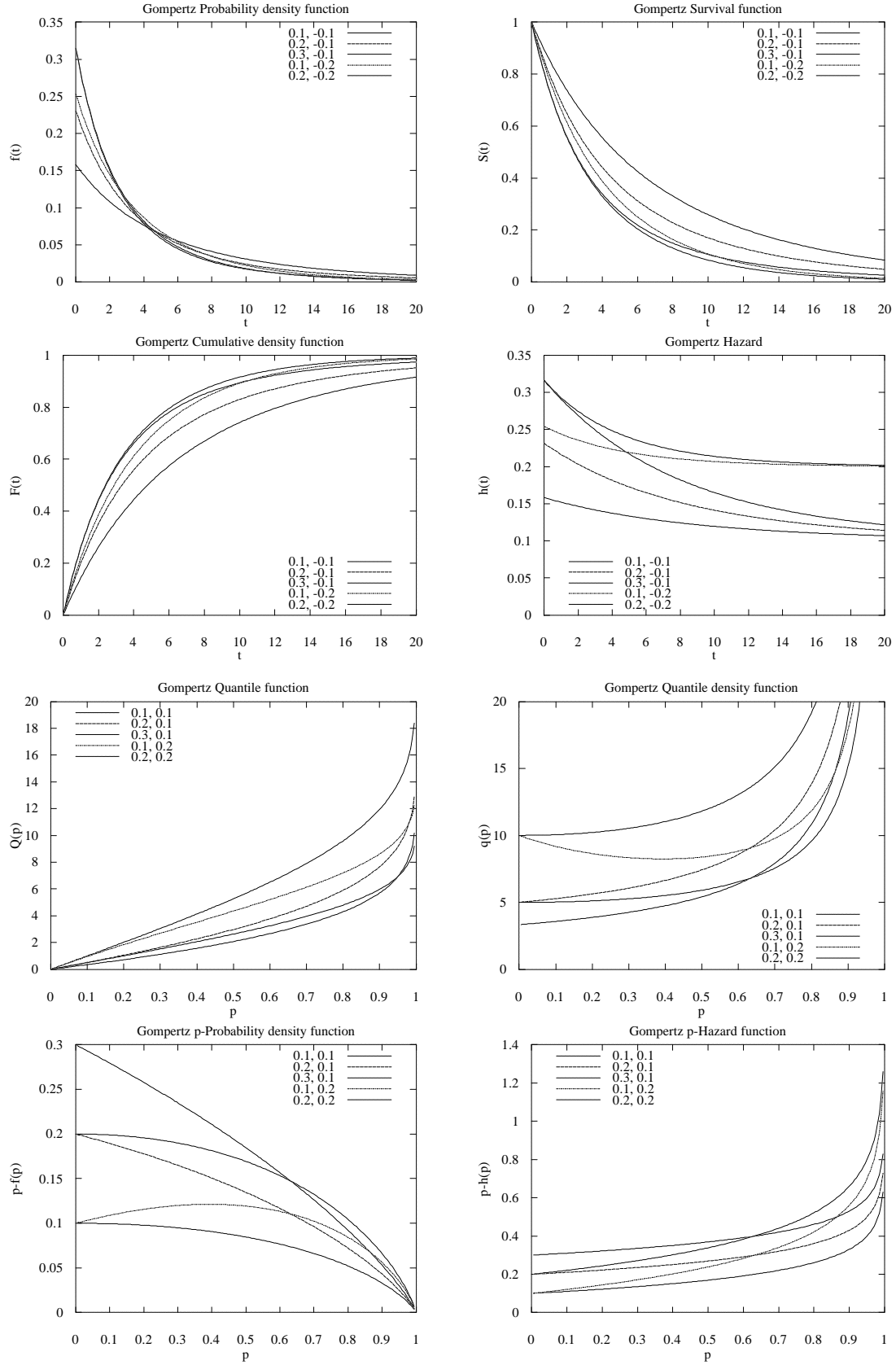
Hazard: $h(t) = a \exp(bt)$

Reduced models: When $b = 0$, the PDF is exponential with parameter a .

References: Johnson et al. (1995); Christensen (1984)

See also: EXPONENTIAL, MAKEHAM, MIXMAKEHAM, SILER

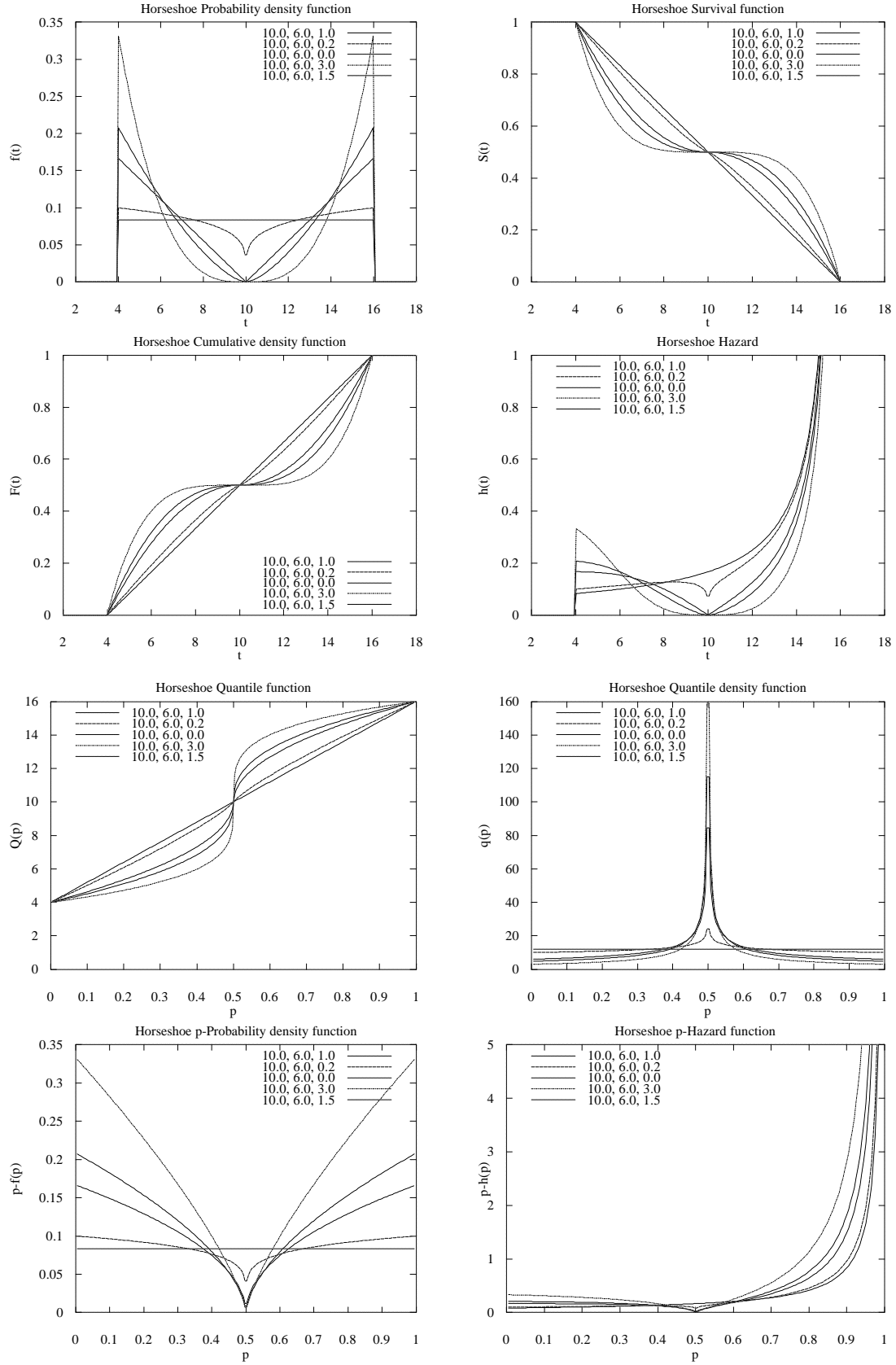




HORSESHOE

This is the horseshoe distribution. The distribution is a mirrored power function; it discontinuous at the mean, and the mirror-like symmetrical is down the mean. Except when c is zero, the distribution is always bimodal.

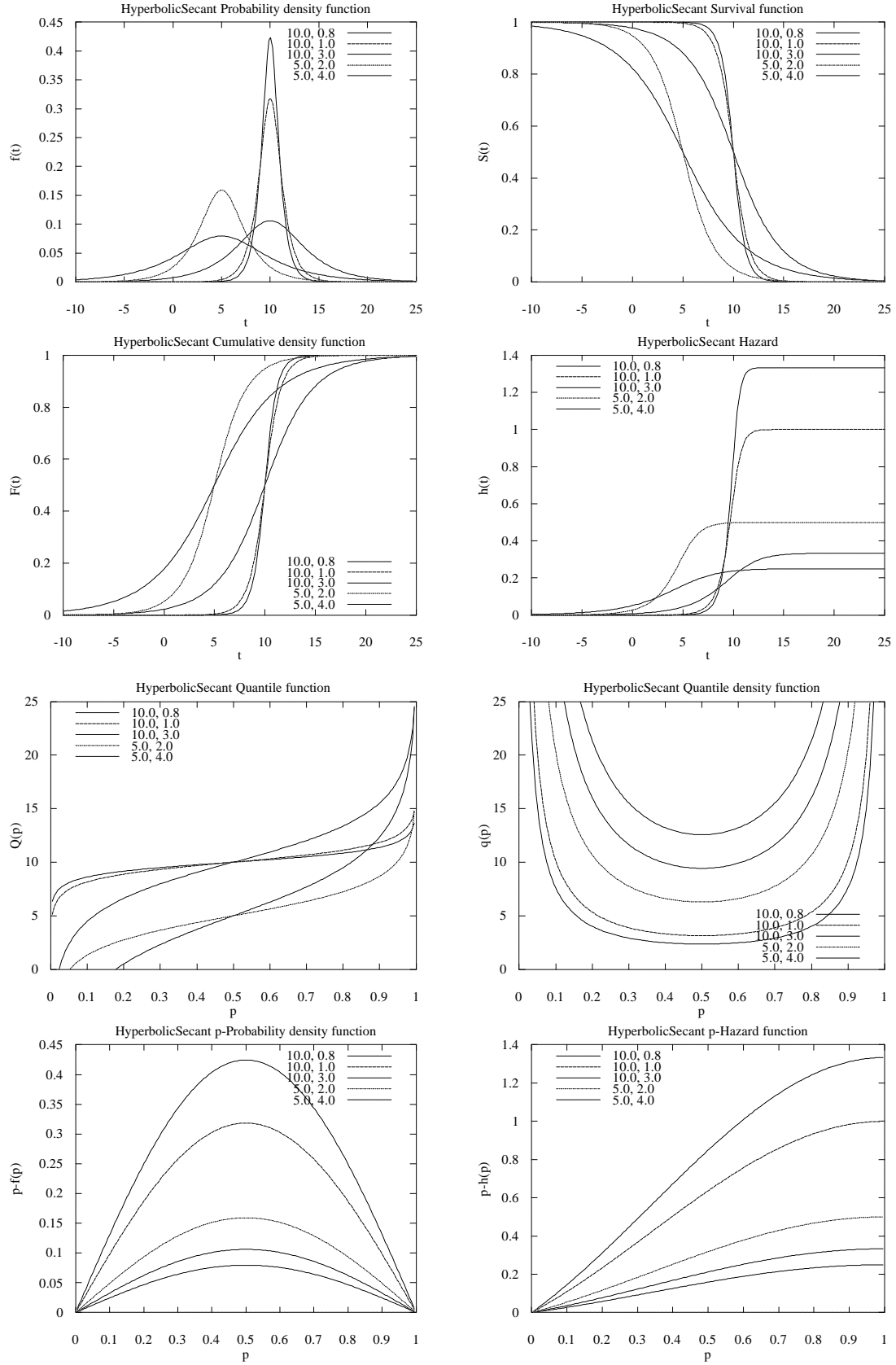
Parameters:	a (location), b (scale), c (shape)
Constraints:	$b > 0, c \geq 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$(a - b) \leq t \leq (a + b)$
PDF:	$f(t) = \frac{c+1}{2b} \left \frac{x-a}{b} \right ^c$
SDF:	$S(t) = \begin{cases} \frac{1}{2} \left[1 - \left(\frac{x-a}{b} \right)^{c+1} \right], & x \geq a \\ \frac{1}{2} \left[1 + \left(-\frac{x-a}{b} \right)^{c+1} \right], & x \leq a \end{cases}$
Quantile	$t_q = \begin{cases} a + b(2q-1)^{(c+1)^{-1}}, & q \geq 1/2 \\ a - b(1-2q)^{(c+1)^{-1}}, & q \leq 1/2 \end{cases}$
Mean:	a
Median:	a
Mode:	$a \pm b$
Variance:	$b^2(c+1)/(c+3)$
Reduced models:	Reduces to a uniform distribution when $c = 0$. Reduces to the symmetric quad when $c = 2$, the symmetric quart when c is 4, and the symmetric sextic when $c = 6$.
References:	Christensen (1985)
See also:	POWER



HYPERBOLICSECANT

This is the hyperbolic secant distribution. The distribution is symmetric about a .

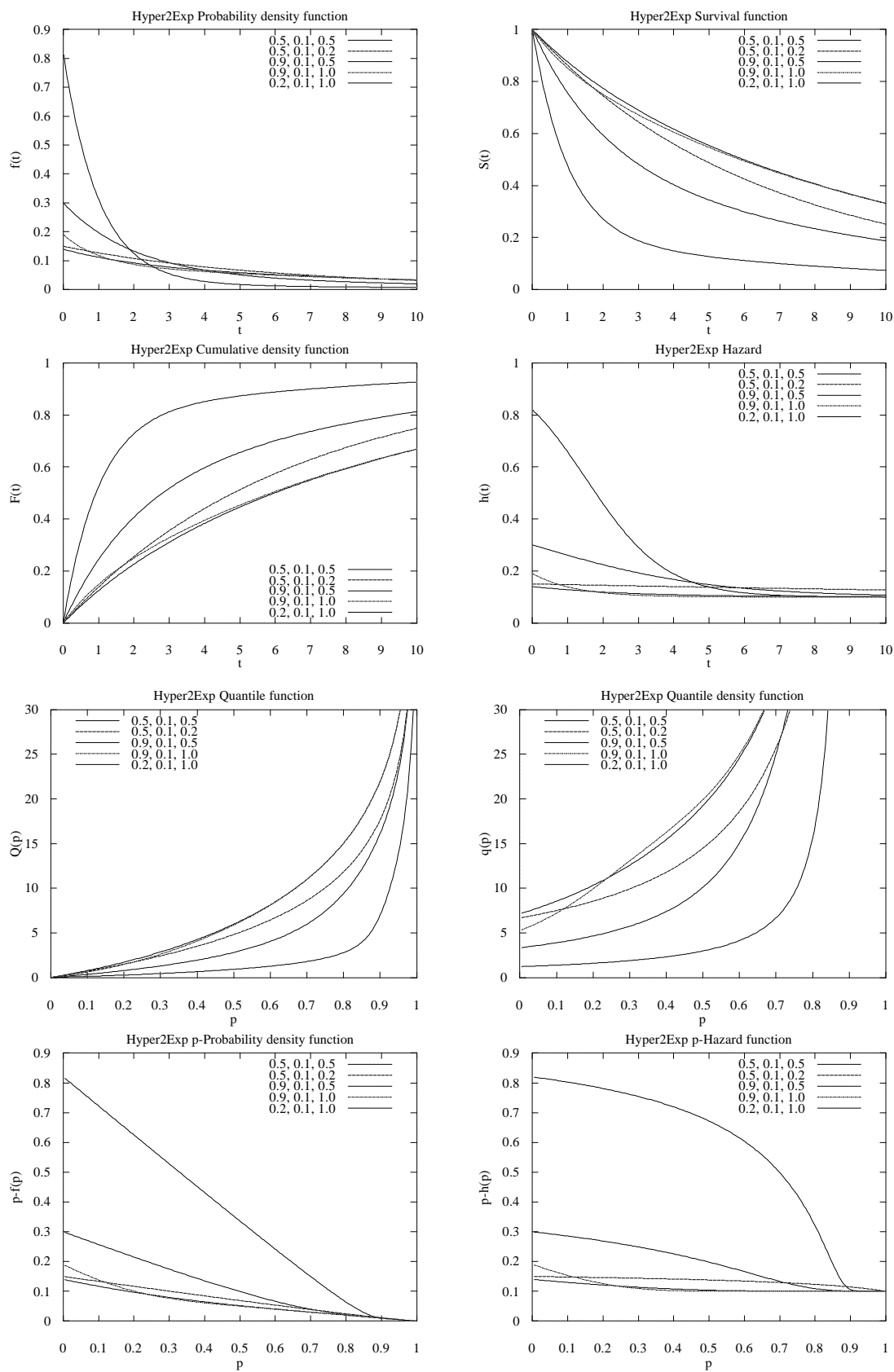
Parameters:	a (location), b (scale)
Constraints:	$b > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{1}{\pi b} \operatorname{sech}\left(\frac{t-a}{b}\right)$
SDF:	$S(t) = 1 - \frac{2}{\pi} \arctan\left[\exp\left(\frac{t-a}{b}\right)\right]$
Quantile:	$t_q = a + b \ln\left[\tan\left(\frac{\pi q}{2}\right)\right]$
Mean:	a
Median:	a
Mode:	a
Variance:	$b^2\pi^2/4$
References:	Christensen (1984), Perks (1932), Talacko (1956)



HYPER2EXP

This is a two-point hyperexponential (or mixed-exponential) distribution. It arises when the entire system fails when either of two constant-hazard components fails. The distribution has been used as a model of fetal loss (Wood 1994; Holman 1996) and CPU service times (Kishor et al. 1982), and many other systems.

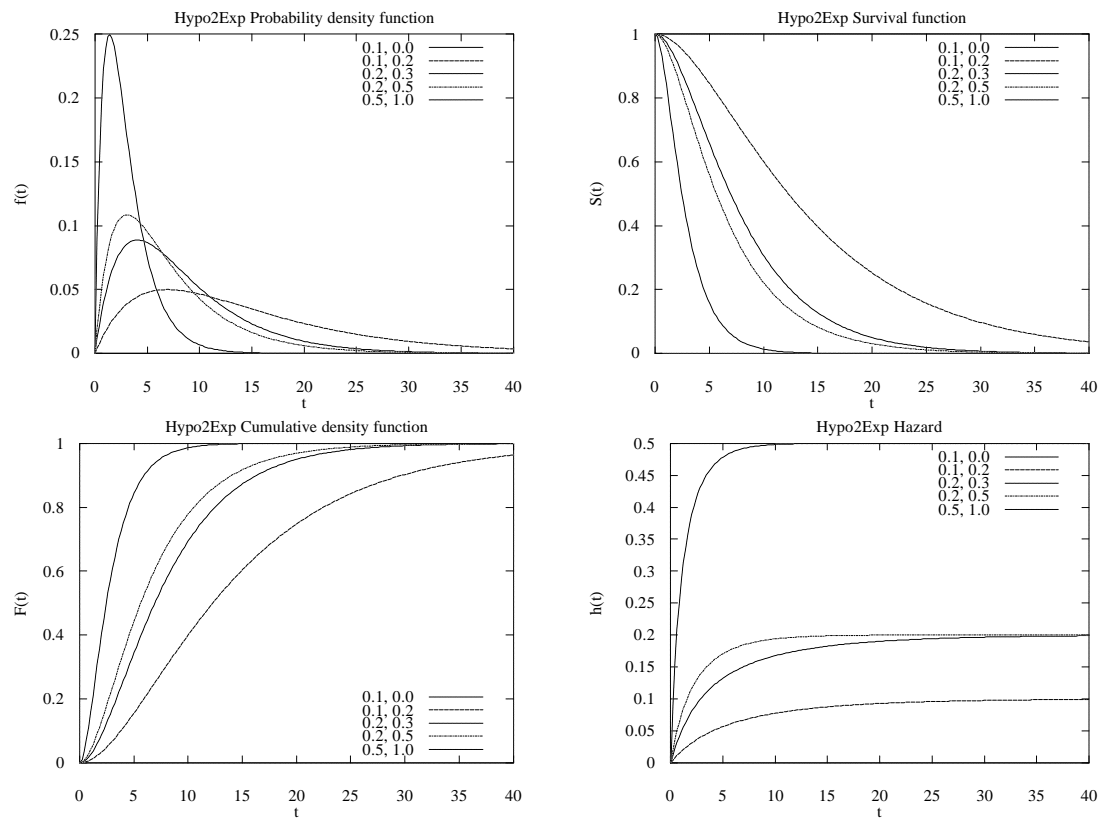
Parameters:	p (initial proportion in subgroup 1); λ_1 (constant hazard in subgroup 1); λ_2 (constant hazard in subgroup 2).
Constraints:	$0 \leq p \leq 1$; $\lambda_1 \geq 0$; $\lambda_2 \geq 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = p\lambda_1 \exp(-\lambda_1 t) + (1 - p)\lambda_2 \exp(-\lambda_2 t)$
SDF:	$S(t) = p \exp(-\lambda_1 t) + (1 - p) \exp(-\lambda_2 t)$
Hazard:	$h(t) = \frac{p\lambda_1 \exp(-\lambda_1 t) + (1 - p)\lambda_2 \exp(-\lambda_2 t)}{p \exp(-\lambda_1 t) + (1 - p) \exp(-\lambda_2 t)}$
Mean:	$\frac{p}{\lambda_1} + \frac{(1 - p)}{\lambda_2}$
Mode:	0
Variance:	$\frac{2p}{\lambda_1^2} + \frac{2(1 - p)}{\lambda_2^2} - \left[\frac{p}{\lambda_1} + \frac{(1 - p)}{\lambda_2} \right]^2$
Reduced models:	When $\lambda_1 = \lambda_2$ and p is fixed to any value between 0 and 1, the PDF is exponential with parameter λ_1 .
Notes:	The distribution is frequently parameterized as $\sigma_1 \rightarrow 1/\lambda_1$, $\sigma_2 \rightarrow 1/\lambda_2$.
Other names:	mixed exponential distribution, Schuhl distribution.
References:	Johnson et al. (1994); Christensen (1984); Holman (1995); Kishor (1982)
See also:	EXPONENTIAL, HYPO2EXP



HYPO2EXP

This is a 2-point hypoexponential distribution. It describes a two stage process in which two independent exponentially distributed components must both fail for the entire system to fail. It arises by taking the convolution of two independent and exponentially distributed components. The distribution has been used to describe I/O operations in computer systems (Kishor et al. 1982).

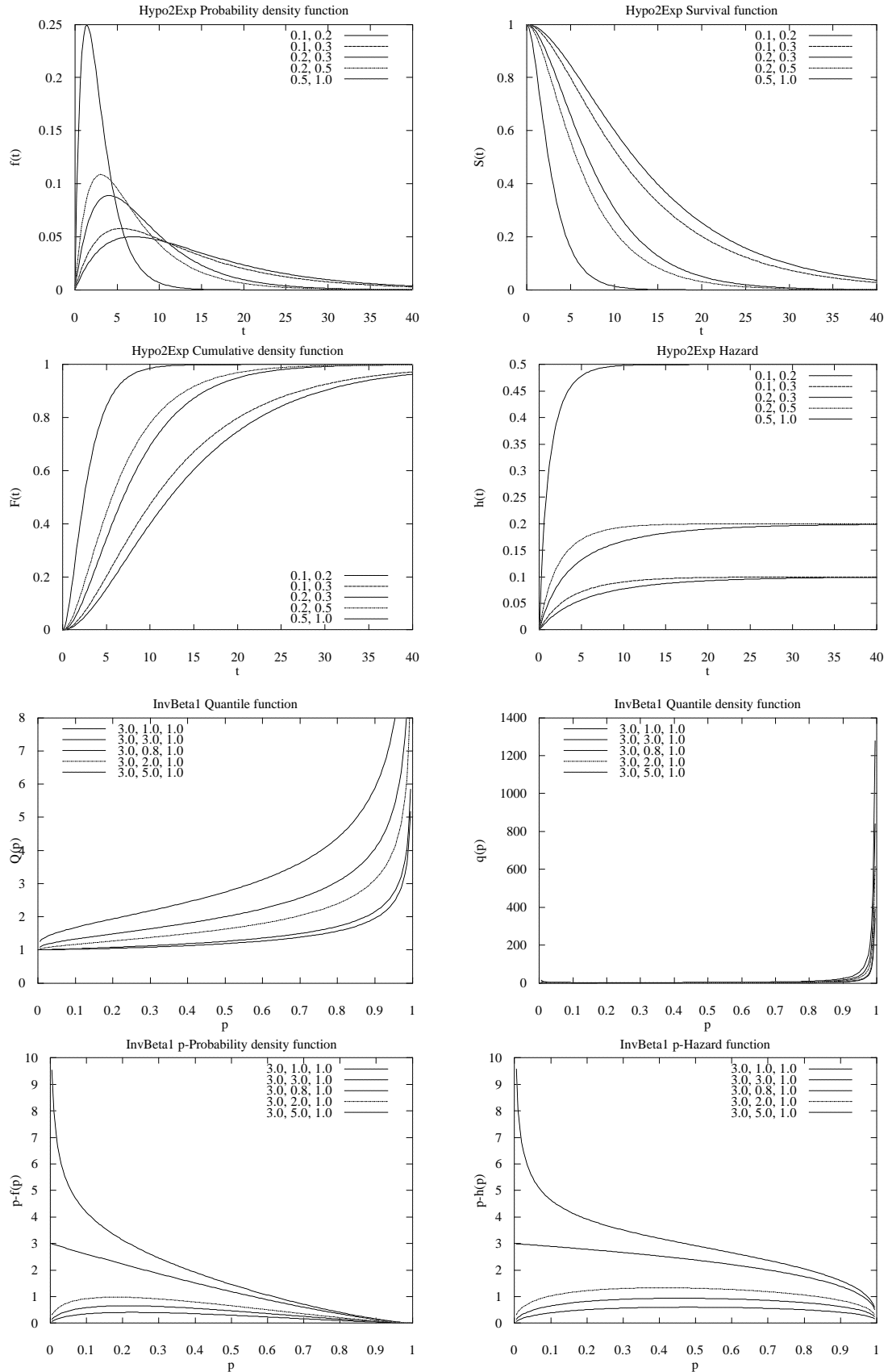
Parameter:	λ_1 (hazard for the first component), λ_2 (hazard for the second component)
Constraints:	$\lambda_1 \geq 0; \lambda_2 \geq 0, \lambda_1 \neq \lambda_2$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{\lambda_1 \lambda_2 [\exp(-\lambda_1 t) - \exp(-\lambda_2 t)]}{\lambda_2 - \lambda_1}$
SDF:	$S(t) = \frac{\lambda_2 \exp(-\lambda_1 t) - \lambda_1 \exp(-\lambda_2 t)}{\lambda_2 - \lambda_1}$
Hazard:	$h(t) = \frac{\lambda_1 \lambda_2 [\exp(-\lambda_1 t) - \exp(-\lambda_2 t)]}{\lambda_2 \exp(-\lambda_1 t) - \lambda_1 \exp(-\lambda_2 t)}$
Mean:	$\frac{1}{\lambda_1} + \frac{1}{\lambda_2}$
Variance:	$\frac{1}{\lambda_1^2} + \frac{1}{\lambda_2^2}$
References:	Christensen (1984); Kishor (1982)
See also:	EXPONENTIAL; HYPER2EXP
Bugs:	The quantile density function does not work correctly



INVBETA1

This is the inverted beta type 1 distribution.

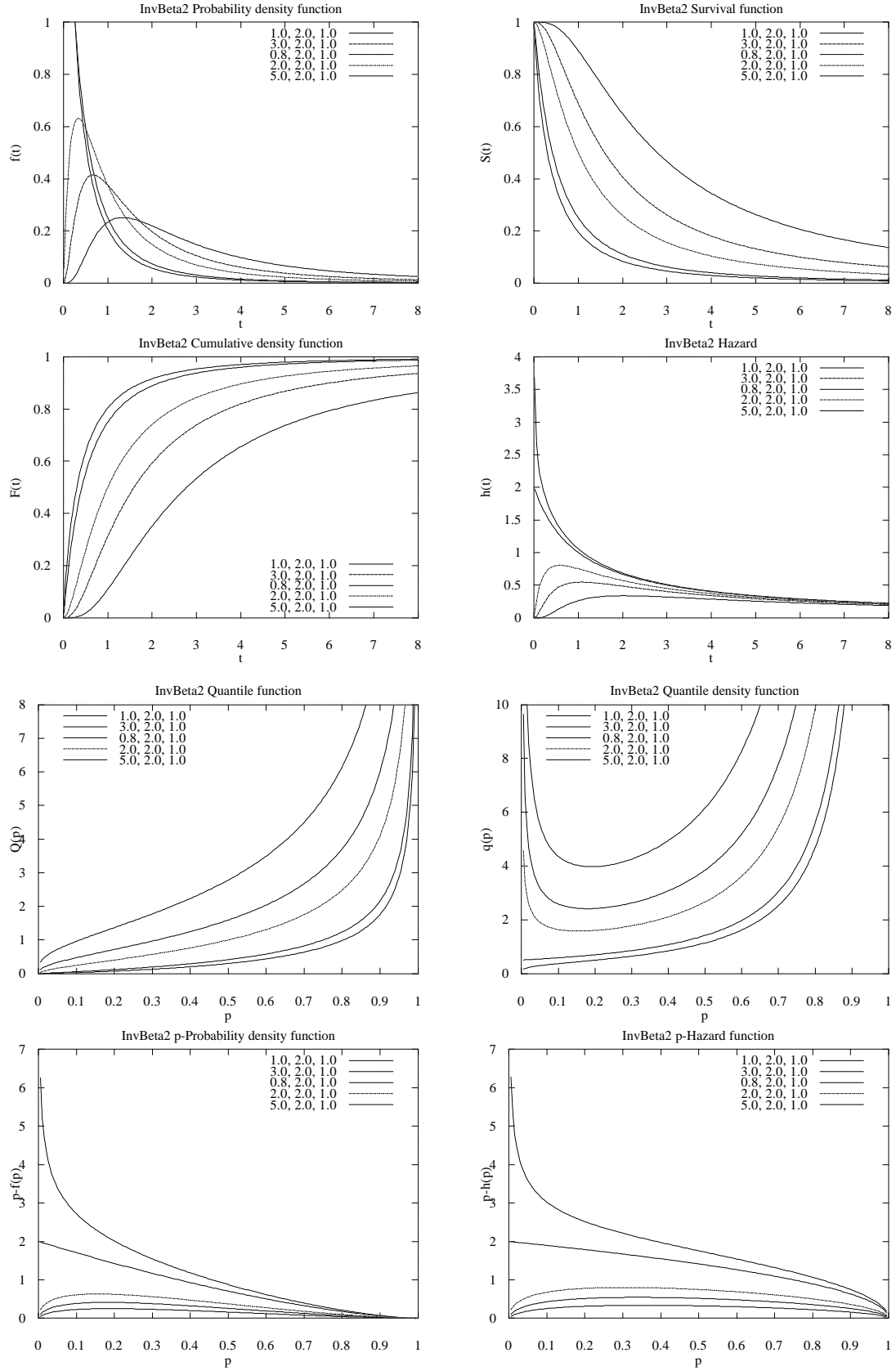
Parameters:	a, b, c
Constraints:	$a \geq 0, b > 0, c \geq 0$
Time variables:	t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq c$
PDF:	$f(t) = \frac{c^a (t-c)^{b-1}}{B(a,b)t^{a+b}}$
SDF:	$S(t) = B_{c/t}(a, b)$
Mean:	$\frac{c(a+b-1)}{a-1} \quad a > 1$
Mode:	$\begin{cases} \frac{c(a+b)}{a+1} & b > 1 \\ c & b \leq 1 \end{cases}$
Variance:	$\frac{bc^2(a+b-1)}{(a-1)^2(a-2)} \quad a > 2$
References:	Christensen (1984); Evans et al. (2000)
See also:	INVBETA2



INVBETA2

This is the inverted beta type 2 distribution.

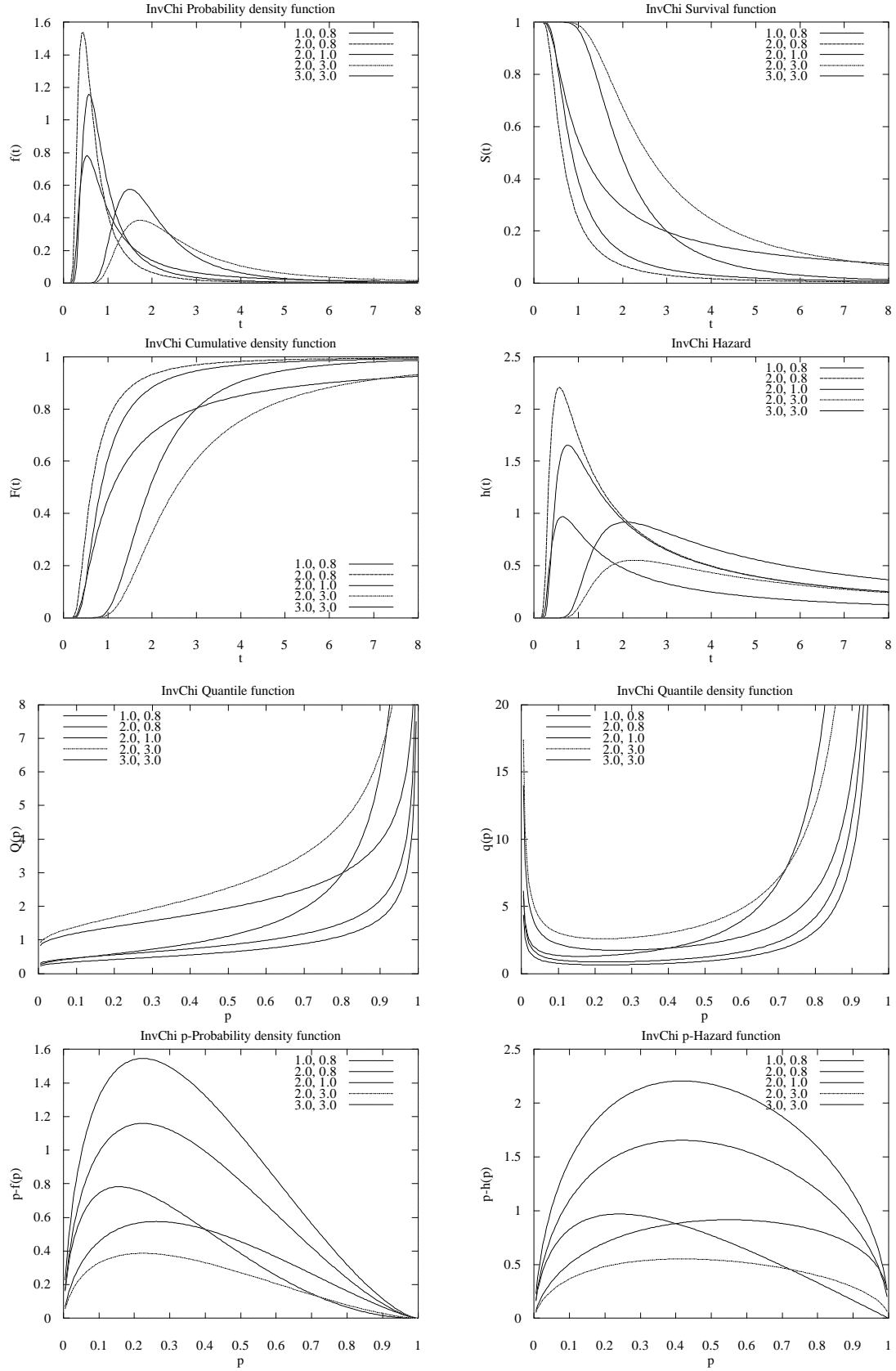
Parameters:	a, b, c
Constraints:	$a > 0, b > 0, c \geq 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{c^b t^{a-1}}{\text{B}(a, b) (t + c)^{a+b}}$
SDF:	$S(t) = \text{B}_{t/(t+c)}(a, b)$
Mean:	$\frac{ca}{b-1} \quad b > 1$
Mode:	$\begin{cases} \frac{c(a-1)}{b+1} & a > 1 \\ 0 & a \leq 1 \end{cases}$
Variance:	$\frac{ac^2(a+b-1)}{(b-1)^2(b-2)} \quad b > 2$
References:	Christensen (1984); Evans et al. (2000)
See also:	INVBETA1



INVCHI

This is the inverted chi distribution.

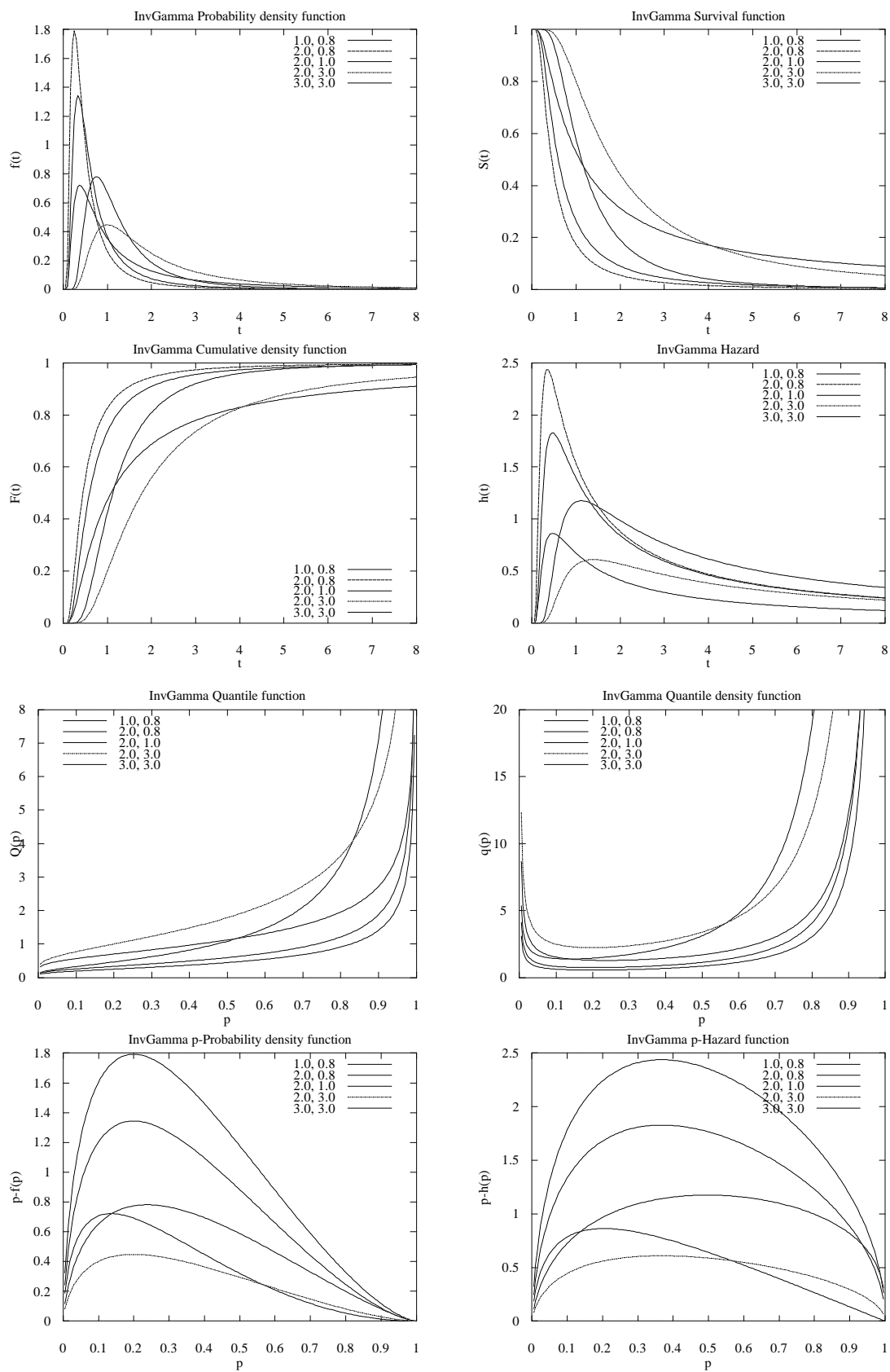
Parameters:	a, b
Constraints:	$a > 0, b > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{e^{-\frac{b^2}{2t^2}} 2^{1-a/2}}{b\Gamma(a/2)} \left(\frac{b}{t}\right)^{a+1}$
SDF:	$S(t) = \frac{\gamma\left[a/2, \left(\frac{b}{4t}\right)^2\right]}{\Gamma(a/2)}$
Mean:	$S(t) = \frac{b\Gamma\left[\frac{a-1}{2}\right]}{\sqrt{2}\Gamma(a/2)} \quad a > 1$
Mode:	$\frac{b}{\sqrt{a+1}}$
Variance:	$S(t) = \frac{b^2}{a-2} - \frac{b^2\Gamma\left[\frac{a-1}{2}\right]^2}{2\Gamma(a/2)^2}, \quad a > 2$
References:	Christensen (1984)
See also:	CHI



INVGAMMA

This is the inverted gamma distribution, also called the Person type V distribution.

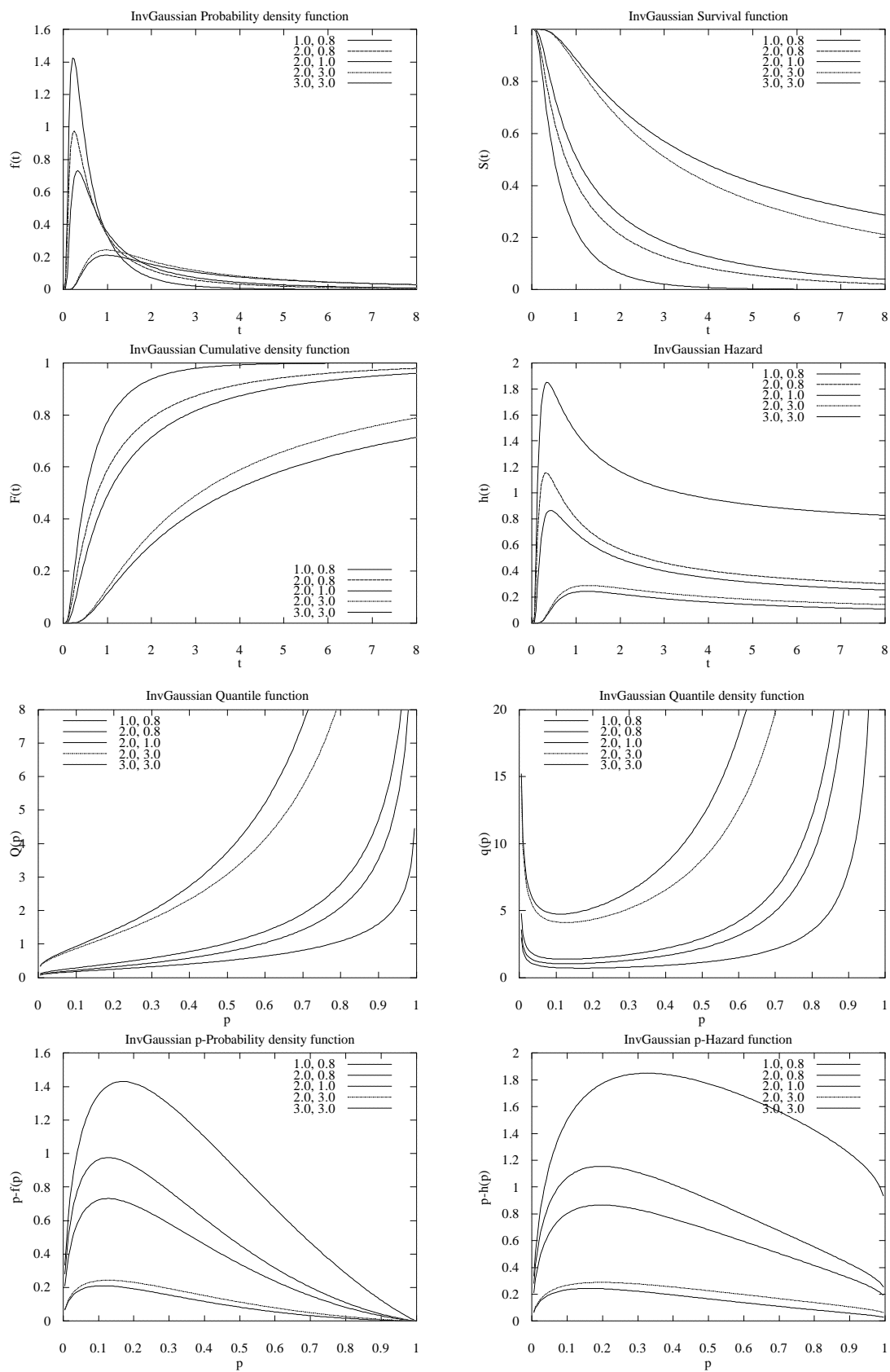
Parameters:	b (scale), c (shape).
Constraints:	$a > 0$, $b > 0$
Time variables:	t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{e^{-b/t}}{b\Gamma(c)} \left(\frac{b}{t}\right)^{c+1}$
SDF:	$S(t) = \frac{\gamma(c, b/t)}{\Gamma(c)}$
Hazard:	$h(t) = \frac{e^{-b/t}}{b\gamma(c, b/t)} \left(\frac{b}{t}\right)^{c+1}$
Quantile:	$t_q = \frac{2b}{\chi^2_{1-q}(2c)}$
Mean:	$b / (c - 1)$, $a > 1$
Mode:	$b / (c + 1)$
Variance:	$b^2(c - 1)^{-2}(c - 2)^{-1}$, $a > 2$
Other names:	Inverse normal distribution.
References:	Christensen (1984); Evans et al. (2000); Pearson (1895)
See also:	GAMMA, INVBETA1, INVBETA2



INVGAUSSIAN

This is the inverse Gaussian distribution, which includes the Wald distribution as a special case. This distribution is a special case of the Pearson Type V distribution.

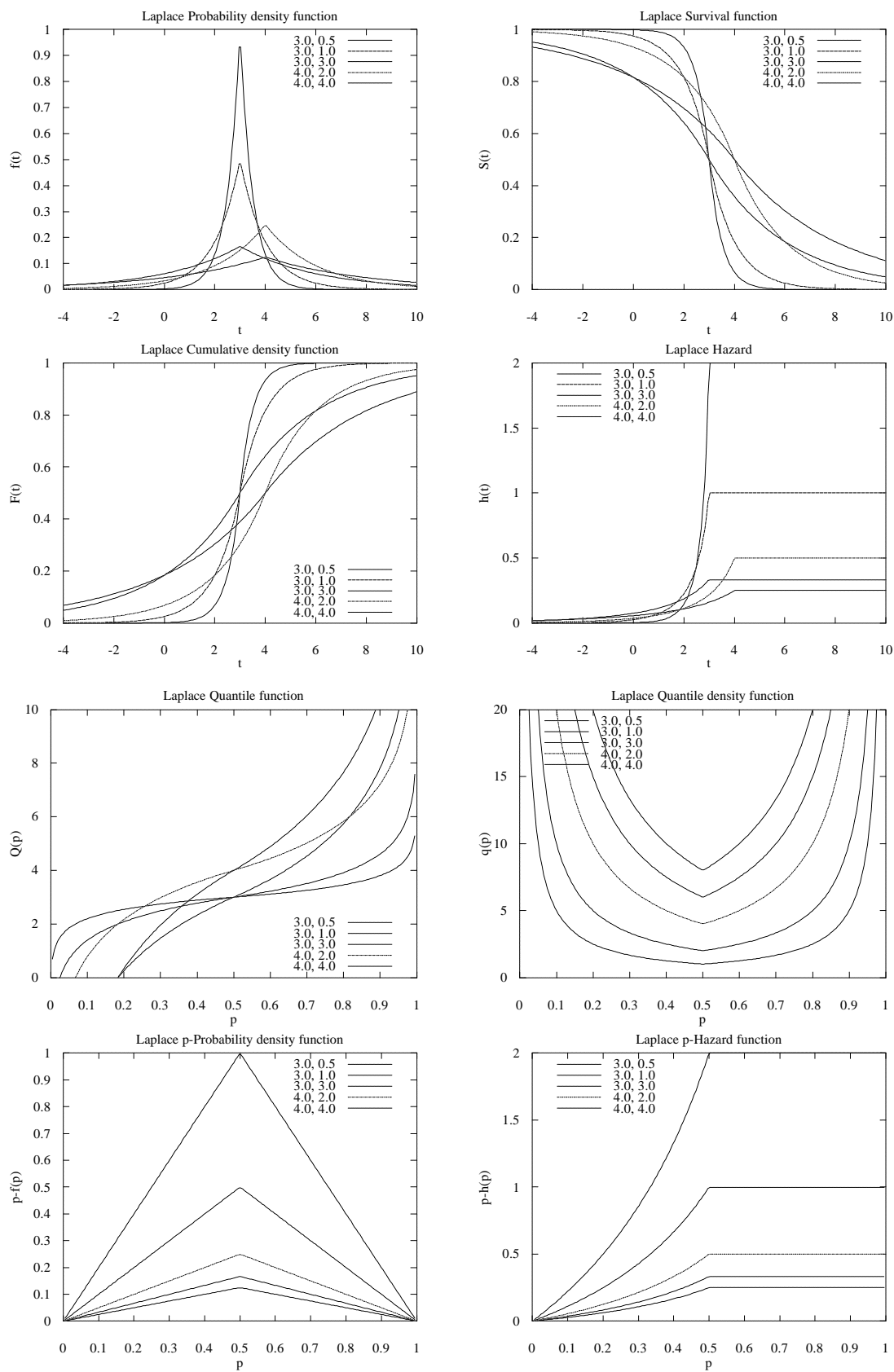
Parameters:	b (scale), c (shape)
Constraints:	$b \geq 0, c \geq 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t > 0$
PDF:	$f(t) = \sqrt{\frac{b}{2\pi t^3}} \exp\left[-\frac{b}{2c^2 t} \left(\frac{t}{b} - c\right)^2\right]$
SDF:	$S(t) = 1 - \Phi\left(\frac{t - bc}{c\sqrt{bt}}\right) - e^{2/c} \Phi\left(\frac{-t - bc}{c\sqrt{bt}}\right)$
Mean:	bc
Mode:	$bc \left(\sqrt{1 + 9c^2/4} + 3c/2\right)$
Variance:	$b^2 c^3$
Reduced models:	Approaches the Normal distribution as $b \rightarrow \infty$. Reduces to the Wald distribution when $b = 1/c$.
References:	Chhikara and Folks (1989), Christensen (1984), Evans et al. (2000), Folks and Chhikara (1978), Jørgensen (1982), Schrödinger (1915), Tweedie (1947), Wald (1947)
See also:	RANDOMWALK



LAPLACE

This is the Laplace distribution, also known as a double-exponential distribution. The distribution is discontinuous at a , and declines to the left and right exponentially.

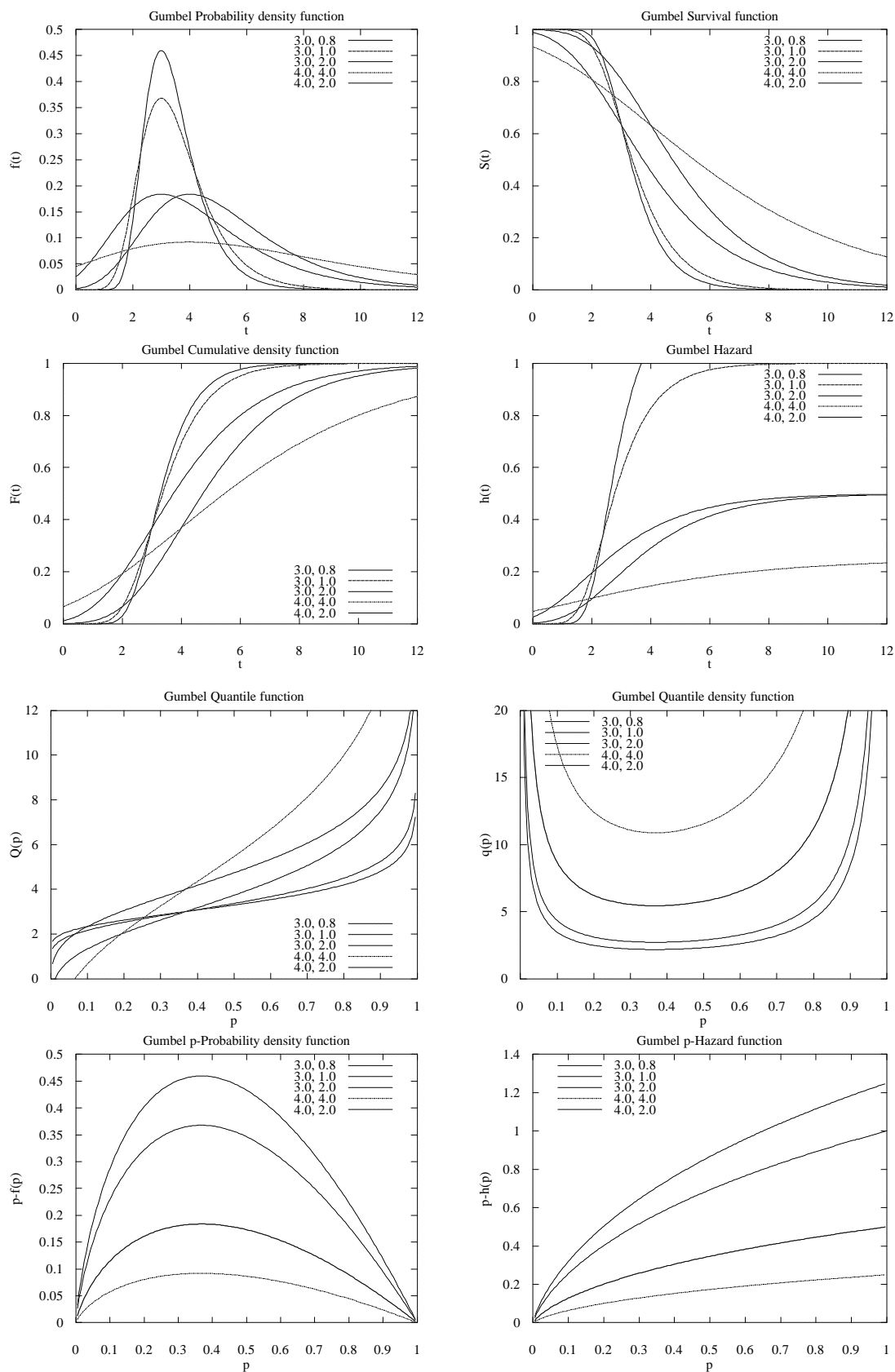
Parameters:	a (location), b (scale)
Constraints:	$b \geq 0$
Time variables:	$t_u, t_e, t_{\alpha}, t_{\omega}$. An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{e^{-\frac{ t-a }{b}}}{2b}$
SDF:	$S(t) = \begin{cases} \frac{1}{2} e^{-\frac{t-a}{b}}, & t \geq a \\ 1 - \frac{1}{2} e^{-\frac{a-t}{b}}, & t < a \end{cases}$
Mean:	a
Median:	a
Mode:	a
Variance:	$2b^2$
Other names:	Bilateral exponential, double exponential, Laplace's first law of error, two-tailed exponential.
References:	Christensen (1984), Evans et al. (2000), Laplace (1774)
See also:	EXPONENTIAL, SUBBOTIN



LARGEEXTREME1 and GUMBEL

This is the type 1 largest extreme value distribution. This is also one type of Gumbel distribution.

Parameters:	a (location) and b (scale)
Constraints:	$b \geq 0$
Time variables:	$t_u, t_e, t_{\alpha}, t_{\omega}$. An exact failure is defined when $t_u = t_e$.
Range:	$-\infty < t < \infty$
PDF:	$f(t) = \frac{1}{b} \exp \left[-\frac{t-a}{b} - \exp \left(-\frac{t-a}{b} \right) \right]$
SDF:	$S(t) = 1 - \exp \left\{ -\exp \left(-\frac{t-a}{b} \right) \right\}$
Hazard:	$h(t) = \frac{\exp \left(-\frac{t-a}{b} \right)}{b \left[\exp \left[-\exp \left(-\frac{t-a}{b} \right) \right] - 1 \right]}$
Mean:	$a - kb$ $k \approx 0.57721...$ is Euler's constant
Median:	$a - b \log[\log(2)]$
Mode:	a (≈ 36.8 th percentile)
Variance:	$b^2 \pi^2 / 6$
References:	Evans et al. (2000), Johnson et al. (1994), Nelson (1982)
See also:	SMALLEXTREME1, LARGEEXTREME2, WEIBULL



LARGEEXTREME2

This is the type 2 largest extreme value distribution, also known as the Frechet distribution.

Parameters: a (location), b (scale), c (shape)

Constraints: $b \geq 0, c \geq 0$

Time variables: $t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.

Range: $a \leq t < \infty$

PDF:
$$f(t) = c \left(\frac{t-a}{b} \right)^{(-c-1)} \exp \left[- \left(\frac{t-a}{b} \right)^{-c} \right]$$

SDF:
$$S(t) = 1 - \exp \left[- \left(\frac{t-a}{b} \right)^{-c} \right]$$

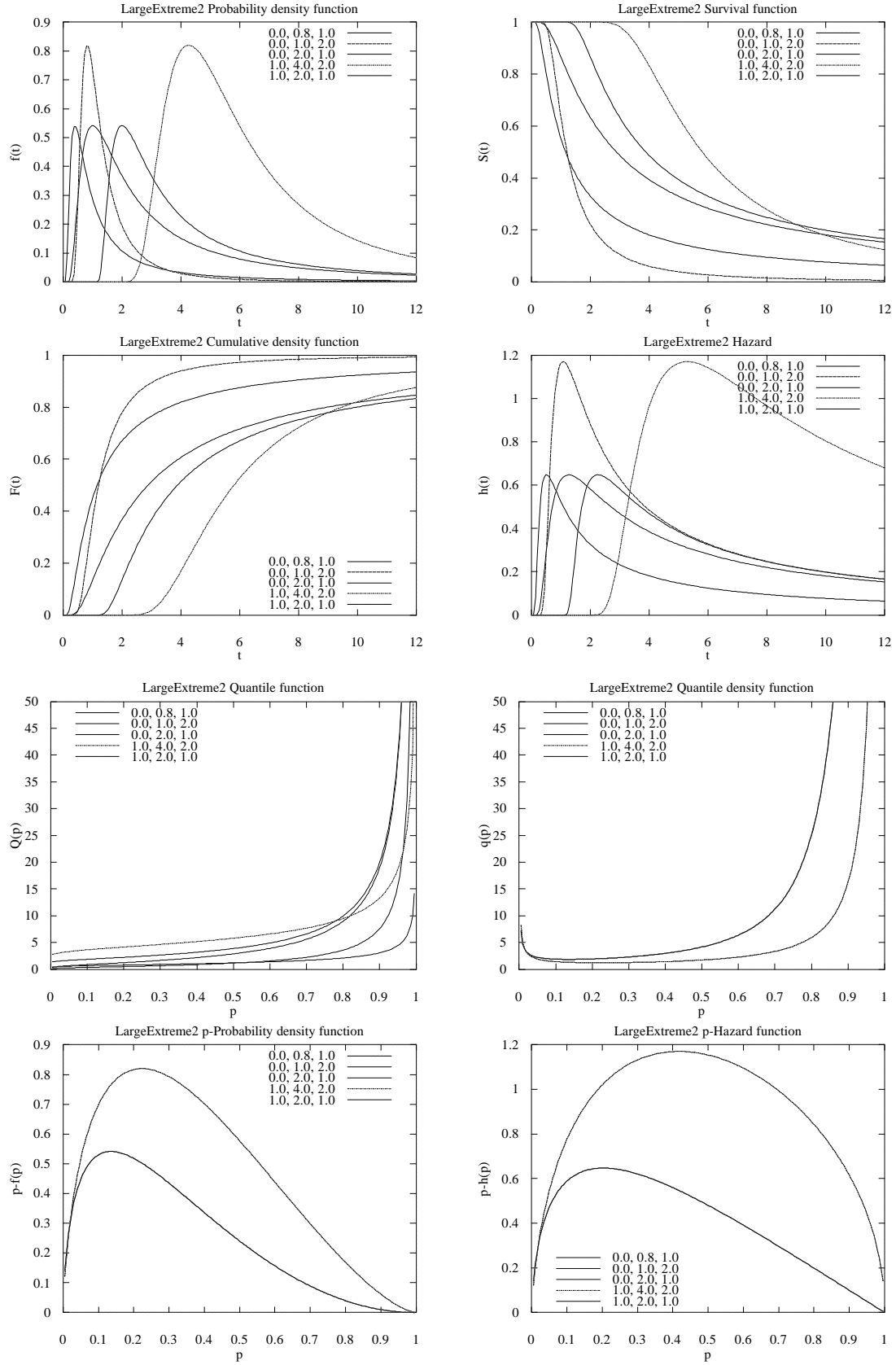
Hazard:
$$h(t) = \frac{c \left(\frac{t-a}{b} \right)^{-c-1} \exp \left[- \left(\frac{t-a}{b} \right)^{-c} \right]}{1 - \exp \left[- \left(\frac{t-a}{b} \right)^{-c} \right]}$$

Quantile: $t_q = a + b[-\ln(q)]^{-1/c}$

Median: $a + b[\ln(2)]^{-1/c}$

References:

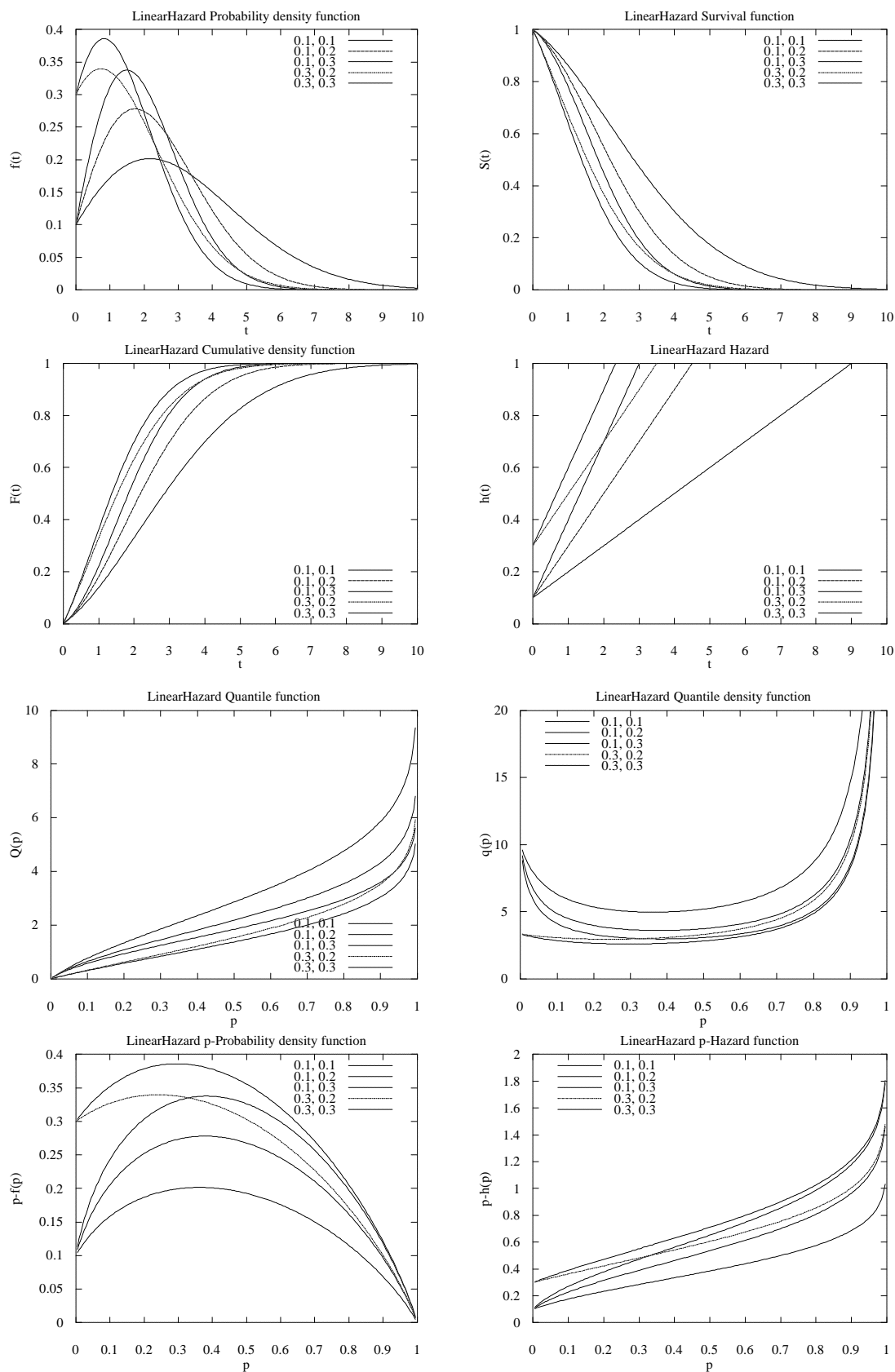
See also: LARGEEXTREME1, SMALLEXTREME2



LINEARHAZARD

The linear hazard rate distribution is so-called because the hazard increases monotonically with time.

Parameters:	λ (constant hazard), b (scale).
Constraints:	$\lambda \geq 0, b > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = (\lambda + bt) \exp(-\lambda t - \frac{1}{2}bt^2)$
SDF:	$S(t) = \exp(-\lambda t - \frac{1}{2}bt^2)$
Hazard:	$h(t) = \lambda + bt$
Other names:	Truncated Rayleigh
References:	Lee (1992); Christensen (1984)



LNGAMMA

This is the log-gamma distribution.

Parameters: a (location), b (scale, inverse of the hazard), c (shape)

Constraints: $b \geq 0, c \geq 0$

Time variables: t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.

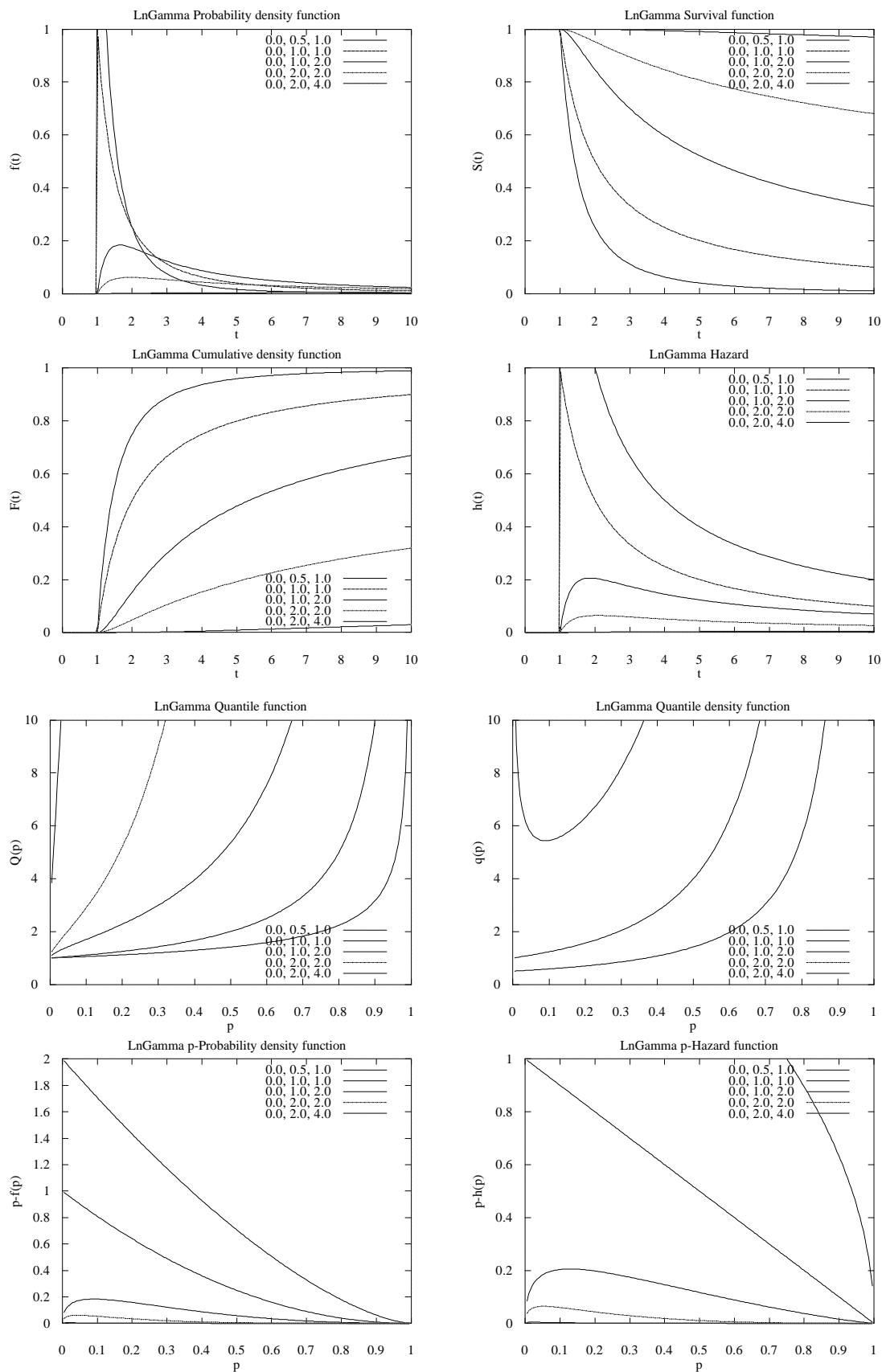
Range: $t \geq a$

PDF:
$$f(t) = \frac{[\ln(t) - a]^{c-1} e^{-\frac{\ln(t)-a}{b}}}{b^c \Gamma(c) t}$$

SDF:
$$S(t) = \frac{\Gamma\left(c, \frac{\ln(t)-a}{b}\right)}{\Gamma(c)}$$

References:

See also: GAMMA



LNLOGISTIC

This is the log-logistic distribution.

Parameters: a (location), b (scale), c (shape).

Constraint: $b > 0, c > 0$

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$

Range: $-\infty < t < \infty$

PDF:
$$f(t) = \frac{\frac{c}{b} \left(\frac{t-a}{b} \right)^{c-1}}{\left[1 + \left(\frac{t-a}{b} \right)^c \right]^2}$$

SDF:
$$S(t) = 1 - \left[1 + \left[\frac{t-a}{b} \right]^c \right]^{-1}$$

Quantile:
$$t_q = a + b \sqrt[c]{\frac{q}{1-q}}$$

Mean:
$$a + b \left[\frac{\pi}{c} \csc \left(\frac{\pi}{c} \right) \right]$$

Median:
$$a + b$$

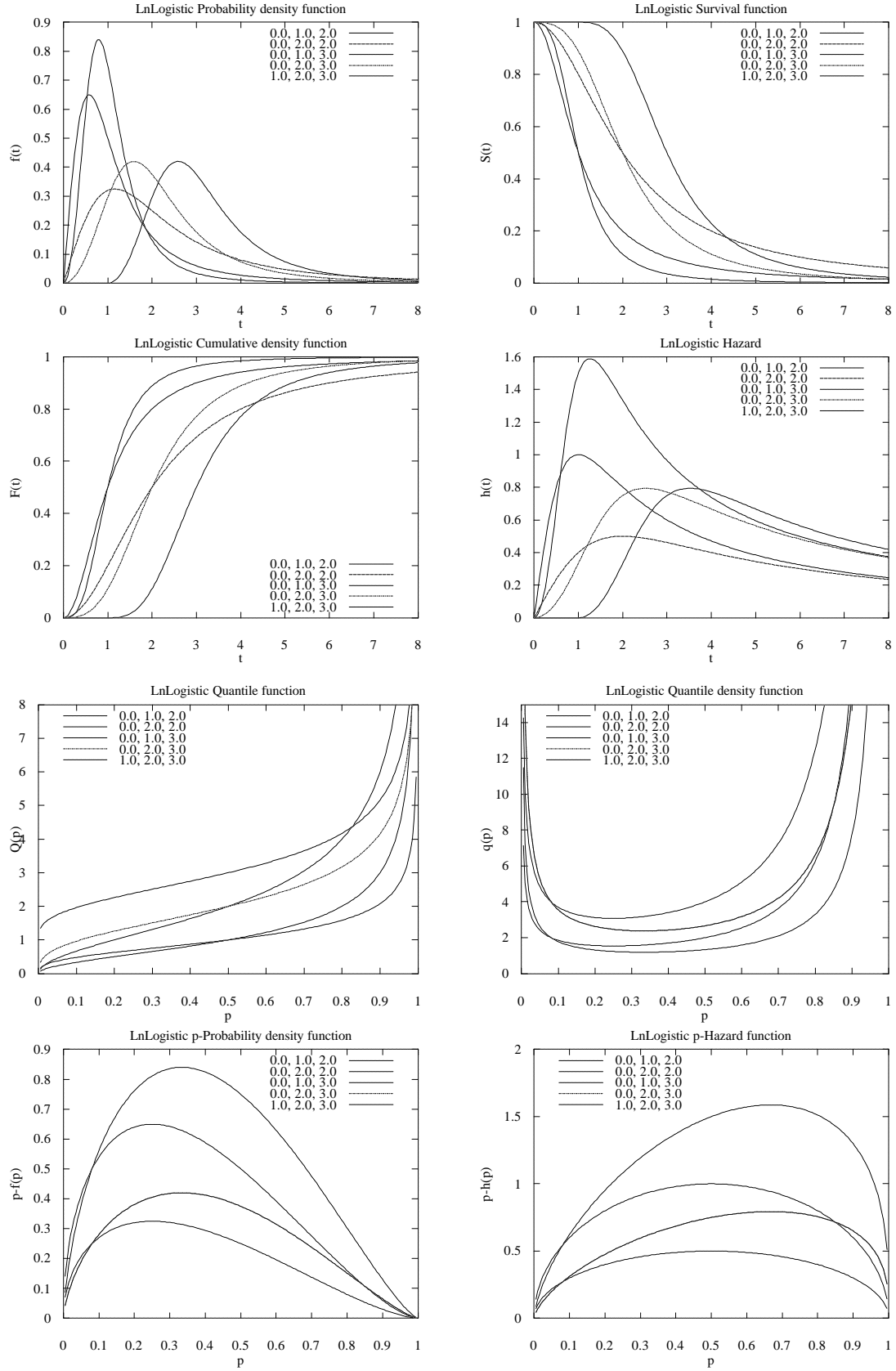
Mode:
$$a + b \sqrt[c]{\frac{c-1}{c+1}}$$

Notes: This distribution is sometimes reparameterized so that $c' \rightarrow c^{-1}$, $b' \rightarrow e^b$, and $c \rightarrow 0$.

References: Johnson et al. (1995); Shah and Dave (1963)

Other names: Fisk

See also: LOGISTIC



LOGISTIC

The logistic distribution.

Parameters: a (location), b (scale).

Constraint: $b > 0$

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$

Range: $-\infty < t < \infty$

PDF:
$$f(t) = \frac{\exp\left(-\frac{t-a}{b}\right)}{b \left[1 + \exp\left(-\frac{t-a}{b}\right)\right]^2}$$

$$= \frac{1}{4b} \operatorname{sech}^2\left(\frac{t-a}{2b}\right)$$

SDF:
$$S(t) = \left\{1 + \exp\left[\frac{t-a}{b}\right]\right\}^{-1}$$

$$= \frac{1}{2} \left[1 - \tanh\left(\frac{t-a}{2b}\right)\right]$$

Hazard:
$$h(t) = \frac{1}{b} \left\{1 + \exp\left[-\frac{t-a}{b}\right]\right\}^{-1}$$

Quantile:
$$t_q = a - b \ln\left(\frac{1-q}{q}\right)$$

Mean: a

Median: a

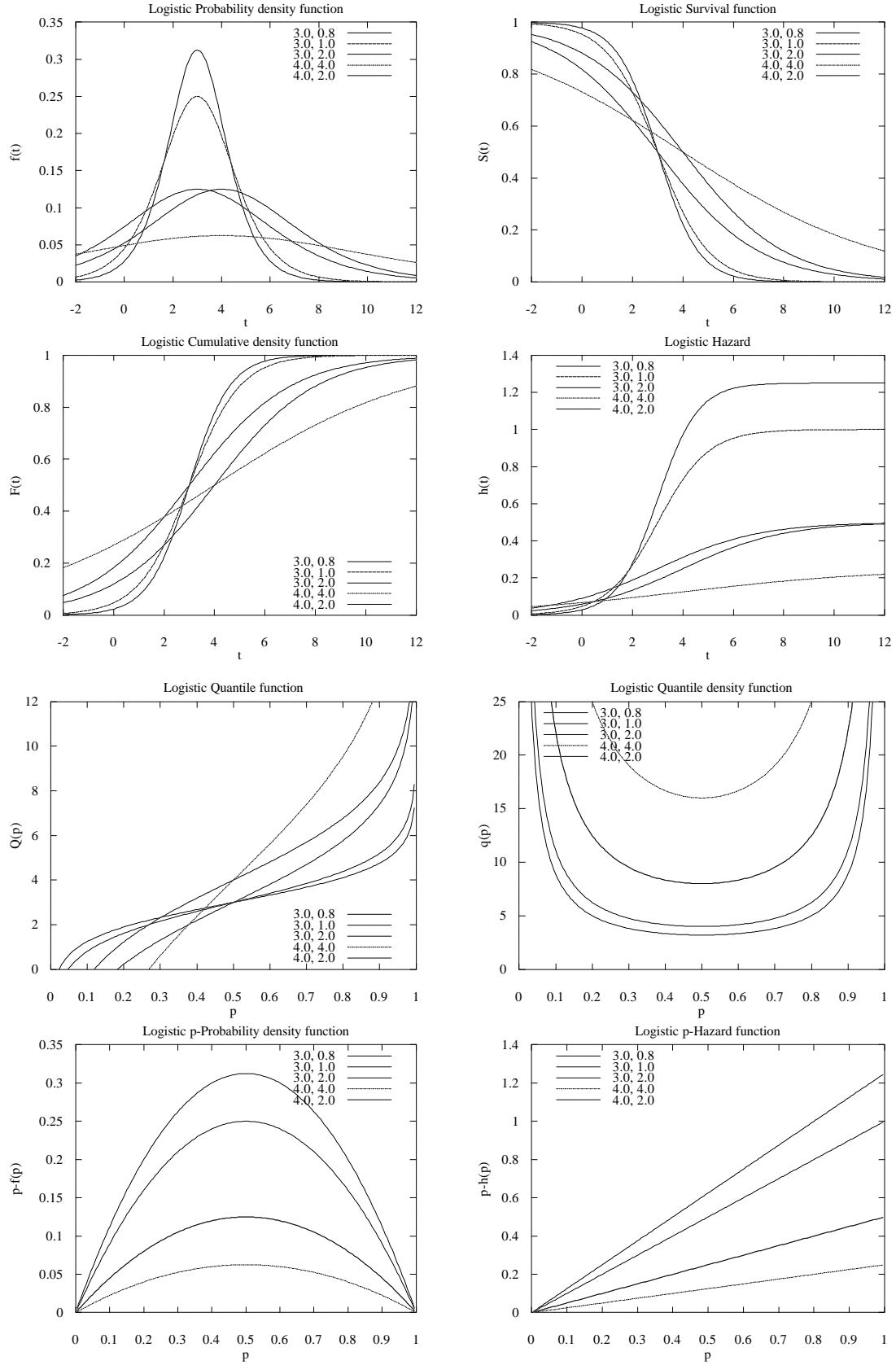
Mode: a

Variance: $\pi^2 b^2 / 3$

Other names: sech-squared distribution.

References: Johnson et al. (1995); Christensen (1984); Evans et al. (2000)

See also: LNLOGISTIC



LOGNORMAL or LNNORMAL

This is the lognormal distribution.

Parameters: a (location), b (scale).

Constraints: $b > 0$

Time variables: $t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$

Range: $t \geq a$

PDF:
$$f(t) = \frac{1}{tb\sqrt{2\pi}} e^{\left[\frac{-(\ln(t)-a)^2}{2b^2} \right]}$$

SDF:
$$S(t) = 1 - \Phi \left[\frac{\ln(t) - a}{b} \right]$$

Quantile: $t_q = \exp[a + b \Phi_q(q)]$

Mean: $\exp(a + b^2/2)$

Median: $\exp(a)$

Mode: $\exp(a)/\exp(b^2)$

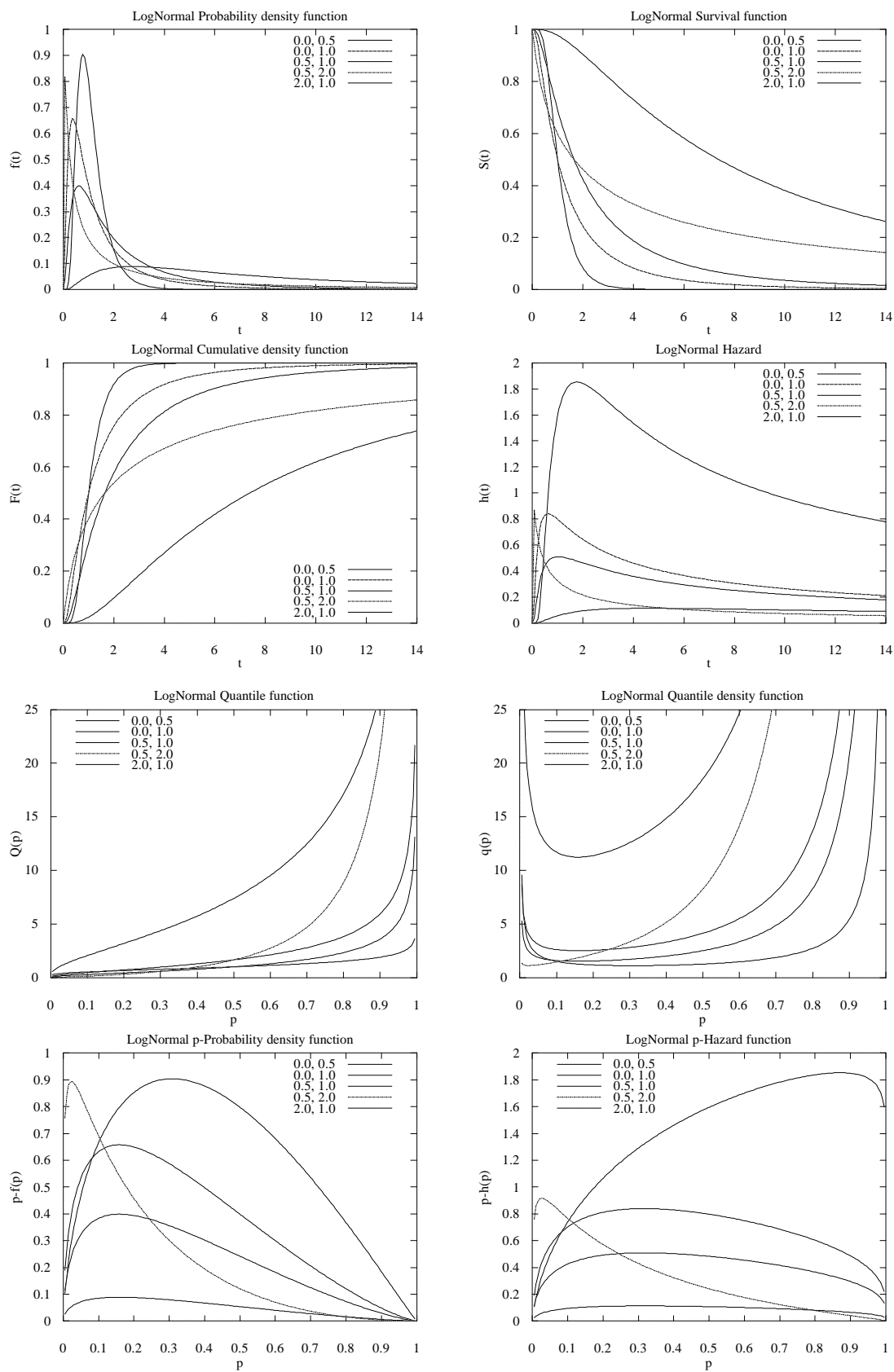
Variance: $\exp(2a + b^2)[\exp(b^2) - 1]$

Notes: This distribution is related to the SHIFTLOGNORMAL distribution as follows. The two-parameter LNNORMAL cumulative density is found from the Normal density by taking $\Phi\{[\ln(t) - a]/b\}$ whereas in the three parameter SHIFTLOGNORMAL we take $\Phi\{\ln[(t - a)/b]/c\}$.

Other names: The lognormal is sometimes called the antilognormal, logarithmico-normal, Cobb-Douglas, the Galton-McAlister, van Uven, or the Kapteyn-Gibrat distributions.

References: Johnson et al. (1994), Evans et al. (2000), Nelson (1982)

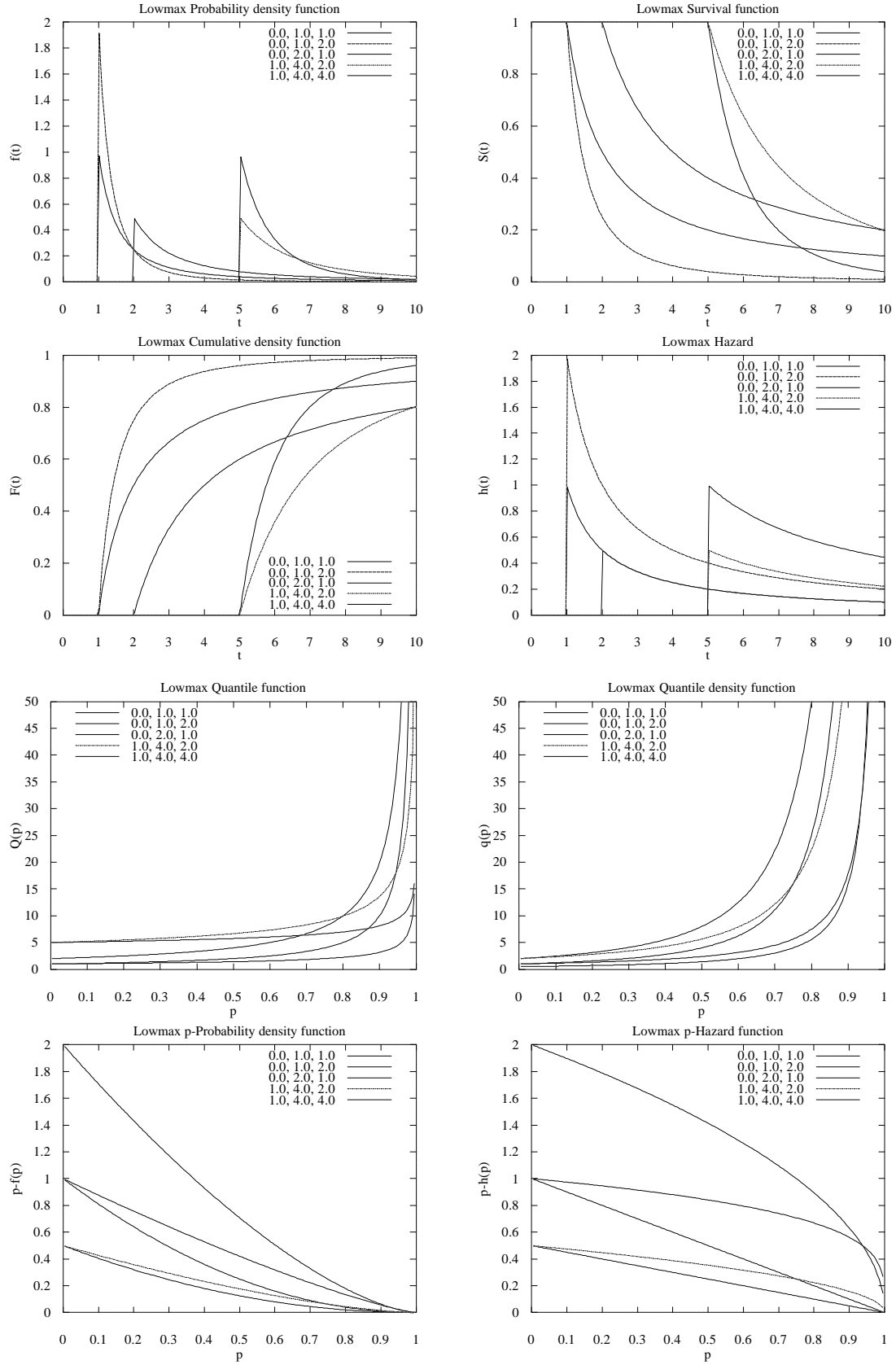
See also: NORMAL, SHIFTLOGNORMAL



LOWMAX

This is the Lowmax distribution, which is a generalized Pareto distribution.

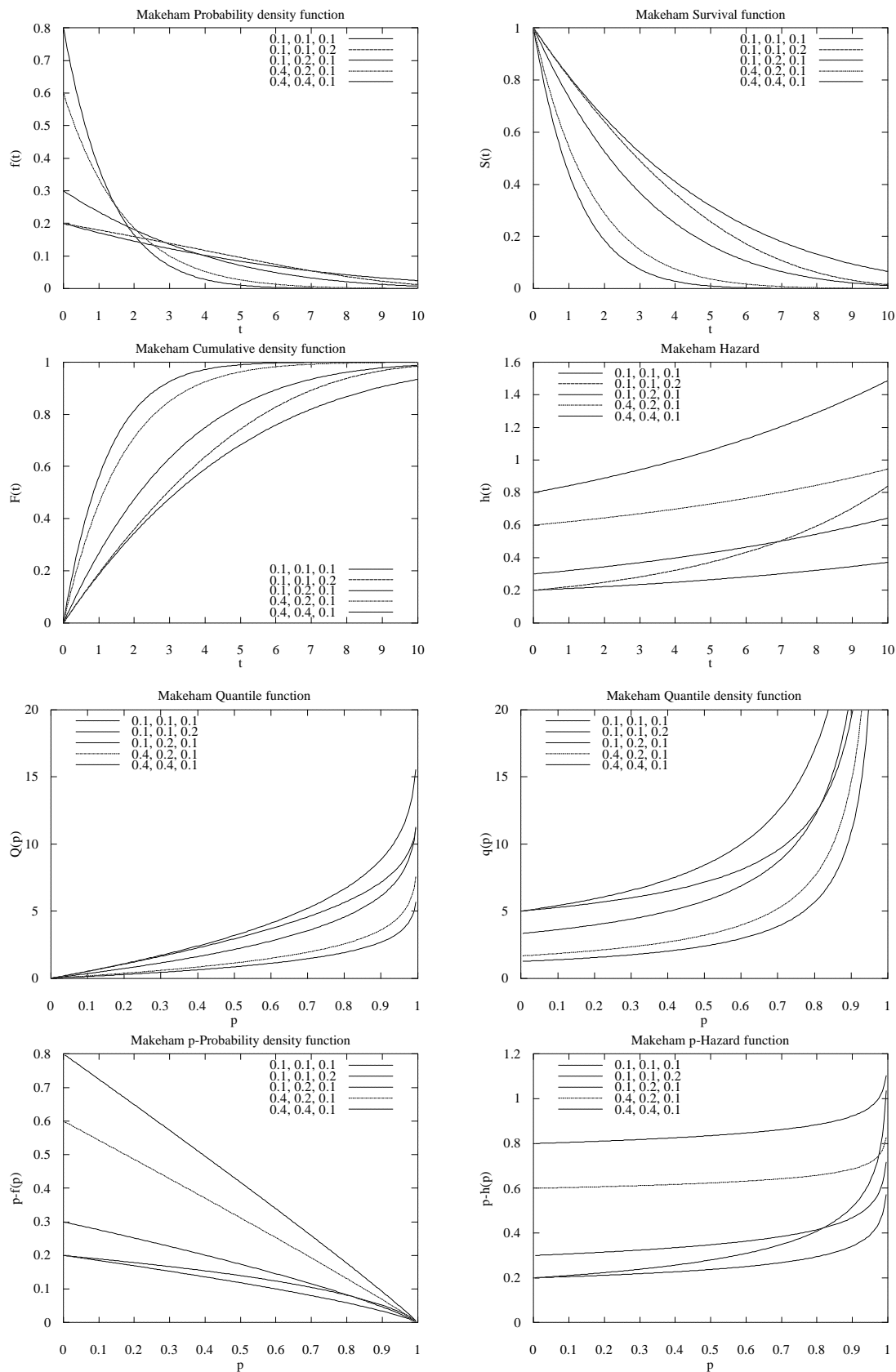
Parameters:	a (location), b (scale), c (shape).
Constraints:	$b \geq 0, c > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$a + b \leq t < \infty$
PDF:	$f(t) = \frac{b^c c}{(t-a)^{c+1}}$
SDF:	$S(t) = [(t-a)/b]^{-c}$
Quantile:	$t_q = a + b(1-q)^{-1/c}$
Mean:	$a + bc/(c-1), \quad c > 1$
Median:	$a + b\sqrt[2]{2}$
Mode:	$a + b$
Variance:	$\frac{b^2 c}{(c-2)(c-1)^2}, \quad c > 2$
Reduced models:	Reduces to the Pareto distribution when $a = 0$
Other names:	Pareto distribution of the second kind, Pearson type VI.
References:	Johnson et al. (1994); Christensen (1984)
See also:	PARETO



MAKEHAM

This is the Makeham-Gompertz (also called the Gompertz-Makeham) distribution frequently used as a competing hazards model for mortality. The a_1 parameter is interpreted as accidental mortality component, and the a_2 and b parameters make up the senescent mortality component.

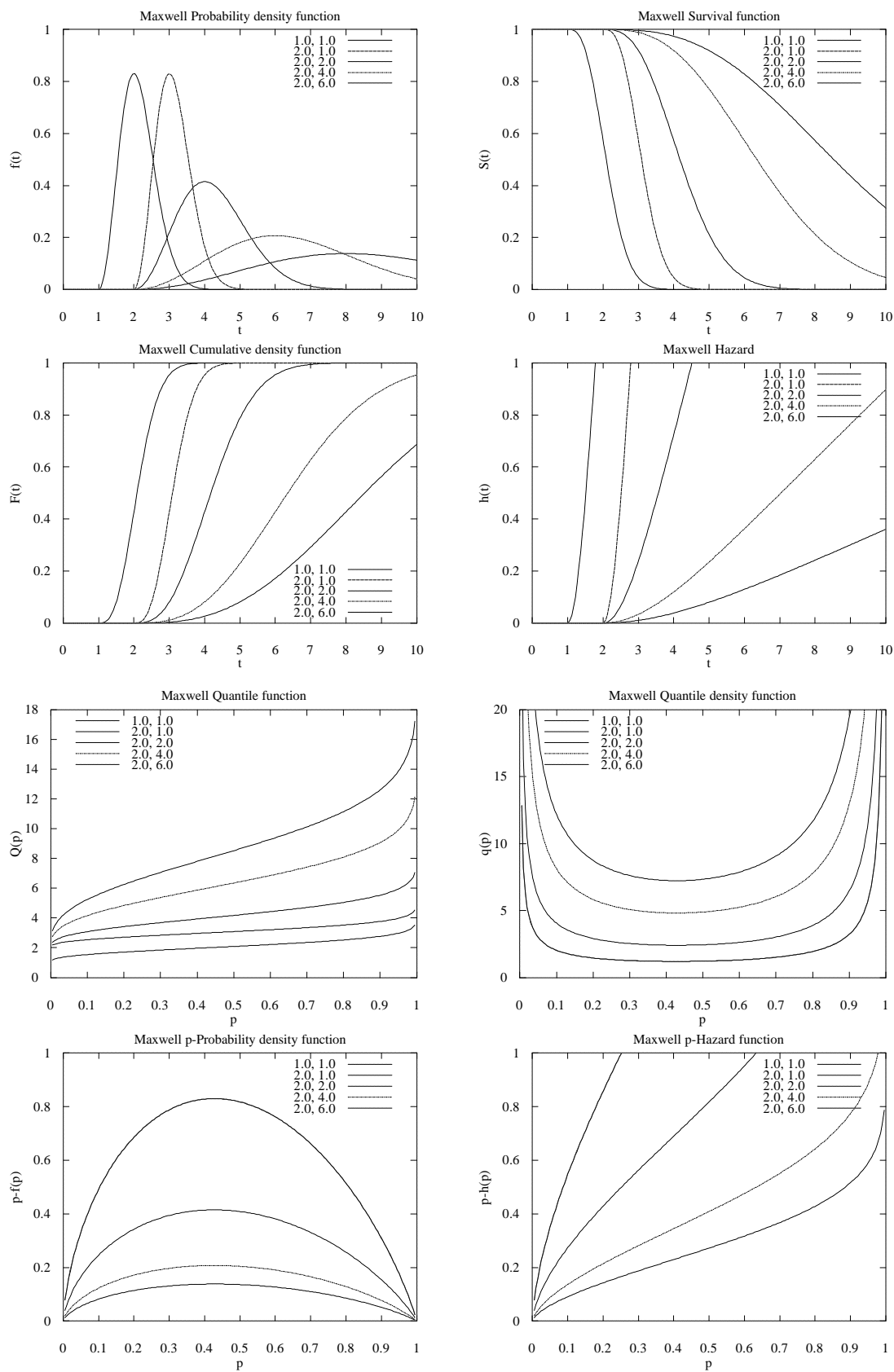
Parameters:	a_1, a_2 and b
Constraints:	$a_1 \geq 0, a_2 \geq 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = (a_1 + a_2 e^{bt}) \exp \left[-a_1 t + \frac{a_2}{b} (1 - e^{bt}) \right],$
SDF:	$S(t) = \exp \left[-a_1 t + \frac{a_2}{b} (1 - e^{bt}) \right]$
Hazard:	$h(t) = a_1 + a_2 \exp(bt)$
Reduced models:	$a_1 = 0$ reduces to a Gompertz PDF with parameters a_2 and b . $b = 0$ and either a_1 or a_2 are constrained, reduces to an exponential with parameter $a_1 + a_2$.
References:	Elandt-Johnson and Johnson (1980)
See also:	EXPONENTIAL, GOMPERTZ, MIXMAKEHAM, SILER



MAXWELL

This is the two-parameter Maxwell-Boltzmann distribution. It is used to model the distribution of particles at equilibrium in statistical mechanics.

Parameters:	a (location), b (scale).
Constraints:	$b \geq 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \frac{4}{b\sqrt{\pi}} \left(\frac{t-a}{b} \right)^2 e^{-\left(\frac{t-a}{b}\right)^2},$
SDF:	$S(t) = \operatorname{erf}\left(\frac{t-a}{b}\right) - \frac{2}{\sqrt{\pi}} \left(\frac{t-a}{b} \right) e^{-\left(\frac{t-a}{b}\right)^2},$
Mean:	$a + \frac{2b}{\sqrt{\pi}}$
Mode:	$a + b$
Variance:	$b^2 \left(\frac{3}{2} - \frac{4}{\pi} \right)$
References:	Christensen (1984), Maxwell (1860a,b), Rao (1973)
See also:	RAYLEIGH, CHISQUARED, CHI



MIXMAKEHAM

This is the mixed-Makeham distribution, which can be used to model the human lifespan

Parameters: p (initial proportion in risk group 1), λ_1 (constant hazard in subgroup 1), λ_2 (constant hazard in subgroup 2), λ_3 (senescent hazard), b (senescent shape)

Constraints: $0 \leq p \leq 1$; $\lambda_1 \geq 0$, $\lambda_2 \geq 0$, $\lambda_3 \geq 0$

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.

Range: $t \geq 0$

PDF:

$$f(t) = Np \exp \left[-\lambda_1 t + \frac{\lambda_3}{b} (1 - e^{bt}) \right] (\lambda_1 + \lambda_3 e^{bt})$$

$$+ N[1-p] \exp \left[-\lambda_2 t + \frac{\lambda_3}{b} (1 - e^{bt}) \right] (\lambda_2 + \lambda_3 e^{bt}),$$

$$N = \begin{cases} 1, & b \geq 0 \\ \left(1 - e^{\frac{\lambda_3}{b}} \right)^{-1} & b < 0 \end{cases}$$

SDF:

$$S(t) = Np \exp \left[-\lambda_1 t + \frac{\lambda_3}{b} (1 - e^{bt}) \right] + N[1-p] \exp \left[-\lambda_2 t + \frac{\lambda_3}{b} (1 - e^{bt}) \right]$$

Hazard: $h(t) = p(t)\lambda_1 + [1-p(t)]\lambda_2 + \lambda_3 e^{bt}$

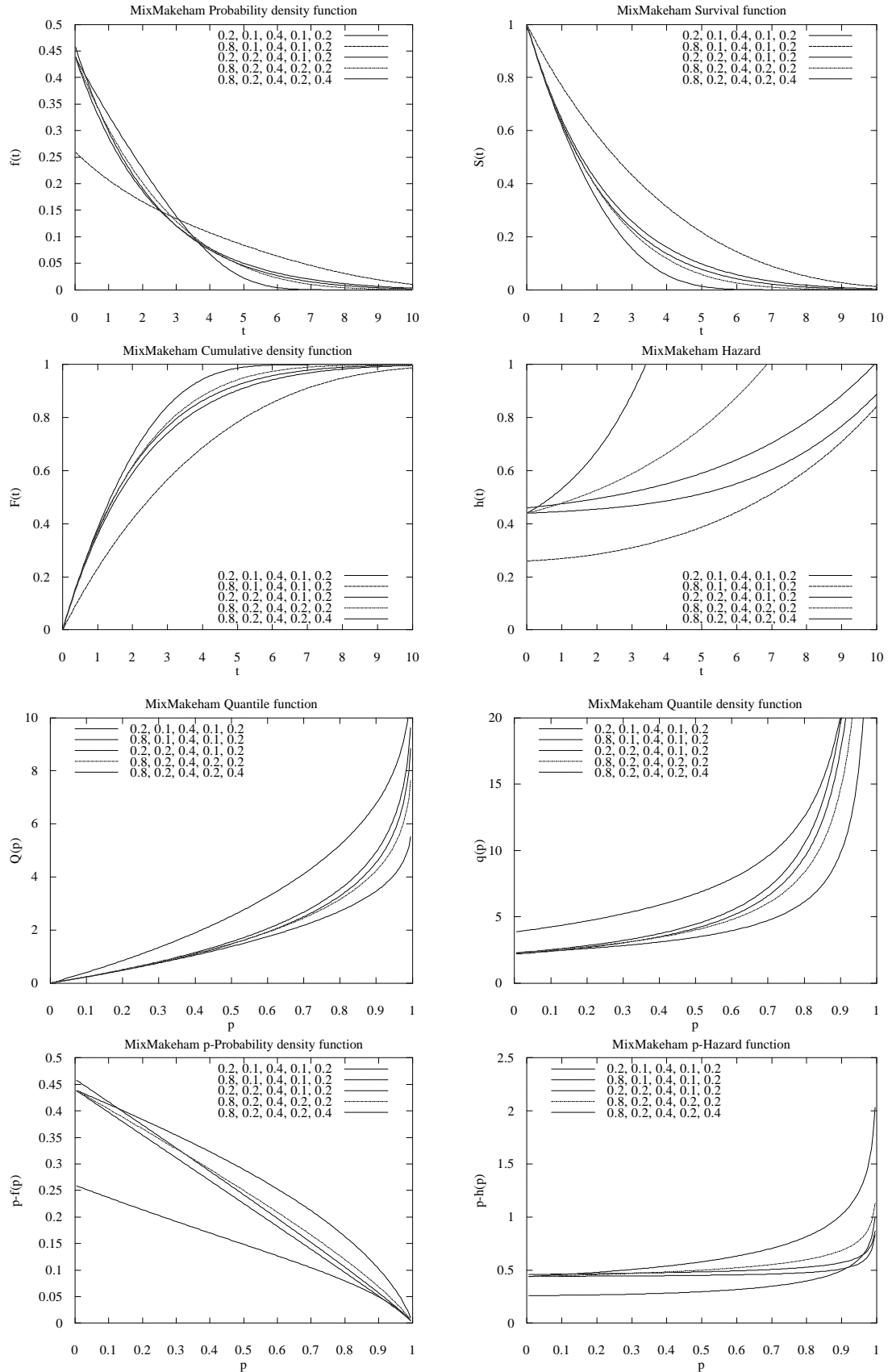
where

$$p(t) = \frac{p \exp \left[-\lambda_1 t + \frac{\lambda_3}{b} (1 - e^{bt}) \right]}{p \exp \left[-\lambda_1 t + \frac{\lambda_3}{b} (1 - e^{bt}) \right] + [1-p] \exp \left[-\lambda_2 t + \frac{\lambda_3}{b} (1 - e^{bt}) \right]}$$

Reduced models: Reduces to a Makeham-Gompertz if $p = 0$ or 1 , reduces to an 2 point hyperexponential exponential if $\lambda_3 = 0$, reduces to an exponential if $\lambda_3 = 0$ and $p = 0$ or 1 , reduces to a Gompertz if $\lambda_1 = 0$ and $\lambda_2 = 0$.

References: Wood et al. (2001)

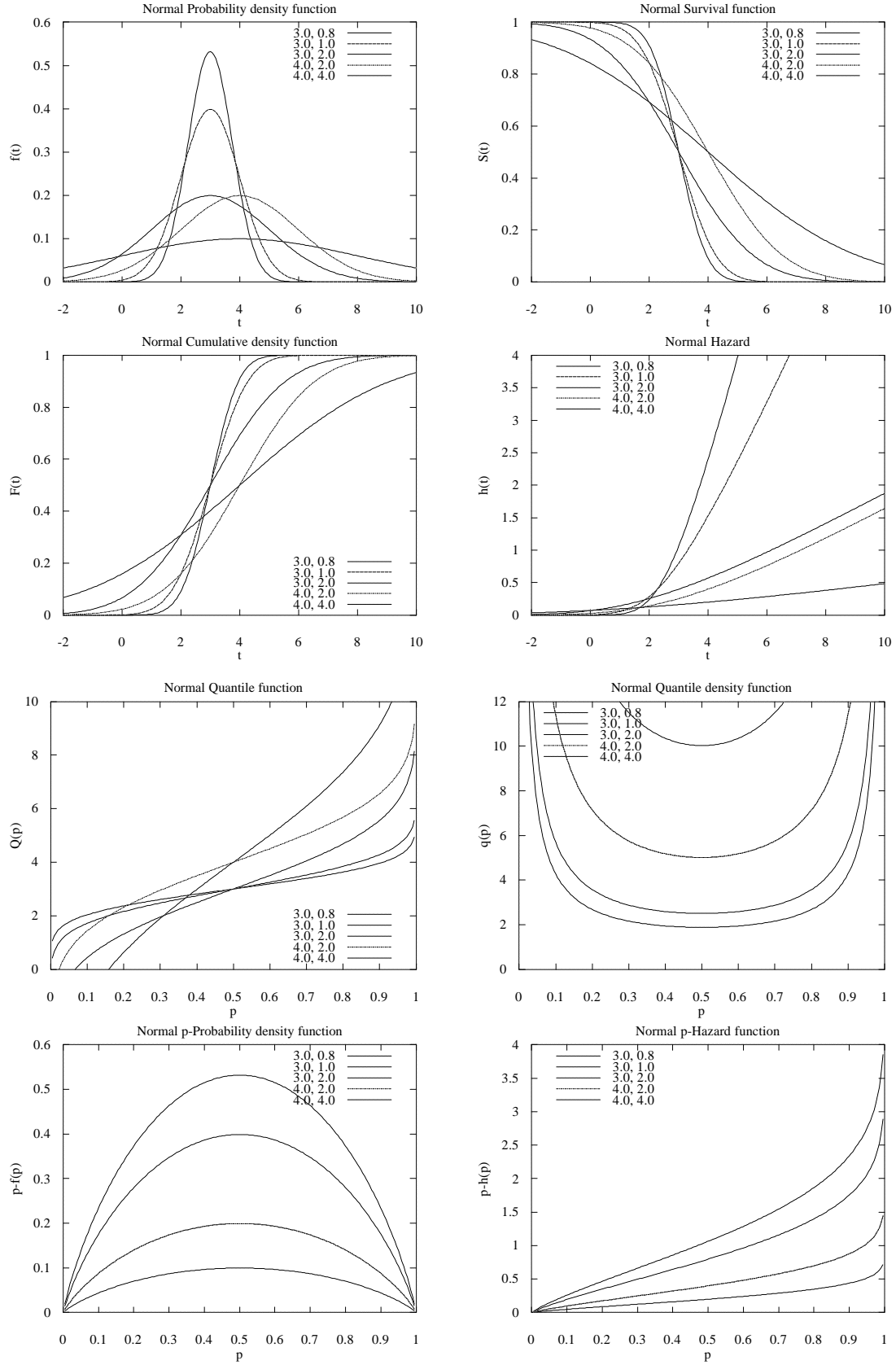
See also: MAKEHAM, GOMPERTZ, SILER, EXPONENTIAL



NORMAL or GAUSSIAN

This is the commonly-used normal distribution, also called the Gaussian distribution and Laplace's second law of error. The normal is odd (but certainly possible) as a failure time distribution because times can take any value from $-\infty$ to ∞ .

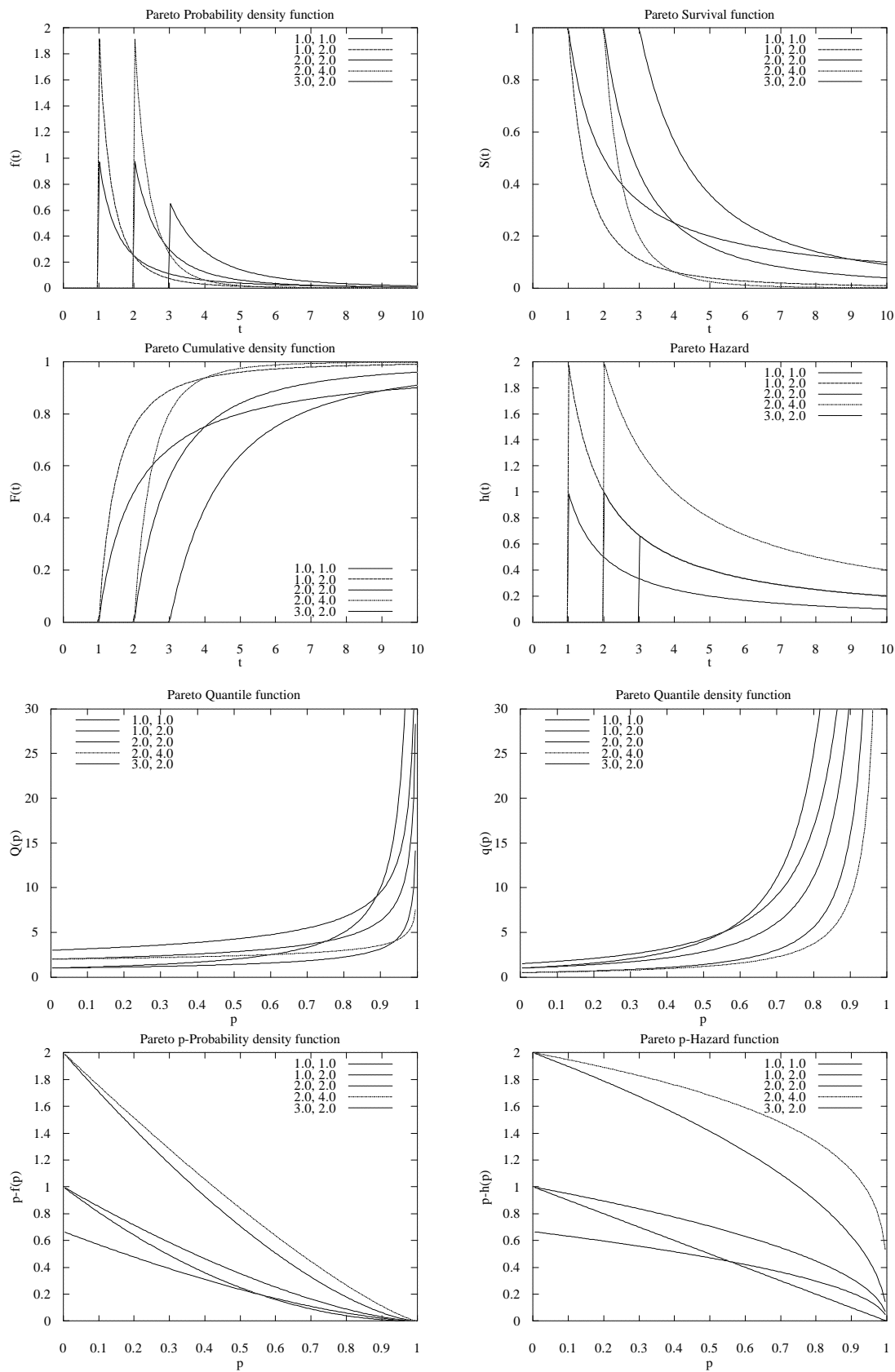
Parameters:	μ is a location parameter, σ is the scale parameter.
Constraints:	$\sigma > 0$
Time variables:	t_u, t_e, t_a, t_w . An exact failure is defined when $t_u = t_e$
Range:	$-\infty < t < \infty$
PDF:	$f(t) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(t-\mu)^2}{2\sigma^2}\right]$
SDF:	$S(t) = 1 - \Phi\left(\frac{t-\mu}{\sigma}\right),$
Mean:	μ
Median:	μ
Mode:	μ
Variance:	σ^2
Notes:	The standard (or unit) normal is defined when $\mu = 0$ and $\sigma = 1$. An accelerated failure time specification of covariates is created for the normal distribution by modeling μ as <code>FORM = LOGLIN</code> and specifying a <code>COVAR</code> list. A probit model is estimated when for every observation, either $t_u = \text{NEGINFINITY}$ or $t_e = \text{INFINITY}$.
Other names:	Gaussian, Laplace's second law of error.
References:	Johnson et al. (1994); Christensen (1984); Evans et al. (2000)
See also:	LOGNORMAL, BIVNORMAL, function ERF, function



PARETO

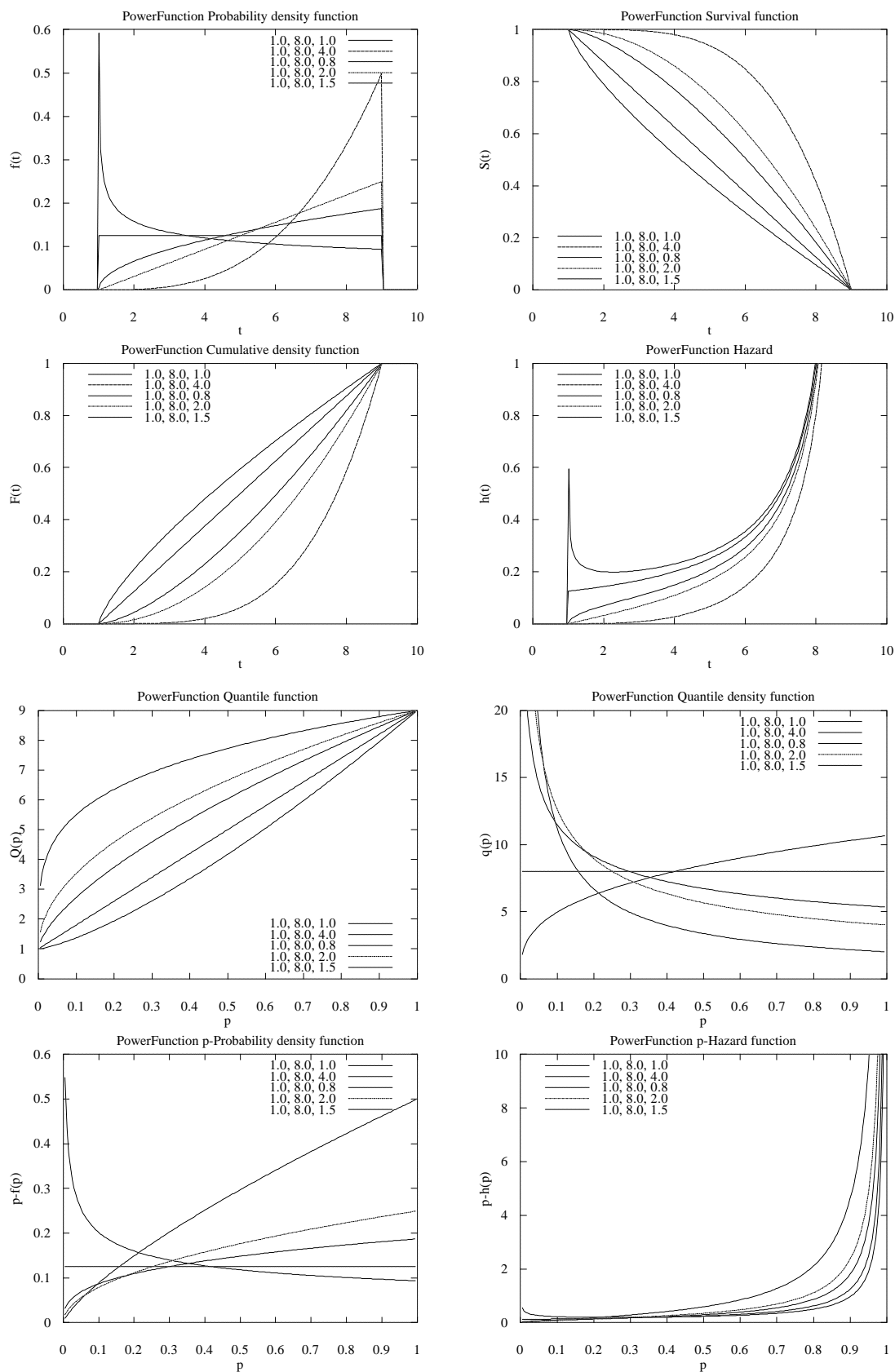
This is the Pareto distribution of the first kind.

Parameters:	b (scale), c (shape).
Constraints:	$b \geq 0, c > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$b \leq t < \infty$
PDF:	$f(t) = \frac{b^c c}{t^{c+1}}$
SDF:	$S(t) = (t/b)^{-c}$
Hazard:	c/t
Quantile:	$t_q = b(1-q)^{-1/c}$
Mean:	$bc/(c-1), c > 1$
Median:	$b\sqrt[c]{2}$
Mode:	b
Variance:	$\frac{b^2 c}{(c-2)(c-1)^2}, c > 2$
References:	Christensen (1984), Evans et al. (2000)
See also:	LOWMAX



POWERFUNCTION

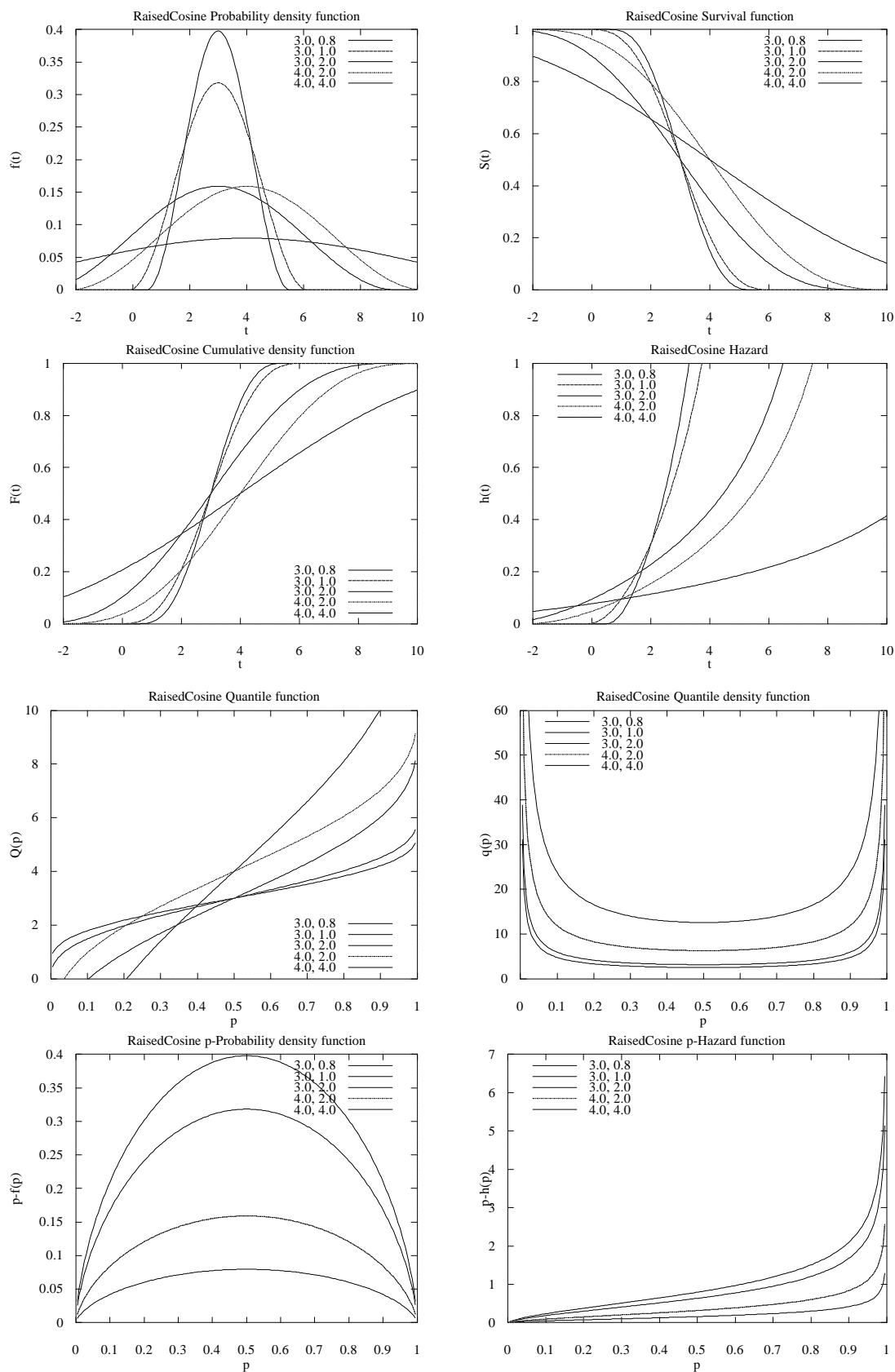
Parameters:	a (location), b (scale), c (shape)
Constraints:	$b > 0, c \geq 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$a \leq t \leq a + b$
PDF:	$f(t) = \frac{c}{b} \left(\frac{t-a}{b} \right)^{c-1}$
SDF:	$S(t) = 1 - \left(\frac{t-a}{b} \right)^c$
Quantile:	$t_q = a + bq^{1/c}$
Mean:	$a + bc/(c+1)$
Median:	$a + b2^{-1/c}$
Mode:	$\begin{cases} a+b, & c \geq 1 \\ a+b/2, & c = 1 \\ a, & c < 1 \end{cases}$
Variance:	$\frac{b^2 c}{(c+2)(c+1)^2}$
Reduced models:	Reduces to the uniform distribution when $c = 1$.
References:	Christensen (1984), Evans et al. (2000)
See also:	REVPOWERFUNCTION, LOGISTIC, WEIBULL, GUMBEL, BETA, PARETO



RAISEDCOSINE

This is the raised cosine distribution.

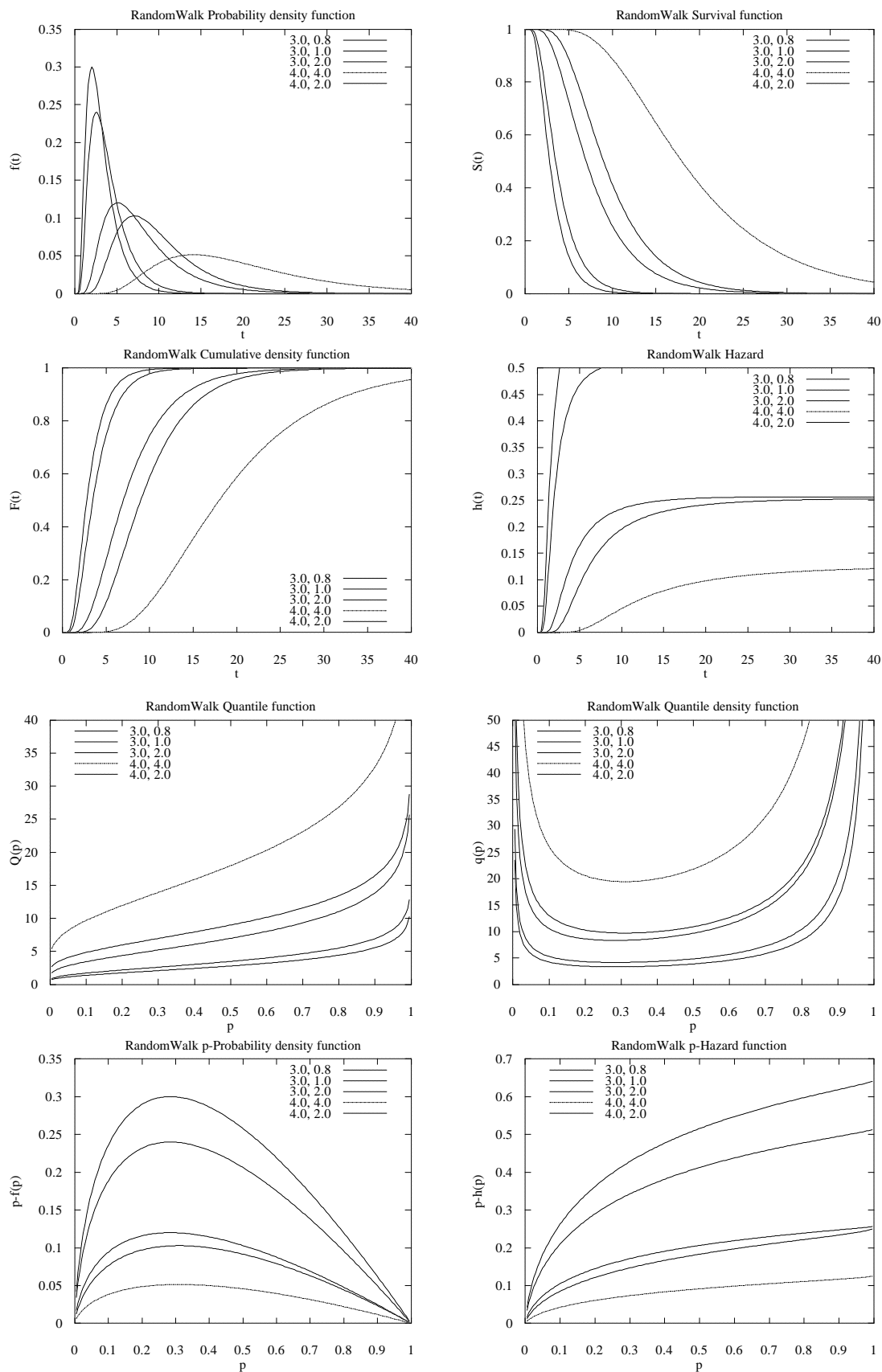
Parameters:	a (location), b (scale)
Constraints:	$b \geq 0$
Time variables:	t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$-\pi b + a \leq t \leq \pi b + a$
PDF:	$f(t) = \frac{1 + \cos\left(\frac{t-a}{b}\right)}{2\pi b}$
SDF:	$S(t) = \frac{1}{2} - \frac{1}{\pi} \left(\frac{t-a}{b}\right) - \frac{1}{\pi} \sin\left(\frac{t-a}{b}\right)$
Mean:	a
Median:	a
Mode:	a
Variance:	$b^2(\pi^2/3 - 2)$
References:	Chew (1968), Christensen (1984)



RANDOMWALK

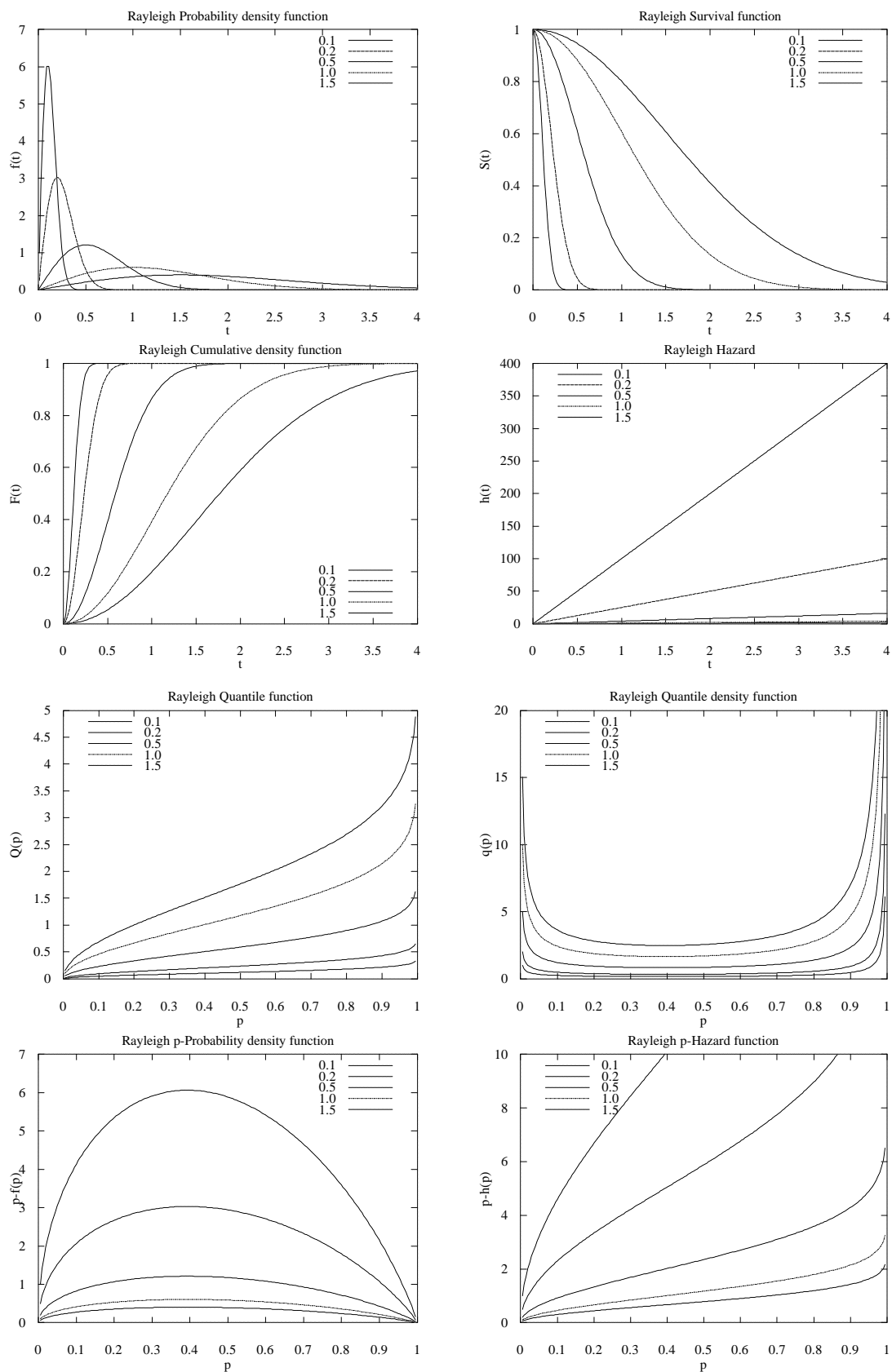
This is the random walk distribution.

Parameters:	b (scale), c (shape)
Constraints:	$b > 0, c > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t > 0$
PDF:	$f(t) = \frac{1}{\sqrt{2\pi bt}} \exp \left[-\frac{c^2 t}{2b} \left(\frac{b}{t} - \frac{1}{c} \right)^2 \right]$
SDF:	$S(t) = 1 - \Phi \left(\frac{bc - t}{\sqrt{bt}} \right) - e^{2c} \Phi \left(\frac{-bc - t}{\sqrt{bt}} \right)$
Mean:	$b(1 + c)$
Mode:	$b\sqrt{c^2 + 1/4} - b/2$
Variance:	$b^2(2 + c)$
References:	Christensen (1984), Wise (1966)
See also:	INVGAUSSIAN
Bugs:	The hazard function sometimes returns bad values (see graphs).



RAYLEIGH

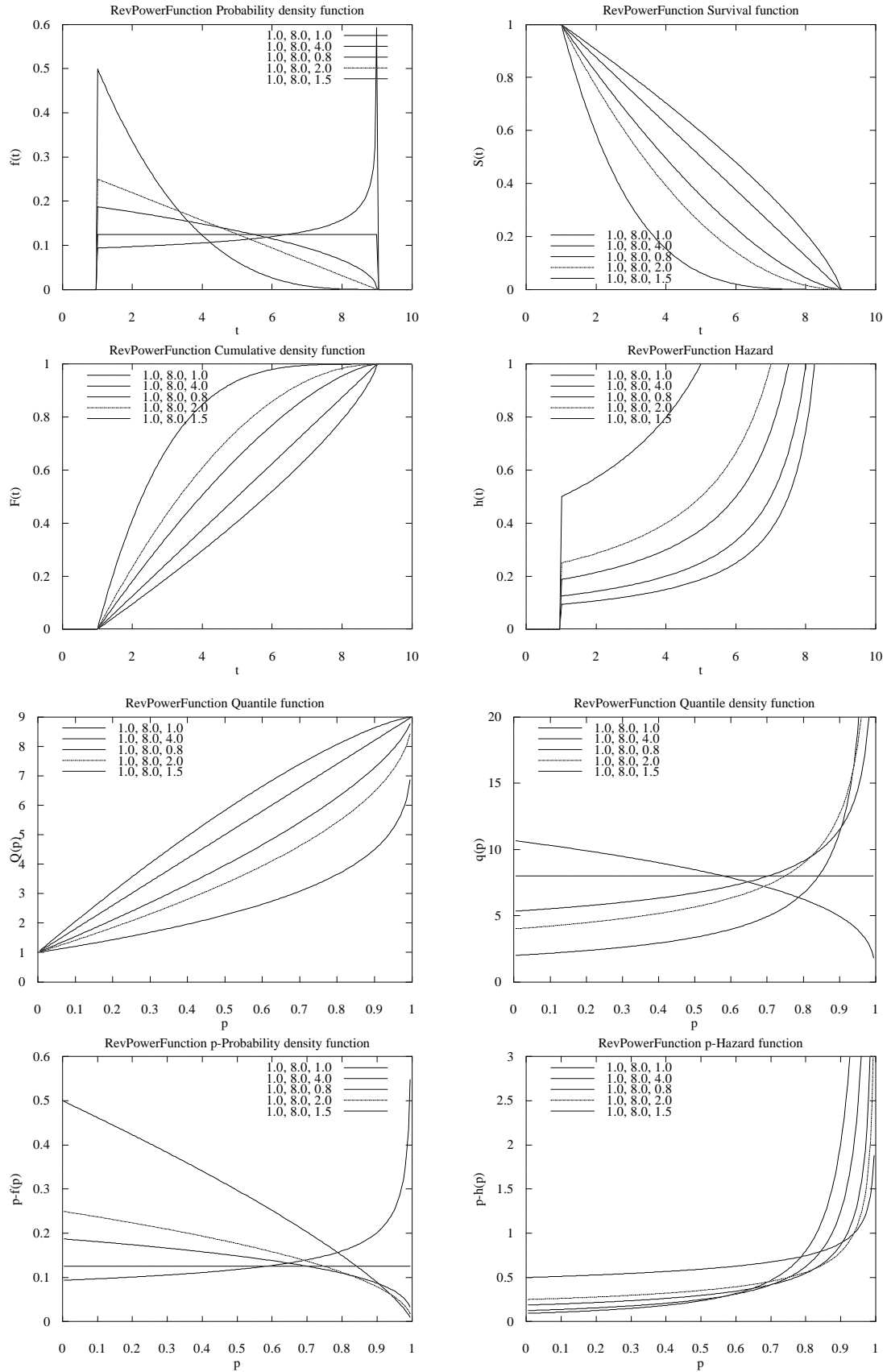
Parameter:	b (scale)
Constraints:	$b > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t > 0$
PDF:	$f(t) = \frac{t}{b^2} \exp\left[-\frac{t^2}{2b^2}\right]$
SDF:	$S(t) = \exp\left[-\frac{t^2}{2b^2}\right]$
Hazard:	$h(t) = t/b^2$
Median:	$b\sqrt{\log(4)}$
Mode:	b
References:	

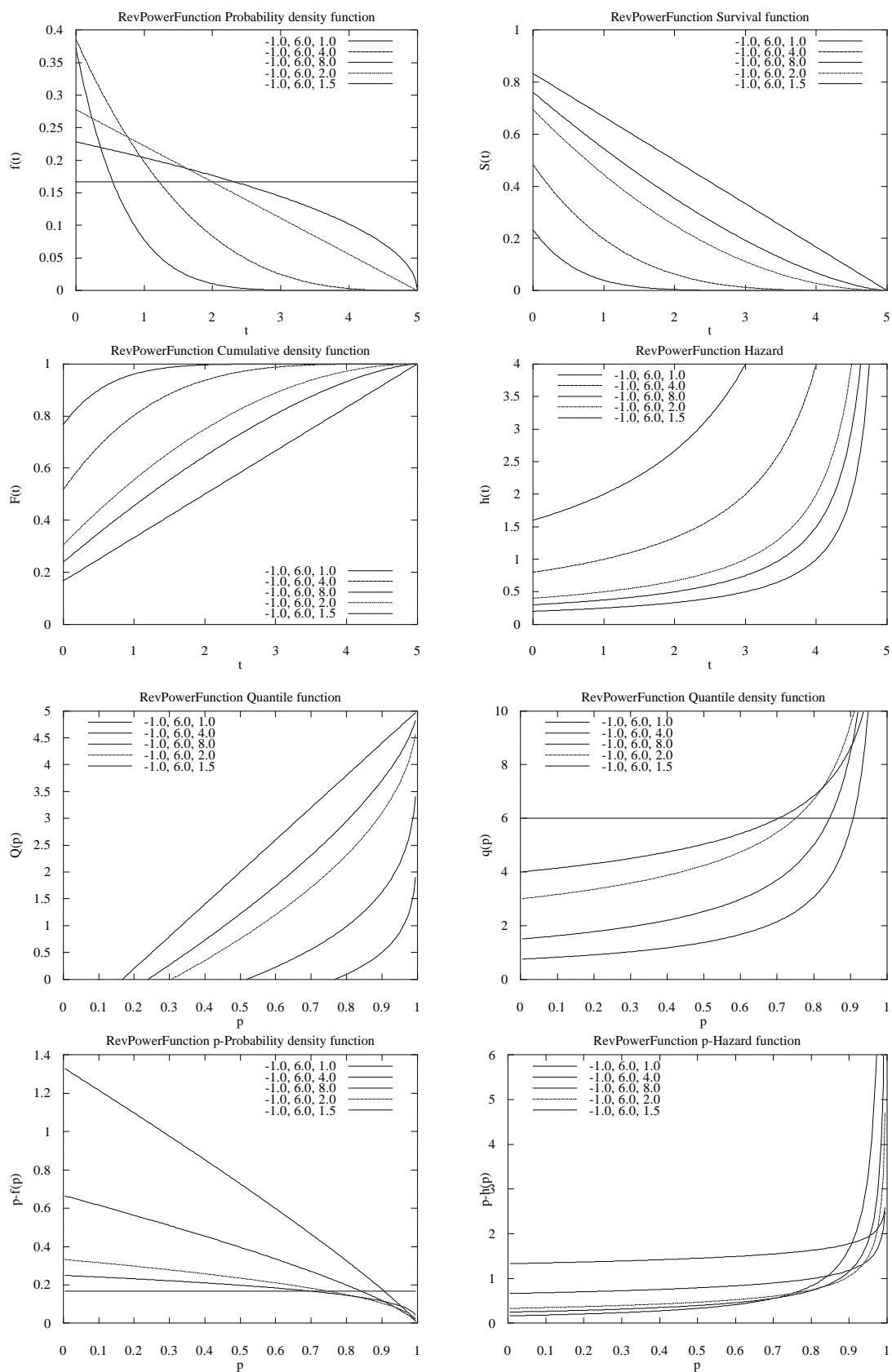


REVPOWERFUNCTION

This is the reversed power distribution.

Parameters:	a (location), b (scale), c (shape)
Constraints:	$b > 0, c \geq 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$a \leq t \leq a + b$
PDF:	$f(t) = \frac{c}{b} \left(\frac{a+b-t}{b} \right)^{c-1}$
SDF:	$S(t) = \left(1 - \frac{t-a}{b} \right)^c$
Quantile:	$t_q = a + b[1-(1-q)^{1/c}]$
Mean:	$a + b/(c+1)$
Median:	$a + b(1-2^{-1/c})$
Mode:	$\begin{cases} a, & c > 1 \\ a + b/2, & c = 1 \\ a + b, & c < 1 \end{cases}$
Variance:	$\frac{b^2 c}{(c+2)(c+1)^2}$
Reduced models:	Reduces to the uniform distribution when $c = 1$.
References:	Christensen (1984).
See also:	POWERFUNCTION





RINGINGEXP0

This is the ringing exponential distribution at phase 0 degrees.

Parameters: a (location), b (scale), c (shape)

Constraints: $b > 0, c \geq 0$

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.

Range: $t \geq a$

PDF:
$$f(t) = \frac{1+2c}{b(1+c)} e^{-\frac{t-a}{b}} \cos^2\left(\frac{t-a}{b} \sqrt{\frac{c}{2}}\right)$$

SDF:
$$S(t) = \frac{e^{-\frac{t-a}{b}}}{1+c} \left[c + \cos^2\left(\frac{t-a}{b} \sqrt{\frac{c}{2}}\right) - \sqrt{\frac{c}{2}} \sin\left(\frac{t-a}{b} \sqrt{2c}\right) \right]$$

Hazard:
$$h(t) = \frac{(1+2c) \cos^2\left(\frac{(a-t)\sqrt{2c}}{2b}\right)}{b \left[2c + 2 \cos^2\left(\frac{(a-t)\sqrt{2c}}{2b}\right) + \sqrt{2c} \sin\left(\frac{(a-t)\sqrt{2c}}{b}\right) \right]}$$

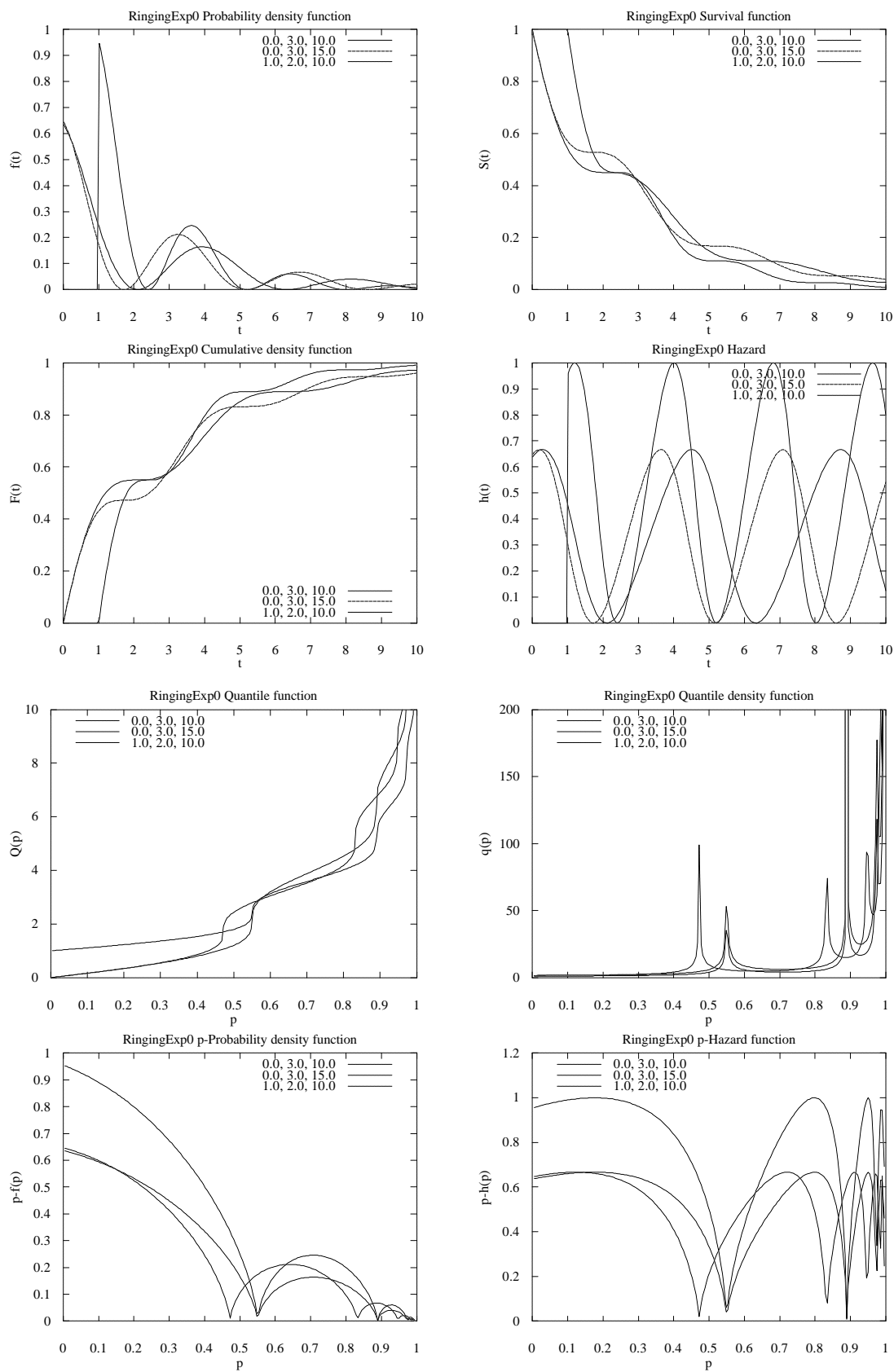
Mean:
$$a + \frac{b(1+c+2c^2)}{1+3c+2c^2}$$

Mode: a

Variance:
$$\frac{b^2(1+7c^2+16c^3+4c^4)}{(1+3c+2c^2)^2}$$

References: Christensen (1984)

See also: RINGINGEXP180



RINGINGEXP180

This is the ringing exponential distribution at phase 180 degrees.

Parameters: a (location), b (scale), c (shape)

Constraints: $b > 0, c > 0$

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.

Range: $t \geq a$

PDF:
$$f(t) = \frac{2c+1}{bc} e^{-\frac{t-a}{b}} \sin^2 \left(\frac{t-a}{b} \sqrt{\frac{c}{2}} \right)$$

SDF:
$$S(t) = \frac{e^{-\frac{t-a}{b}}}{c} \left[c + \sin^2 \left(\frac{t-a}{b} \sqrt{\frac{c}{2}} \right) + \sqrt{\frac{c}{2}} \sin \left(\frac{t-a}{b} \sqrt{2c} \right) \right]$$

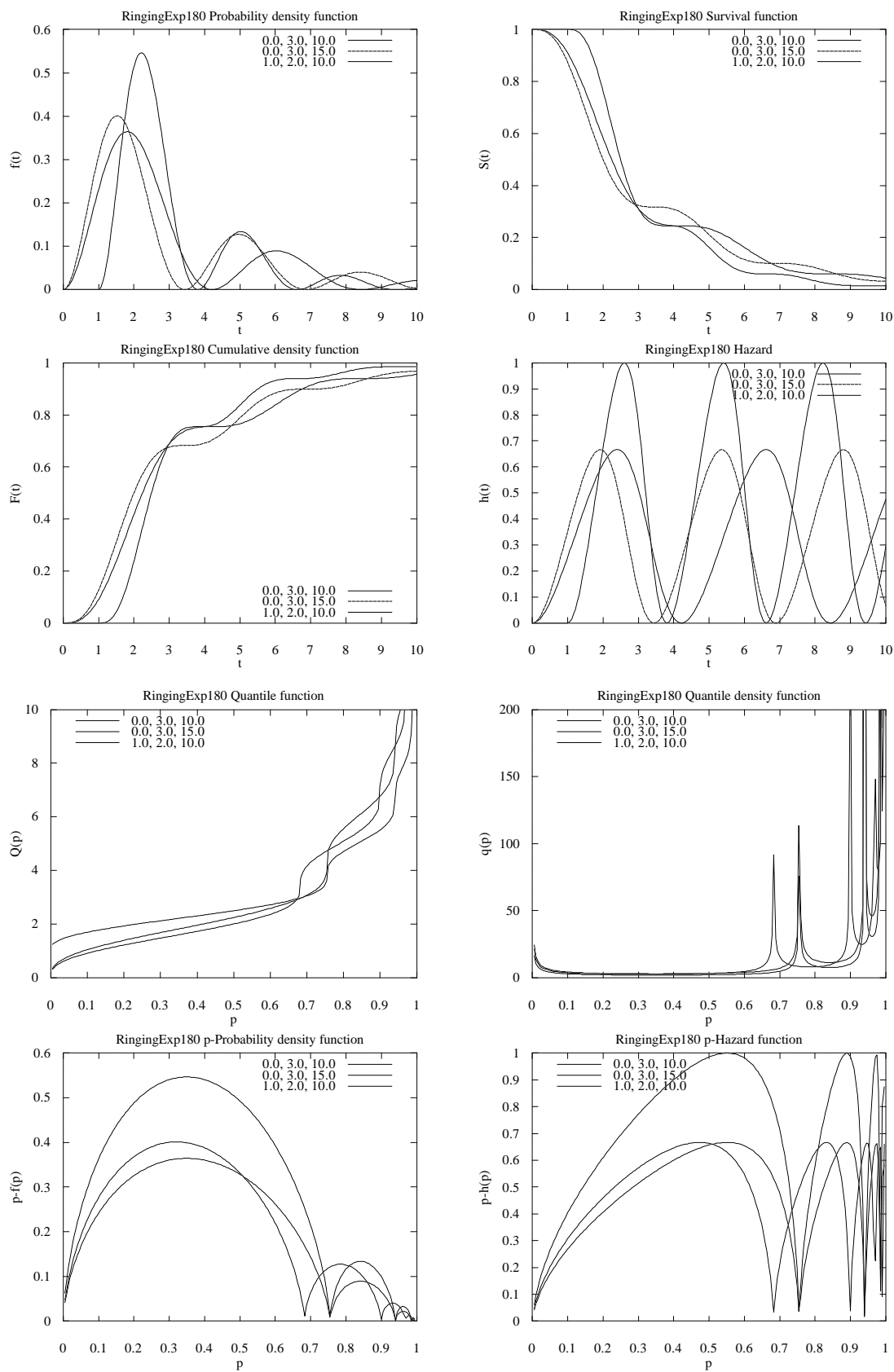
Mean:
$$a + \frac{b(3+2c)}{1+2c}$$

Mode:
$$a + \frac{2b \arctan(\sqrt{2c})}{\sqrt{2c}}$$

Variance:
$$\frac{b^2(3+4c^2)}{(1+2c)^2}$$

References: Christensen (1984)

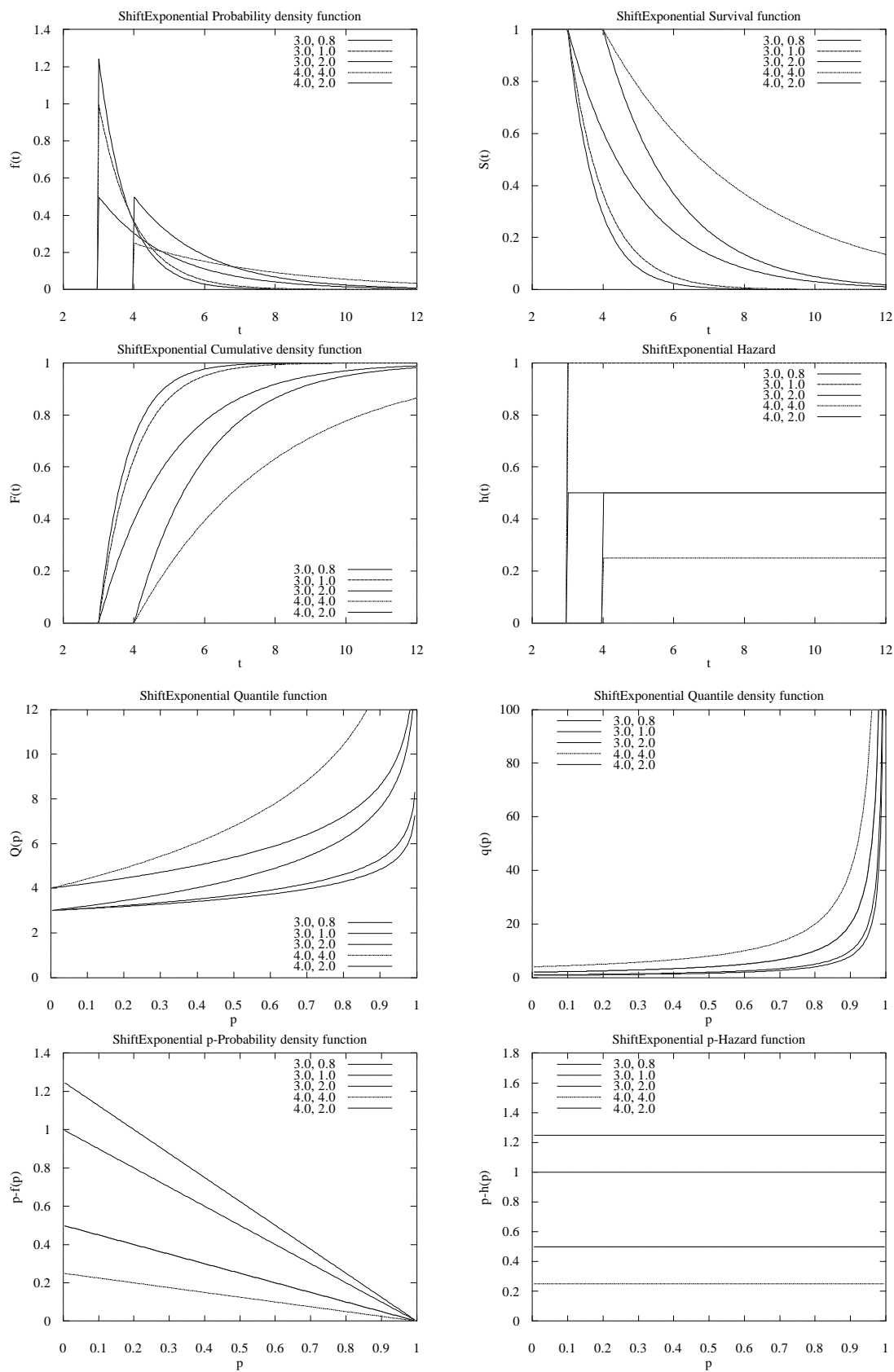
See also: RINGINGEXP0



SHIFTEXPONENTIAL

The exponential is commonly used in reliability engineering, queuing theory and biology. The 'memoryless' property of the exponential distribution is an important characteristic. It says, in effect, that for a survivor, future times to failure are completely independent of the past. This form of the exponential distribution provides for a location parameter. Also, this version uses a scale parameter b , rather than a hazard parameter λ ($b = 1/\lambda$).

Parameter:	a (location), b (scale = $1/\lambda$).
Constraints:	$b > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$
Range:	$t \geq a$
PDF:	$f(t) = \exp(-(t - a)/b)/b$
SDF:	$S(t) = \exp(-(t - a)/b)$
Hazard:	$h(t) = 1/b$
Quantile:	$t_q = a - b \ln(1 - q)$
Mean:	$a + b$
Median:	$a + b \ln(2)$
Mode:	a
Variance:	b^2
References:	Christensen (1984), Evans et al. (2000), Nelson (1982)
See also:	The EXPONENTIAL distribution is a 1 parameter (hazard) version of this distribution.



SHIFTGAMMA

This is the three parameter (shifted) gamma distribution, also known as the Pearson Type III distribution.

Parameters: a (location), b (scale, inverse of the hazard), c (shape)

Constraints: $b \geq 0, c \geq 0$

Time variables: t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.

Range: $t \geq a$

PDF:
$$f(t) = \frac{\left(\frac{t-a}{b}\right)^{c-1} e^{-\frac{t-a}{b}}}{b\Gamma(c)}$$

SDF:
$$S(t) = \frac{\Gamma\left(c, \frac{t-a}{b}\right)}{\Gamma(c)}$$

Hazard:
$$h(t) = \frac{\left(\frac{t-a}{b}\right)^{c-1} e^{-\frac{t-a}{b}}}{b\Gamma\left(c, \frac{t-a}{b}\right)}$$

Quantile:
$$t_q = a + \frac{b}{2} \chi_q^2(2c)$$

Mean: $a + bc$

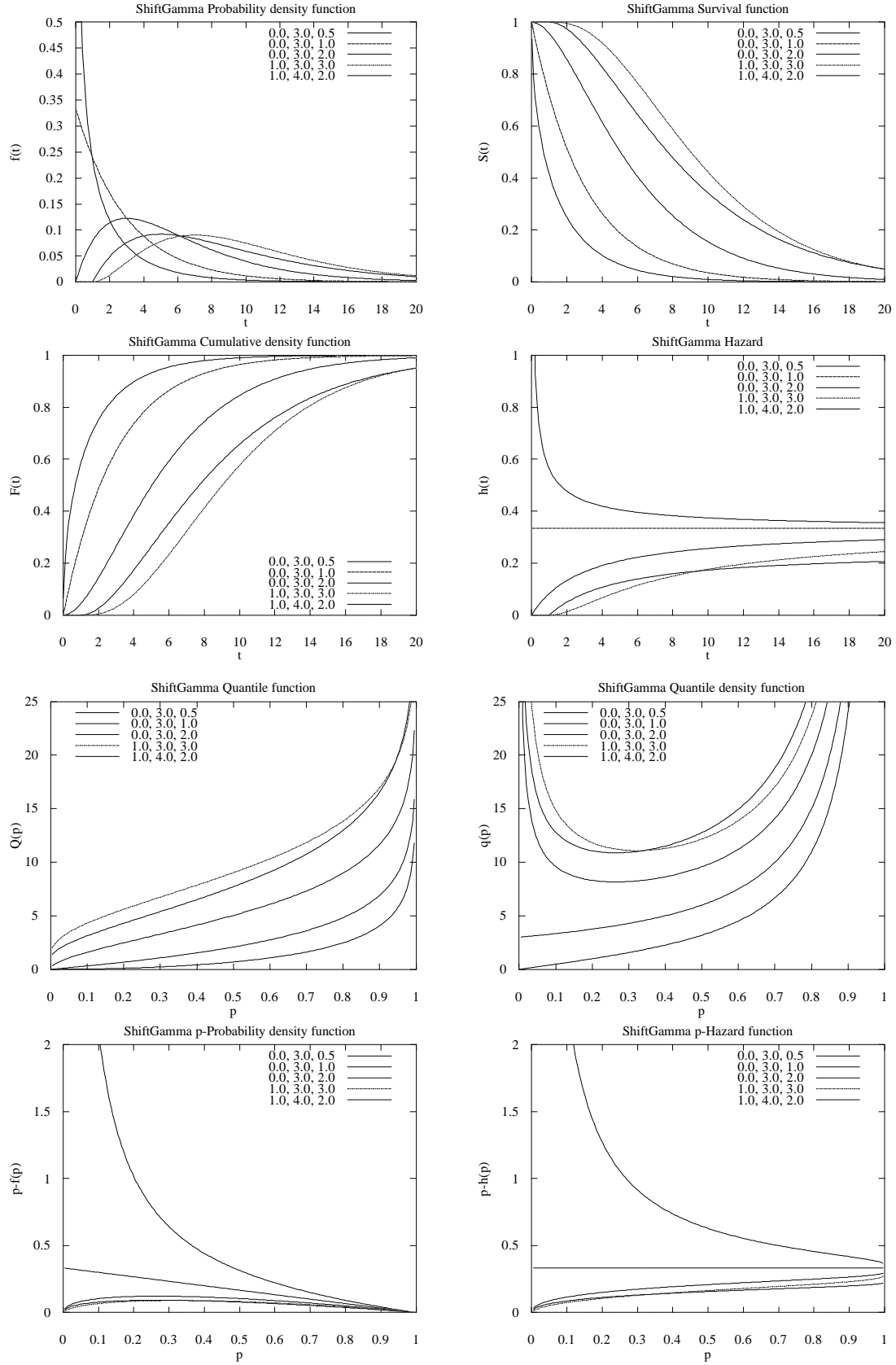
Mode: $a + b(c-1), \quad c > 1$
 $a, \quad c \leq 1$

Variance: b^2c

Reduced models: Reduces to a shifted exponential distribution when $c = 1$. Reduces to an Erlang distribution with integer parameter c . Reduces to a Chi-squared distribution with v degrees of freedom with $a = 2, b = v/2$. Reduces to a gamma distribution with $a = 0$ and $b = 1/\lambda$.

References: Christensen (1984), Elandt-Johnson and Johnson (1980), Evans et al. (2000), Kalbfleisch and Prentice (1980).

See also: SHIFTEXPONENTIAL, GAMMA, GENGAMMA, CHISQUARED



SHIFTLOGNORMAL

This is a three-parameter shifted lognormal distribution, which is a reparameterization of the LOGNORMAL distribution.

Parameters: a (location), b (scale), c (shape).

Constraints: $b > 0, c > 0$

Time variables: t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$

Range: $t \geq a$

PDF:
$$f(t) = \frac{1}{(t-a)c\sqrt{2\pi}} e^{\left[-\frac{1}{2c^2} \ln\left(\frac{t-a}{b}\right)^2\right]}$$

SDF:
$$S(t) = 1 - \Phi\left[\frac{1}{c} \ln\left(\frac{t-a}{b}\right)\right]$$

Mean: $a + b\exp(c^2/2)$

Median: $a + b$

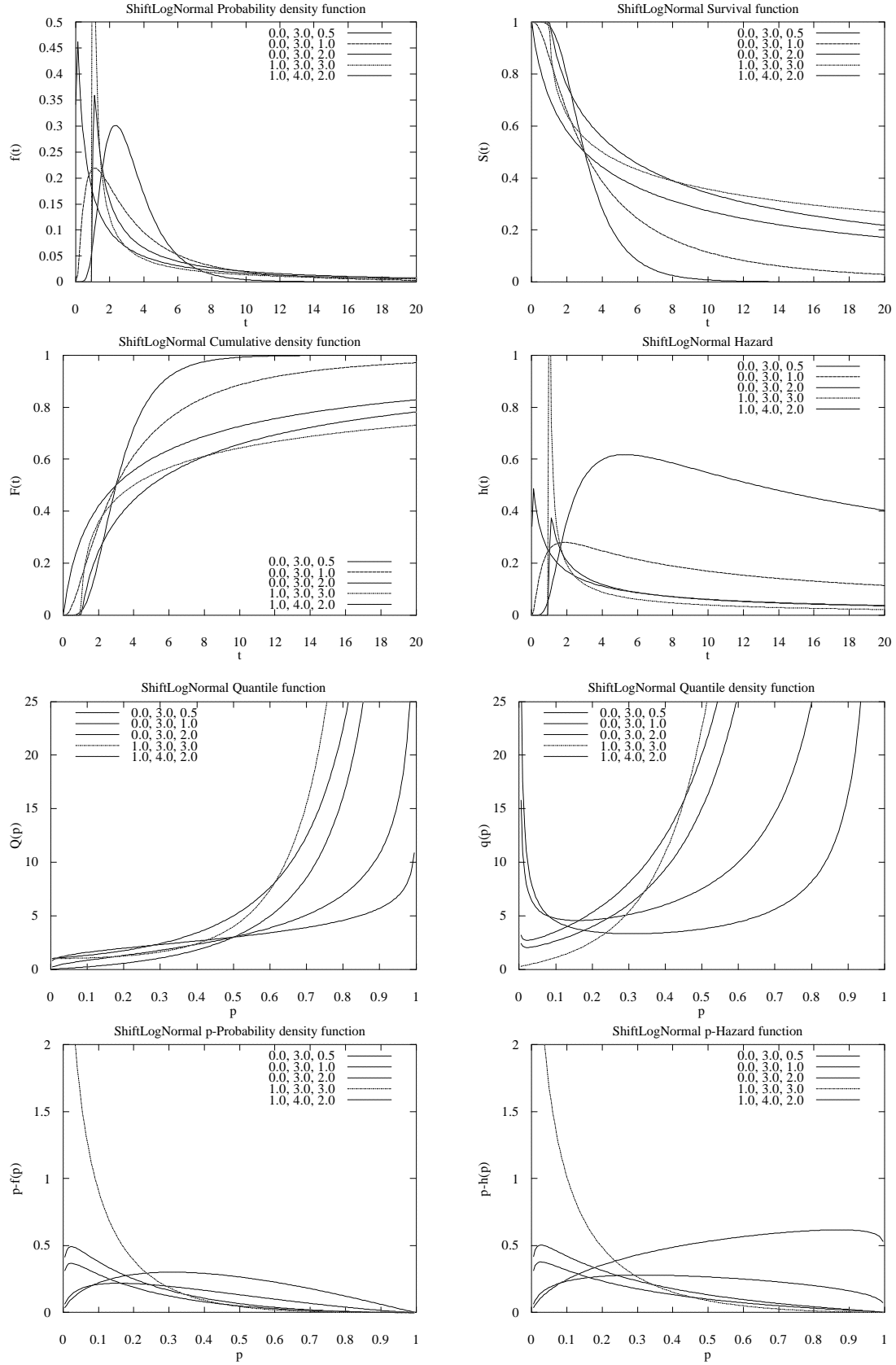
Mode: $a + b\exp(c^2/2)$

Variance: $b^2\exp(c^2)[\exp(c^2) + 2]$

Notes: This distribution is related to the SHIFTLOGNORMAL distribution as follows. The two-parameter LNNORMAL cumulative density is found from the Normal density by taking $\Phi\{[\ln(t) - a]/b\}$ whereas in the three parameter SHIFTLOGNORMAL we take $\Phi\{\ln[(t - a)/b]/c\}$.

References: Johnson et al. (1994); Nelson (1982)

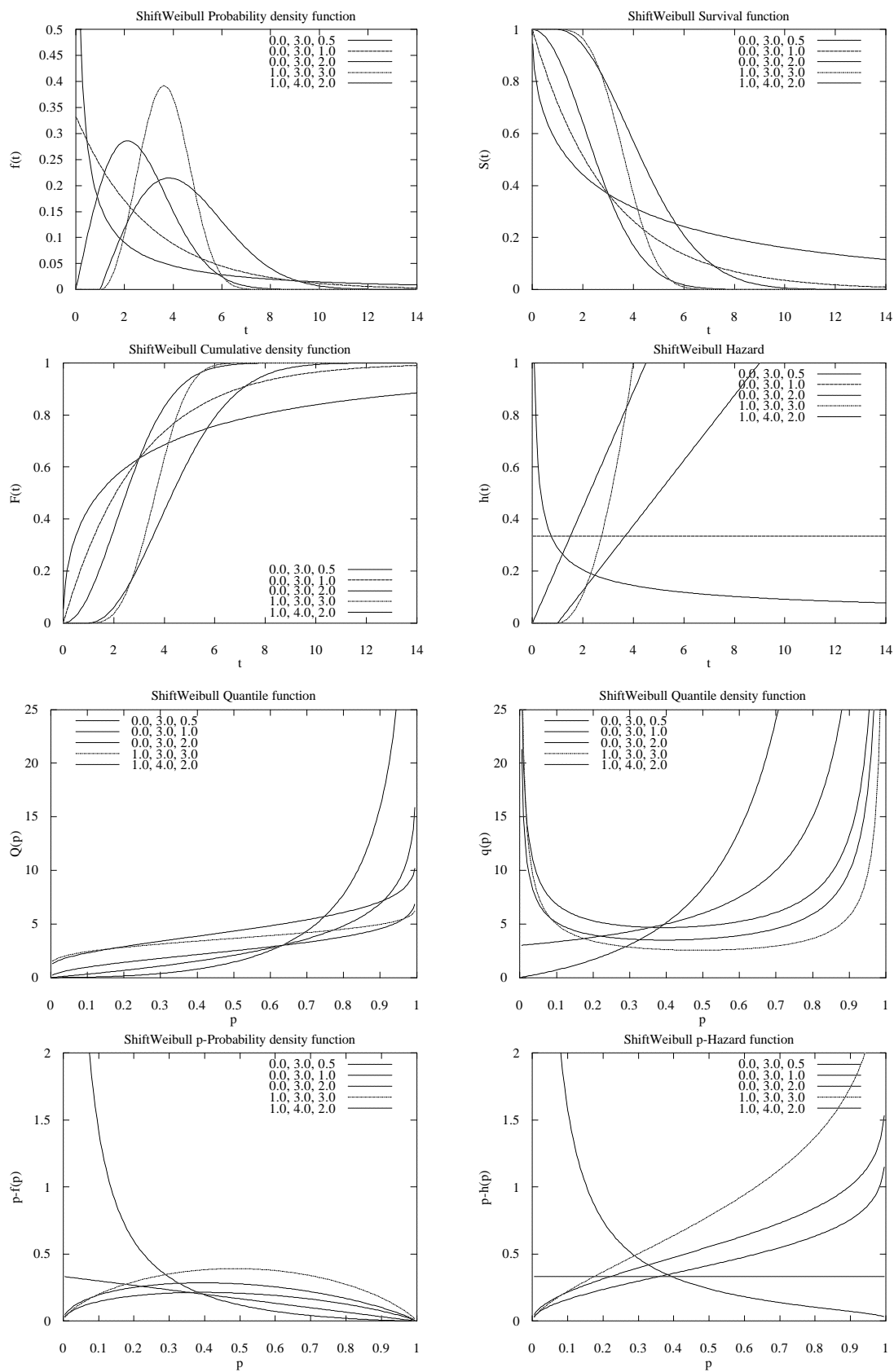
See also: NORMAL, LOGNORMAL



SHIFTWEIBULL

This is the three-parameter Shifted Weibull distribution.

Parameters:	a (location) b (scale and characteristic life= 63rd percentile), c (shape).
Constraints:	$b > 0$, $c > 0$
Time variables:	t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq a$
PDF:	$f(t) = \frac{c}{b} \left(\frac{t-a}{b} \right)^{c-1} \exp \left[- \left(\frac{t-a}{b} \right)^c \right]$
SDF:	$S(t) = \exp \left[- \left(\frac{t-a}{b} \right)^c \right]$
Hazard:	$h(t) = ct^{c-1}b^{-c}$
Mean:	$a + b\Gamma(1 + 1/c)$
Median:	$a + b\sqrt[1/c]{\ln(2)}$
Mode:	$\begin{cases} a + b\sqrt[1/c]{1-1/c}, & c > 1 \\ a, & c \leq 1 \end{cases}$
Variance:	$b^2 \left\{ \Gamma(1 + 2/c) - [\Gamma(1 + 1/c)]^2 \right\}$
Reduced models:	Reduces to the exponential distribution when $c = 1$, reduces to the Rayleigh distribution when $c = 2$.
References:	Christensen (1983); Nelson (1982)
See also:	WEIBULL



SILER

This is the Siler distribution frequently used as a competing hazards model for mortality (see Gage 1989).

Parameters: a_1, b_1 ("infant mortality" component), a_2 (constant), a_3, b_3 (senescent mortality component).

Constraints: $a_1 \geq 0, a_2 \geq 0, a_3 \geq 0, b_1 \geq 0, b_3 \geq 0$

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.

Range: $t \geq 0$

PDF:

$$f(t) = (a_1 e^{-b_1 t} + a_2 + a_3 e^{b_3 t}) \exp \left[-\frac{a_1}{b_1} (1 - e^{-b_1 t}) - a_2 t + \frac{a_3}{b_3} (1 - e^{b_3 t}) \right],$$

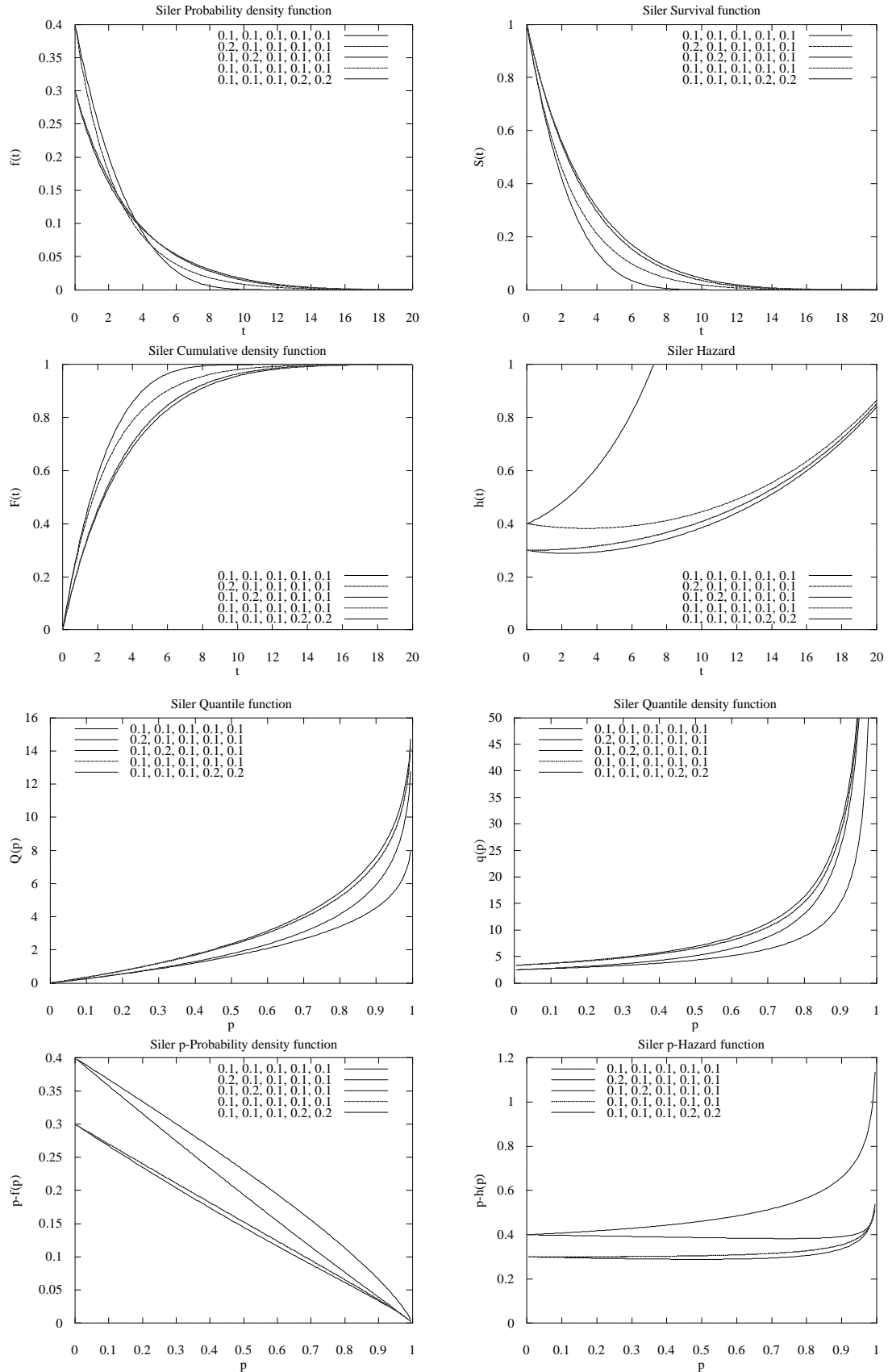
SDF:
$$S(t) = \exp \left[-\frac{a_1}{b_1} (1 - e^{-b_1 t}) - a_2 t + \frac{a_3}{b_3} (1 - e^{b_3 t}) \right]$$

Hazard: $h(t) = a_1 \exp(-b_1 t) + a_2 + a_3 \exp(b_3 t)$

Reduced models: $a_1 = 0$ reduces to a Gompertz-Makeham. $a_1 = 0, a_2 = 0$ reduces to a Gompertz. Reduces to an exponential in a number of ways.

References: Wood et al. (2001); Gage (1989)

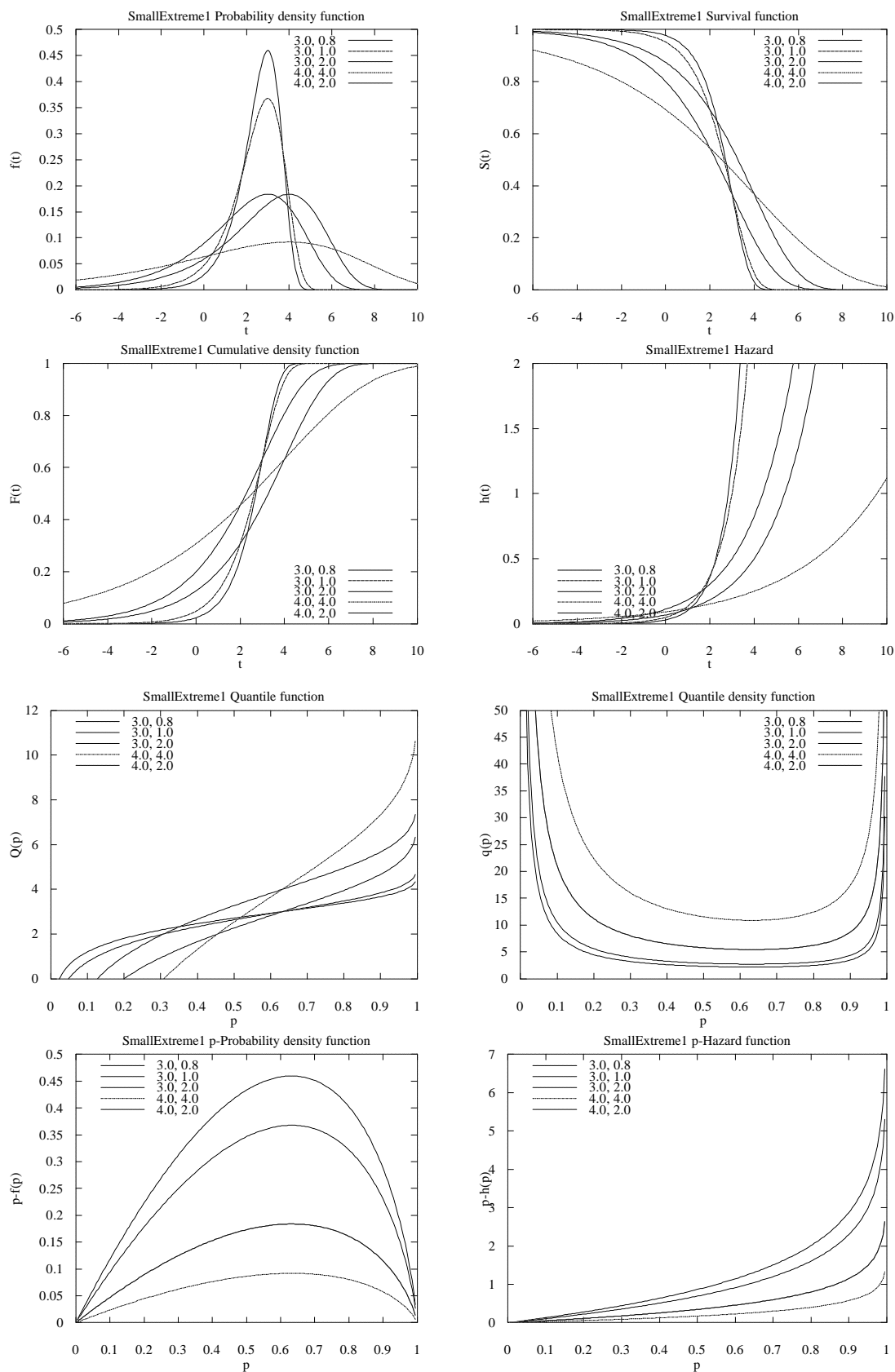
See also: EXPONENTIAL, GOMPERTZ, MAKEHAM, MIXMAKEHAM



SMALLEXTREME1

This is the type 1 smallest extreme value distribution. This is also one type of Gumbel distribution

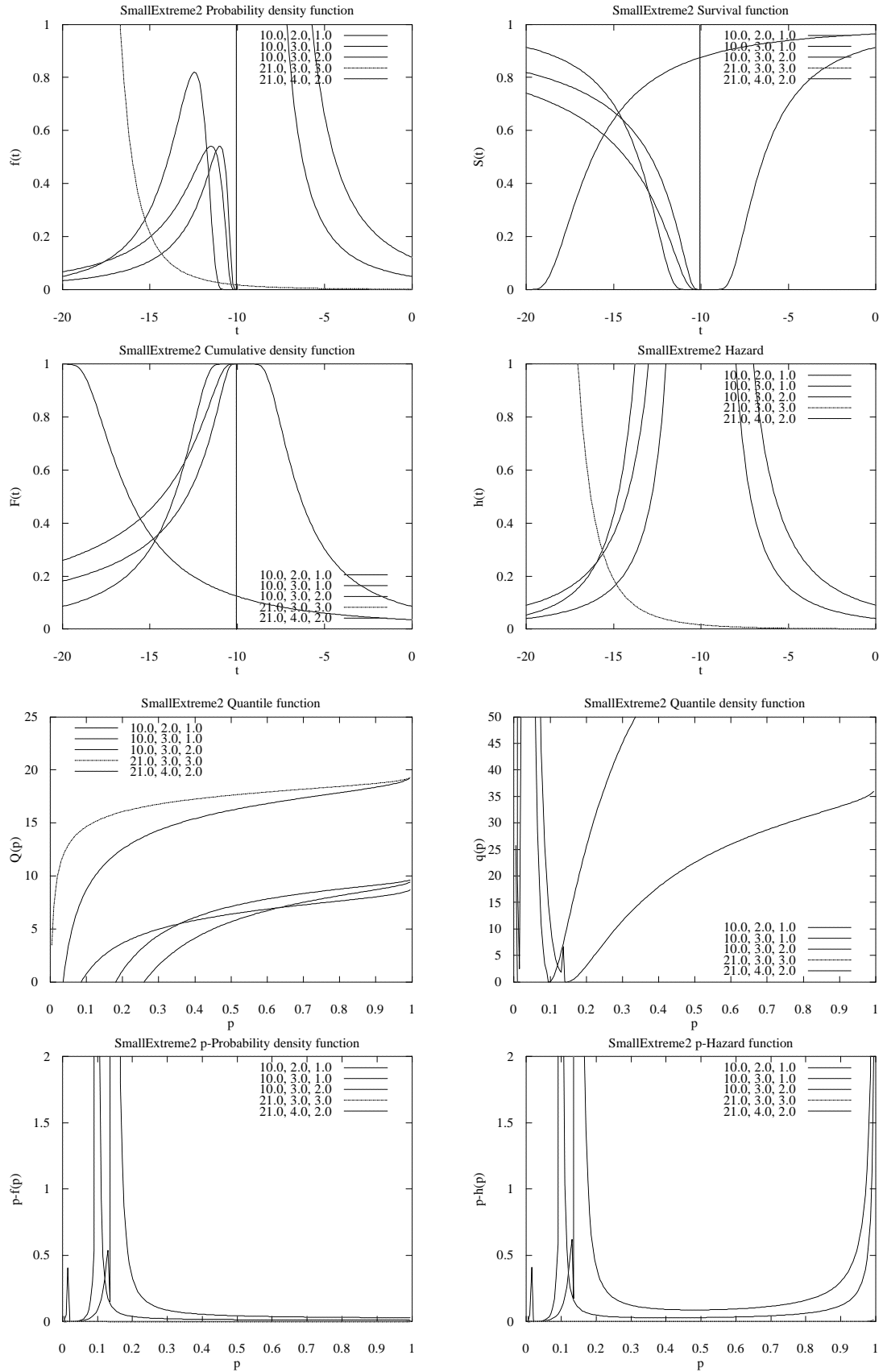
Parameters:	a (location) and b (scale)
Constraints:	$b \geq 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$-\infty < t < \infty$
PDF:	$f(t) = b^{-1} \exp \left[\frac{t-a}{b} - \exp \left(\frac{t-a}{b} \right) \right]$
SDF:	$S(t) = \exp \left\{ -\exp \left(\frac{t-a}{b} \right) \right\}$
Hazard:	$h(t) = b^{-1} \exp \left[\frac{t-a}{b} \right]$
Mean:	$a - kb$ $k=0.57721\dots$ is Euler's constant
Median:	$a - b \log[\log(2)]$
Mode:	a (63.2nd percentile)
Variance:	$b^2 \pi^2/6$
References:	Evans et al. (2000), Johnson et al. (1994), Nelson (1982)
See also:	LARGEEXTREME1, GUMBEL, SMALLEXTREME2, WEIBULL



SMALLEXTREME2

This is the type 2 smallest extreme value distribution, also known as the Frechet distribution.

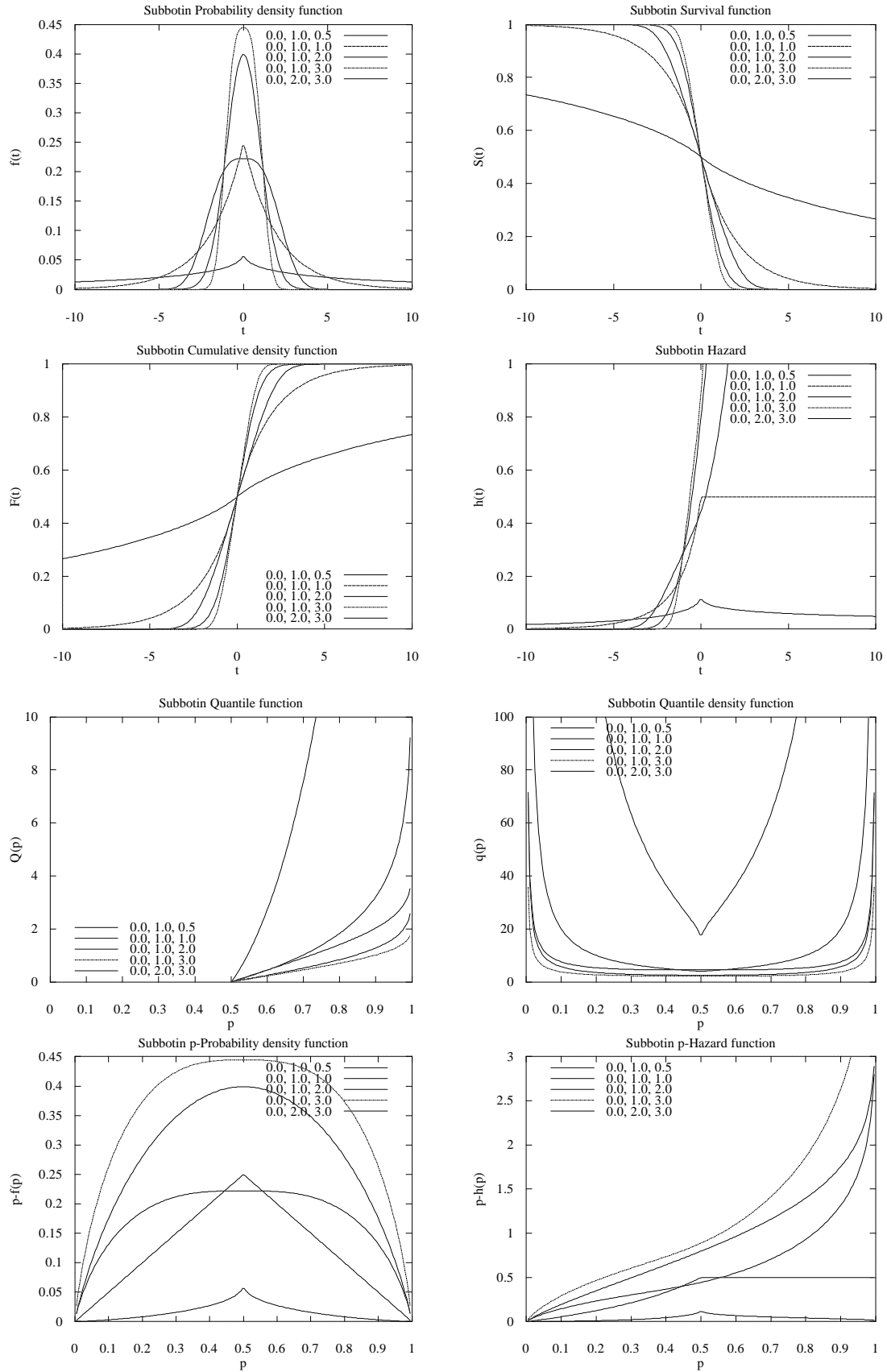
Parameters:	a (location), b (scale), c (shape)
Constraints:	$b \geq 0, c \geq 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$-\infty \leq t < a$
PDF:	$f(t) = c \left(\frac{-t-a}{b} \right)^{(-c-1)} \exp \left[- \left(\frac{-t-a}{b} \right)^{-c} \right]$
SDF:	$S(t) = \exp \left[- \left(\frac{-t-a}{b} \right)^{-c} \right]$
Hazard:	$f(t) = c \left(\frac{-t-a}{b} \right)^{(-c-1)}$
Quantile:	$t_q = a - b[-\ln(1-q)]^{-1/c}$
Median:	$a + b[\ln(2)]^{-1/c}$
References:	Evans et al. (2000), Johnson et al. (1994), Nelson (1982)
See also:	LARGEEXTREME1, SMALLEXTREME2
Bugs:	This distribution does not work correctly.



SUBBOTIN

This is the Subbotin distribution, which includes a number of important distributions as special cases, including the uniform, Laplace, and normal distributions. The distribution becomes discontinuous at the median when $c < 1$.

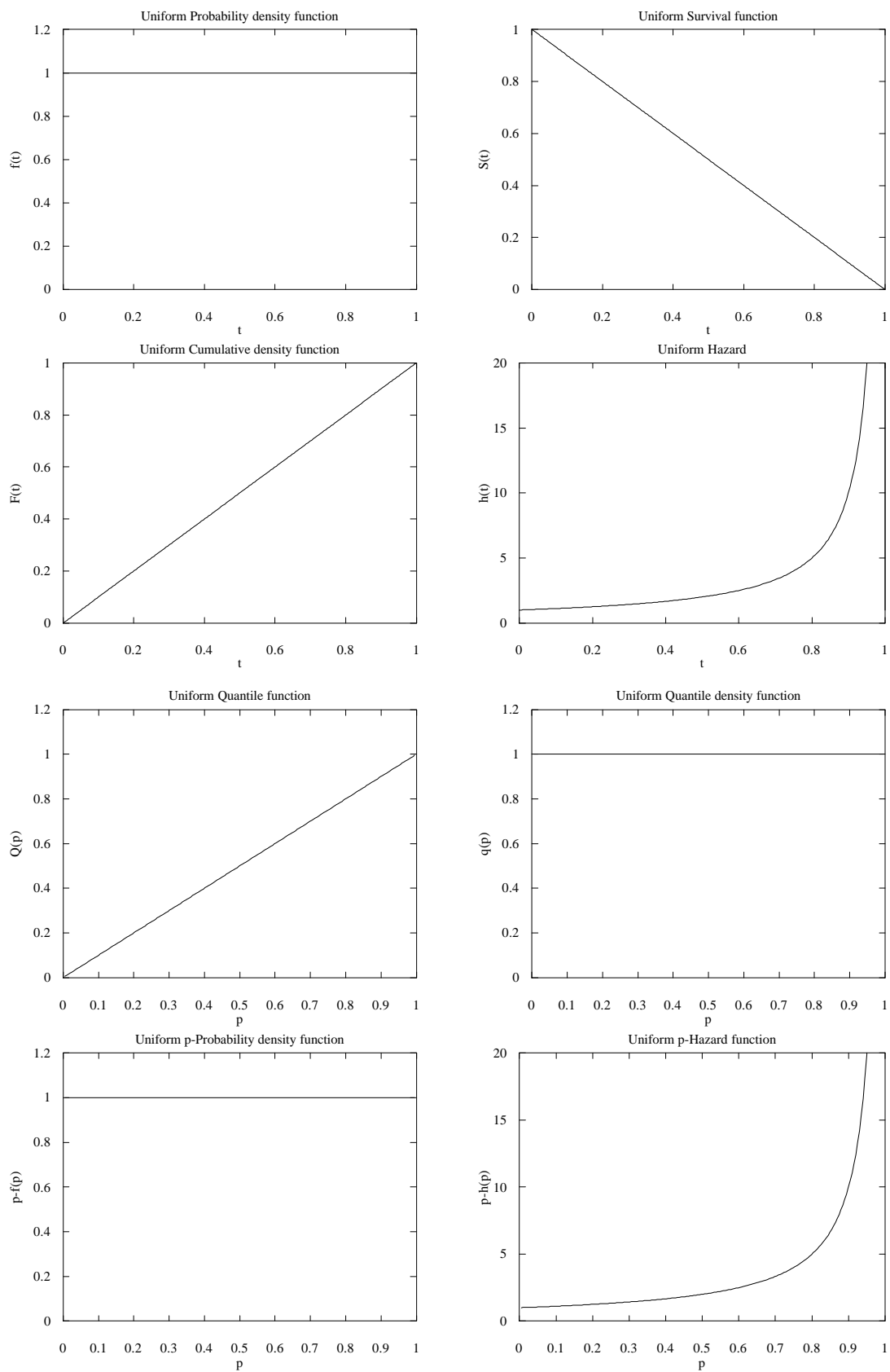
Parameters:	a (location), b (scale), c (shape)
Constraints:	$b \geq 0, c > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$-\infty < t < \infty$
PDF:	$f(t) = \frac{1}{b 2^{1+1/c} \Gamma(1+1/c)} e^{-\frac{1}{2} \left \frac{t-a}{b} \right ^c}$
SDF:	$S(t) = \frac{1}{2} - \frac{\operatorname{sgn}(t-a) \gamma\left(\frac{1}{c}, \frac{1}{2} \left \frac{t-a}{b} \right ^c\right)}{\Gamma(1/c)}$
Mean:	a
Median:	a
Mode:	a
Variance:	$\frac{b^2 2^{2/c} \Gamma(3/c)}{\Gamma(1/c)}$
Reduced models:	Approaches the rectangular distribution as $c \rightarrow \infty$, reduces to the Laplace distribution when c is 1, and reduces to the normal distribution when c is 2.
Other names:	Generalized Laplace.
References:	Johnson et al. (1995); Christensen (1984); Subbotin (1923)
See also:	NORMAL, LAPLACE, UNIFORM



UNIFORM or RECTANGULAR (Continuous)

The continuous uniform distribution has no parameters. However, truncation limits t_α and t_ω should be specified. When truncation limits are not specified (i.e. only one or two time variables are specified), then the standard uniform distribution is assumed and used: $t_\alpha = 0$ and $t_\omega = 1$.

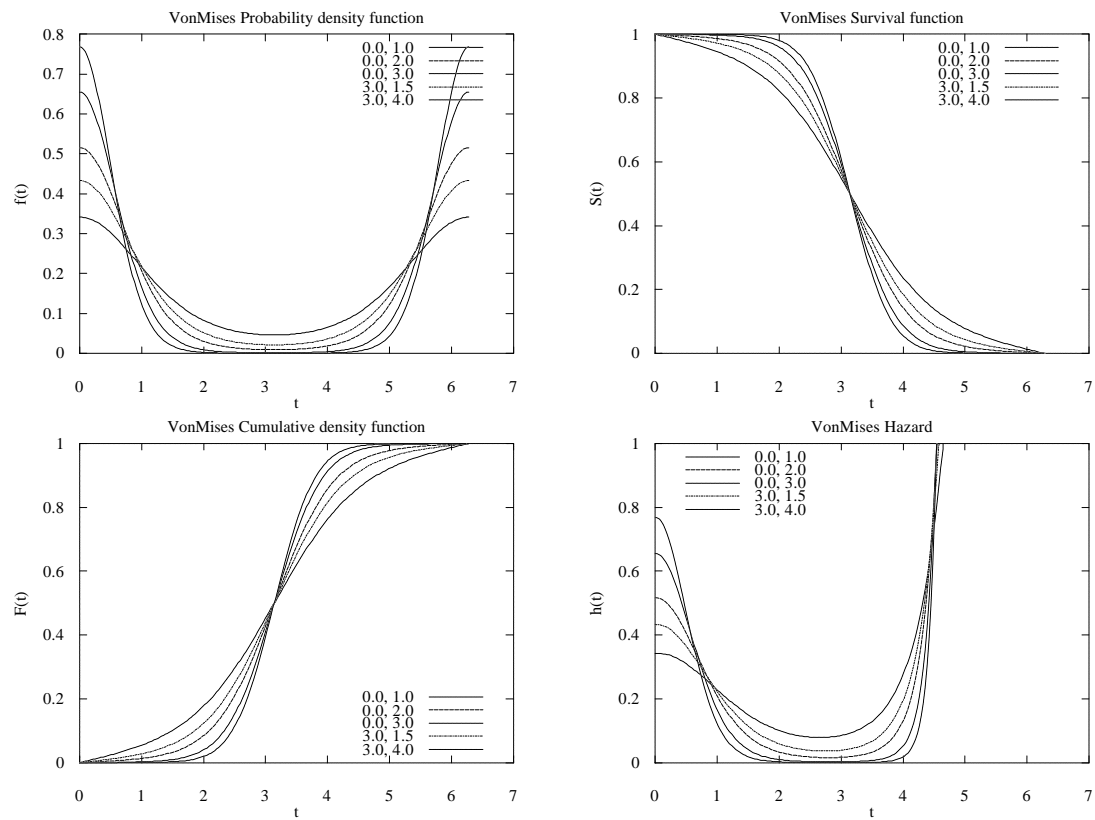
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$t_\alpha \leq t < t_\omega$
PDF:	$f(t) = 1/(t_\omega - t_\alpha)$
SDF:	$S(t) = \frac{t_\omega - t}{t_\omega - t_\alpha}$
Hazard:	$h(t) = 1/(t_\omega - t)$
Mean:	$(t_\omega - t_\alpha) / 2$
Median:	$(t_\omega - t_\alpha) / 2$
Variance:	$(t_\omega - t_\alpha)^2 / 12$
References:	Evans et al. (2000), Nelson (1982)



VONMISES

This is the von Mises distribution, which is the analog of a normal distribution on a circular range.

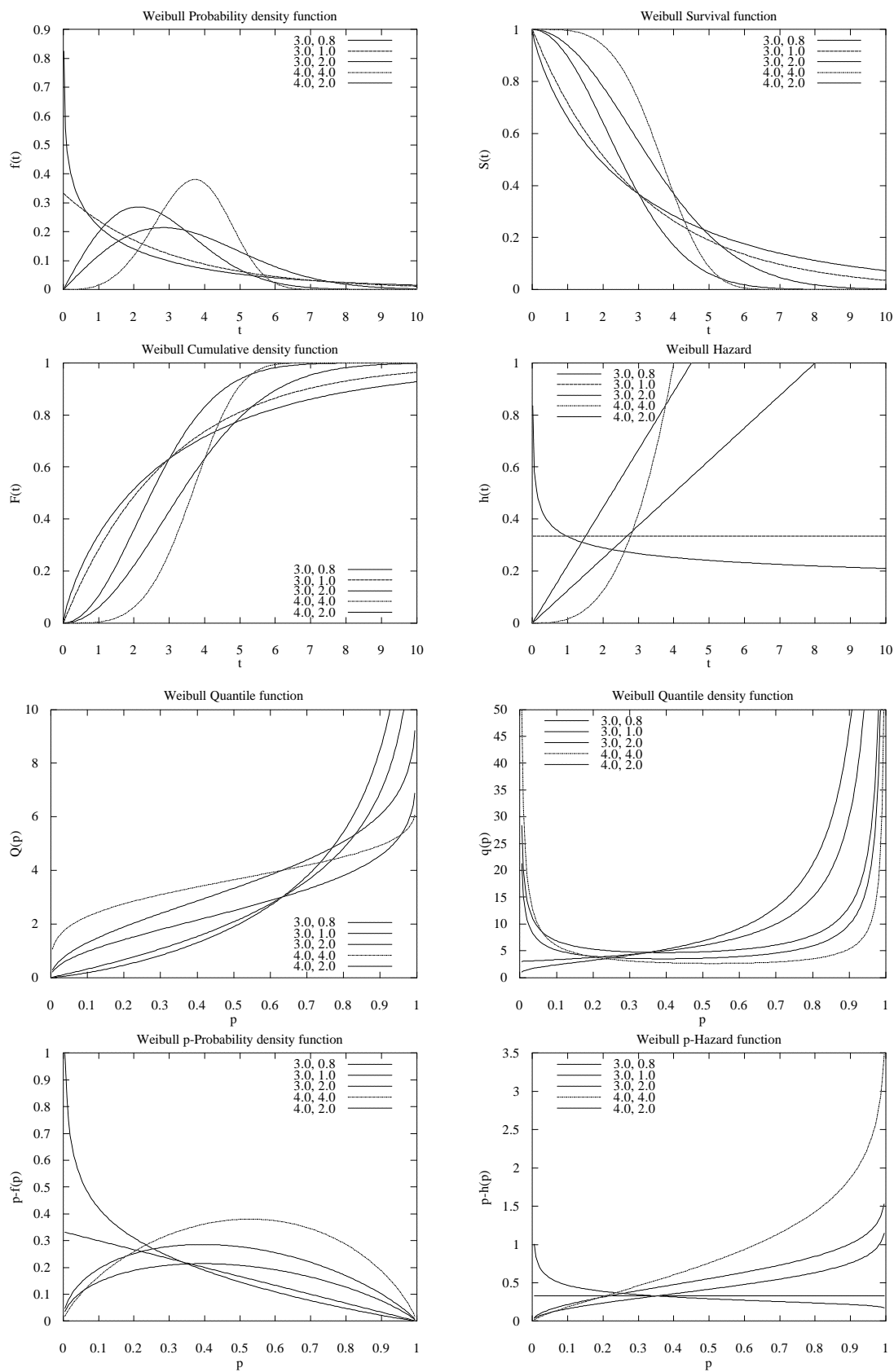
Parameters:	a (location), b (scale)
Constraints:	$0 < a < 2\pi$, $b > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$0 < t < 2\pi$
PDF:	$f(t) = \frac{\exp[b \cos(t - a)]}{2\pi I_0(b)}$
SDF:	$S(t) = 1 - \frac{t I_0(b) + 2 \sum_{j=1}^{\infty} [I_j(b) [\sin(jt - ja)] j^{-1}]}{2\pi I_0(b)}$
Mean direction:	a
Median:	a
Mode:	a
Antimode:	a
Circular variance:	$1 - I_1(b)/I_1(b)$
References:	Evans et al. (2000), Rao (1973)
Bugs:	The quantile functions do not work properly (see graphs).



WEIBULL

This is the Weibull distribution.

Parameters:	b (scale and characteristic life= 63rd percentile), c (shape).
Constraints:	$b > 0, c > 0$
Time variables:	t_u, t_e, t_α, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = ct^{c-1}b^{-c} \exp\left[-\left(\frac{t}{b}\right)^c\right]$
SDF:	$S(t) = \exp\left[-\left(\frac{t}{b}\right)^c\right]$
Hazard:	$h(t) = ct^{c-1}b^{-c}$
Quantile:	$b \ln[1/(1-p)]^{1/c}$
Mean:	$b\Gamma(1 + 1/c)$
Median:	$b\sqrt{\ln(2)}$
Mode:	$\begin{cases} b\sqrt[3]{1-1/c}, & c > 1 \\ 0, & c \leq 1 \end{cases}$
Variance:	$b^2 \left\{ \Gamma(1 + 2/c) - [\Gamma(1 + 2/c)]^2 \right\}$
Reduced models:	Reduces to the exponential distribution when $c = 1$, reduces to the Rayleigh distribution when $c = 2$.
Other names:	Frechet, generalized Rayleigh distribution, Weibull-Gnedenko.
References:	Christensen (1983), Evans et al. (2000), Nelson (1982)
See also:	SHIFTWEIBULL



Chapter 10

Discrete probability density functions

This chapter describes the discrete probability density functions that are defined in *mle*. Discrete PDFs are defined for integer values of “time” (or t).

Notation and format

The following type of information is typically given:

- A brief description of the PDF
- The intrinsic parameters and any constraints on the intrinsic parameters
- The “time” variables, and the range of non-zero probabilities.
- The probability density function
- The survival function
- The hazard function
- The quantile function.
- One or more measures of central tendency (mean, median, mode) and sometimes the variance given in terms of the intrinsic parameters.
- Reduced forms of the model, other names commonly used for the PDF, and references to related distributions and functions in How to read a data set into *mle*.

- References to the primary source, or to a source that further describes the characteristics of the distribution.

BERNOULLITRIAL

This distribution returns the probability from a single Bernoulli trial. The distribution has single intrinsic parameter (call it p) that is the probability of success. A single variable (call it *event*) is passed to the distribution and returns:

PDF: $p^{\delta(t,1)}(1-p)^{\delta(t,0)}$

$$p, \quad \text{for } event \neq 0$$

$$1 - p, \quad \text{for } event = 0$$

Range: $t = 0, 1$

Constraints: $0 \leq p \leq 1$

Mean: p

Variance: $p(1 - p)$

Notes: This is a special case of the Binomial distribution with $n = 1$. The PDF is p for event = 1, and $(1-p)$ for event = 0. Left- and right-truncation is not available. Covariate effects can be modeled on parameter p , but not on the hazard.

Example: fairness of a coins can be determined from tossing experiments as:

```
MODEL
DATA
  PDF BERNOULLITRIAL( is_heads )
  PARAM p LOW = -999 HIGH = 999 START = 0 FORM = LOGISTIC
  COVAR mint PARAM b_mint LOW = -5 HIGH = 5 START = 0 END
  COVAR year PARAM b_year LOW = -5 HIGH = 5 START = 0 END
  END {param p}
  END {pdf}
  END {data}
END {model}
```

See also: BINOMIAL

Reference: Bernoulli (1713)

BINOMIAL

This is the binomial distribution with two parameters. The distribution describes the number of successes in n independent Bernoulli trials, where each trial has probability of success p .

Parameters:	p (proportion), n (count of Bernoulli trials).
Constraints:	$0 \leq p \leq 1$, $0 < n < \infty$, n is an integer.
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$, t is an integer.
PDF:	$f(t) = \binom{n}{t} p^t (1-p)^{n-t}$
SDF:	$\begin{aligned} S(t) &= 1 - \sum_{i=1}^t \binom{n}{i} p^i (1-p)^{n-i} \\ &= B_p(t+1, n-t) \end{aligned}$
Mean:	np
Mode:	$p(n+1)$
Variance:	$np(1-p)$
References:	Bernoulli (1713); Rao (1973)
Reduced model:	Reduces to a BERNOULLITRIAL with $n = 1$
See also:	BERNOULLITRIAL, BETA

GEOMETRIC

This is the discrete geometric distribution,. The distribution describes the times up to and including the first success in a sequence of Bernoulli trials. The geometric distribution is the discrete analogue of the negative exponential distribution.

Parameters:	p (probability)
Constraints:	$0 \leq p \leq 1$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 1, t$ is an integer
PDF:	$f(t) = p(1 - p)^{t-1}$
SDF:	$S(t) = (1 - p)^t$
Mean:	$1/p$
Mode:	1
Variance:	$(1 - p)p^2$
Other names:	Furry
References:	Evans et al. (2000)
See also:	BERNOULLITRIAL, EXPONENTIAL, NEGBINOMIAL, HYPERGEOMETRIC

HYPERGEOMETRIC

This is the hypergeometric distribution.

Parameters:	p (probability), m, n
Constraints:	$t \geq 0$, $m - n + np \leq t \leq \min(m, np)$, $0 \leq p \leq 1$, $1 \leq m \leq n$, np is an integer.
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	integer $t \geq 0$
PDF:	$f(t) = \frac{\binom{np}{t} \binom{n-np}{m-t}}{\binom{n}{m}}$
SDF:	$S(t) = \sum_{i=0}^t \frac{\binom{np}{i} \binom{n-np}{m-i}}{\binom{n}{m}}$
Mean:	mp
Mode:	$\frac{(np+1)(m+1)}{n+2}$
Variance:	$\frac{mp(p+1)(n-m)}{n-1}$
References:	Laplace (1774)
See also:	GEOMETRIC

LOGSERIES

This is the logarithmic series distribution. This distribution can be derived from both a power series distribution and a negative binomial distribution.

Parameters:	p (probability)
Constraints:	$0 \leq p \leq 1$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	integer $t \geq 1$
PDF:	$f(t) = -\frac{p^t}{t \ln(1-p)}$
SDF:	$S(t) = 1 + \frac{1}{\ln(1-p)} \sum_{j=1}^t \frac{p^j}{j}$
Mean:	$-\frac{p}{\ln(1-p)(1-p)}$
Mode:	1
Variance:	$-\frac{p \left(1 + \frac{p}{\ln(1-p)} \right)}{\ln(1-p)(1-p)^2}$
References:	Christensen (1984), Evans et al. (2000), Fisher (1943)
See also:	POWERSERIES, NEGBINOMIAL

NEGBINOMIAL

This is the negative binomial distribution, also known as the Pascal distribution when t is an integer.

Parameters:	p (probability), n (count).
Constraints:	$0 \leq p \leq 1, n > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	$t \geq 0$
PDF:	$f(t) = \binom{t+n-1}{n-1} p^n (1-p)^t$ (this is wrong--need continuous version)
SDF:	$S(t) = \beta_p(n, t+1)$
Mean:	$n(1-p)/p$
Mode:	$\text{Floor}[p(n-1)/(1-p)]$
Variance:	$n(1-p)/p^2$
Reduced models:	Reduces to a Pascal distribution when t is an integer. Reduces to the geometric distribution when $n = 1$.
References:	Christensen (1984), Evans et al. (2000)
See also:	GEOMETRIC, POWERSERIES, BINOMIAL

NEGHYPERGEOMETRIC

This is the negative hypergeometric distribution. It is a special case of the beta-binomial distribution. The distribution describes the number of draws without replacement from an urn initially filled with n balls, fraction p that are marked, until m marked balls are drawn.

Parameters: p (probability), m (count), n (count).

Constraints: $0 < p \leq 1$, $0 < m < np$, $n > 0$, np is an integer.

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.

Range: $m \leq t \leq m + n - np$

PDF:
$$f(t) = \frac{\binom{np}{m-1} \binom{n-np}{t-m}}{\binom{n}{t-1}} \frac{np-m+1}{n-t+1}$$

Mean: $m(n+1)/(np+1)$

Mode: $n(m-1)/(np-1) + 1$

Variance: $mn(n+1)(1-p)(np+1-m)/[(np+2)^2(np+1)]$

References: Johnson and Kotz (1969); Christensen (1984).

See also: HYPERGEOMETRIC

PASCAL

This is the Pascal distribution, which is the discrete version of the negative binomial distribution.

Parameters:	p (probability), n (count).
Constraints:	$0 \leq p \leq 1$, integer $n > 0$
Time variables:	$t_u, t_e, t_{\alpha}, t_{\omega}$. An exact failure is defined when $t_u = t_e$.
Range:	integer $t \geq 0$
PDF:	$f(t) = \binom{t+n-1}{n-1} p^n (1-p)^t$
SDF:	$S(t) = \beta_p(n, t+1)$
Mean:	$n(1-p)/p$
Mode:	$\text{Floor}[p(n-1)/(1-p)]$
Variance:	$n(1-p)/p^2$
Reduced models:	Reduces to the geometric distribution when $n = 1$.
References:	Christensen (1984), Evans et al. (2000)
See also:	NEGBINOMIAL, GEOMETRIC, POWERSERIES, BINOMIAL

POISSON

This is the discrete Poisson distribution.

Parameters:	λ (hazard)
Constraints:	$\lambda > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	integer $t \geq 0$
PDF:	$f(t) = \frac{\lambda^t e^{-\lambda}}{t!}$
SDF:	$S(t) = \frac{\Gamma(t+1, \lambda)}{\Gamma(t+1)}$
Mean:	λ
Mode:	Floor(λ)
Variance:	λ
References:	Christensen (1984), Evans et al. (2000)
See also:	POWERSERIES

POLYAEGGENBERGER

This is the Polya-Eggenberger distribution. It is a type of contagious distribution. The distribution describes the number of times a marked ball is drawn from an urn in the following experiment. Initially fraction p of the m initial balls are marked. One ball is drawn, and if it is marked, the ball is replaced with $m \cdot c$ additional balls. The procedure is repeated a total of n times.

Parameters: p (probability), n (count), c (shape).

Constraints: $0 \leq p \leq 1, n \geq 1, c > -(n-1)^{-1}$.

Time variables: t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.

Range: $0 \leq t \leq n$

PDF:
$$f(t) = \frac{\binom{n}{t} \prod_{j=0}^{t-1} (p + cj) \prod_{j=0}^{n-t-1} (1 - p + cj)}{\prod_{j=0}^{n-1} (1 + cj)}$$

Mean: np

Mode: $(n+1)(p-c)/(1-2c)$

Variance: $np(1-p)(1+nc)/(1+c)$

References: Johnson and Kotz (1969); Christensen (1984).

See also: HYPERGEOMETRIC, BINOMIAL

STERILE or IMMUNE

This distribution is used to form degenerate distributions of the forms $f(t) = (1 - p)f_1(t)$ and $S(t) = (1 - p)S_1(t) + p$. This is done by using the `MIX` function with `STERILE` and specifying a mixture of $f_1(t)$ and the `STERILE` distribution. Given failure times t_u and t_e and, perhaps, the left truncation limits t_α and t_ω , this distribution returns:

0 if an exact failure or an interval censored failure occurs: $t_e < t_\omega$

1 if a right censored observation occurs: $t_e \geq t_\omega$

In words, the distribution returns 0 if there is no possibility that the observation may have been "sterile" (because some failure was observed), and returns 1 if there is a possibility that the observation was a "sterile" observation (right censored observation). This distribution has no intrinsic parameters, and covariates cannot be modeled on the hazard function of this distribution.

Example. Suppose individuals fail with an underlying normal distribution, but the population is contaminated by an unknown fraction, p , of non-susceptible individuals. The *mle* code is:

```
MODEL
DATA
  MIX(
    PARAM p LOW=0 HIGH=1 FORM=NUMBER END
    ,
    PDF NORMAL( last_alive first_dead )
      PARAM mu LOW=100 HIGH=200 START=150 END
      PARAM sigma LOW=0.1 HIGH=20 START=10 END
    END {pdf}
    ,
    PDF STERILE( last_alive first_dead ) END
  )
END {data}
END {model}
```

Call L_n the likelihood from the normal PDF. Then, the likelihood for an exact failure or interval censored failure will be pL_n . For right censored observations, the likelihood will be $pL_n + (1-p)$.

References: Holman (1995, 1998), Nelson (1982)

THOMAS

This is the Thomas distribution, which was originally used describes the abundance of plant species in a quadrat (square lattice) based on the notion that some plants tend to cluster. The process is developed from a Poisson distribution of clusters distributed in space and for each of those clusters a poisson distribution for the number of subpoints within the cluster (see Thomas 1949). The distribution describes the probability of finding 0 plants, 1 plant, 2 plants, . . . in a randomly selected quadrat.

Parameters:	a, b
Constraints:	$a > 0, b > 0$
Time variables:	t_u, t_e, t_o, t_w . An exact failure is defined when $t_u = t_e$.
Range:	$0 < t < \infty$
PDF:	$f(t) = e^{-a} \sum_{i=1}^t \frac{a^i e^{-ib} (jb)^{t-i}}{i!(t-i)!}$
SDF:	$S(t) = 1 - \sum_{k=0}^t \left[e^{-a} \sum_{i=1}^k \frac{a^i e^{-ib} (jb)^{k-i}}{i!(k-i)!} \right]$
Mean:	$a(1 + b)$
Variance:	$a(1 + 3b + b^2)$
Other names:	double Poisson.
References:	Thomas (1949); Christensen (1984)
See also:	POISSON

ZIPF

This is the zipf distribution. This distribution has been used to describe word frequency distributions, species abundances, and many other natural phenomena.

Parameters:	a
Constraints:	$a > 0$
Time variables:	$t_u, t_e, t_\alpha, t_\omega$. An exact failure is defined when $t_u = t_e$.
Range:	integer $t \geq 1$
PDF:	$f(t) = \frac{t^{-(a+1)}}{\zeta(a+1)}$
SDF:	$f(t) = 1 - \frac{\sum_{j=0}^t j^{-(a+1)}}{\zeta(a+1)}$
Mean:	$\zeta(a)/\zeta(a+1)$
Mode:	1
Variance:	$\zeta(a-1)\zeta(a+1) - \zeta(a)^2/\zeta(a+1)^2$
Other names:	Zeta distribution, Zipf-Estoup law, discrete Pareto distribution..
References:	Christensen (1984), Zipf (1949), Johnson and Kotz (1969)
See also:	PARETO, LOGSERIES, function ZETA

Chapter 11

Mathematical symbols and functions

Symbols

T	Denotes a random variable, whether continuous or discrete.
t	An instance of random variable T .
ρ	A parameter that defines a correlation coefficient.
σ	A scale parameter that defines the standard deviation of the distribution.
μ	A location parameter that also defines the mean of the distribution.
a	A location parameter
b	A scale parameter
c	A shape parameter
h	A parameter that is directly interpretable as a hazard.
p	A parameter that is directly interpretable as a probability.
t_q	The q th quantile: $[1-S(t_q)] = q$.
$f(t)$	The probability density function (PDF): $f(t) = df(t)/dt$.
$F(t)$	The cumulative distribution function (CDF): $F(t) = \int_0^t f(x)dx$.
$S(t)$	The survival distribution (SDF): $S(t) = 1 - \int_0^t f(x)dx = \int_t^\infty f(x)dx$.
$h(t)$	The hazard function: $h(t) = \frac{f(t)}{S(t)}$.

$$\frac{d[g(x)]}{dx}$$

The first derivative of the function $g(x)$

Constants

π	PI, The value pi $\approx 3.141\ 592\ 653\ 589\ 793\ 238\ 462\ 643$
e	E, The base of the natural log $\approx 2.718\ 281\ 828\ 459\ 045\ 235\ 360\ 287$
γ	EULERSC, Euler's constant $\approx 0.577\ 215\ 664\ 901\ 532\ 860\ 606\ 512$

Function Definitions

$\chi_q^2(x)$	The Chi-squared q quantile: $\Pr[\chi^2 \leq \chi_q^2(x)] = q$
$\Phi_q(x)$	The standard normal q quantile: $\Pr(x \leq \Phi_q) = q$
$\Phi(x)$	The standard normal cumulative density function: $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{-\frac{u^2}{2}} du.$
$\text{erf}(x)$	The error function, ERF(x): $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du = 2\Phi(x\sqrt{2}) - 1.$
$I_k(x)$	The modified Bessel function of the 1 st kind, order k , BESSELI(k, x): $I_k(x) = \frac{x^k}{2^k} \sum_{j=0}^{\infty} \frac{\left(\frac{1}{4}x^2\right)^j}{j!\Gamma(1+j+k)}.$
$I_q^{-1}(\nu, \omega)$	Inverse beta function.
$K_k(x)$	The modified Bessel function of the 2 nd kind, order k , BESSELK(k, x): $K_k(x) = \pi \frac{I_{-k}(x) - I_k(x)}{2 \sin(i\pi)}.$
$B(\nu, \omega)$	The beta function BETAF(ν, ω): $B(\nu, \omega) = \int_0^1 x^{\nu-1} (1-x)^{\omega-1} dx = \frac{\Gamma(\nu)\Gamma(\omega)}{\Gamma(\nu+\omega)}.$
$B_p(\nu, \omega)$	The normalized incomplete beta function, IBETA(ν, ω): $B_p(\nu, \omega) = \frac{\int_0^p x^{\nu-1} (1-x)^{\omega-1} dx}{B(\nu, \omega)}$

$\beta_p(v, \omega)$	The complement of the normalized beta function (IBETAC): $\beta_p(v, \omega) = 1 - B_p(v, \omega)$.
$\Gamma(v)$	The gamma function (GAMMAF): $\Gamma(v) = \int_0^{\infty} x^{v-1} e^{-x} dx$.
$\gamma(v, \omega)$	The incomplete gamma function of the 1 st kind: $\gamma(v, \omega) = \int_0^{\omega} x^{v-1} e^{-x} dx$
$\Gamma(v, \omega)$	The incomplete gamma function of the 2 nd kind: $\Gamma(v, \omega) = \int_{\omega}^{\infty} x^{v-1} e^{-x} dx$ The IGAMMA(x, y) function returns $\gamma(v, \omega)/\Gamma(v)$ The IGAMMAC(x, y) function returns $\Gamma(v, \omega)/\Gamma(v) = 1 - \gamma(v, \omega)/\Gamma(v)$
$\psi(x)$	The digamma function. $\psi(x) = \frac{d[\ln \Gamma(x)]}{dx}$.
$\psi'(x)$	The trigamma function. $\psi'(x) = \frac{d[\psi(x)]}{dx} = \frac{d^2[\ln \Gamma(x)]}{dx^2}$.
$\ln(x)$	The natural (Napierian) log of x . LN(x) and LOG(x)
$\delta(x, y)$	Kronecker's delta function: $\delta(x, y) = \begin{cases} 1, & x = y \\ 0, & x \neq y \end{cases}$. DELTA(x, y).
$\binom{n}{k}$	Combinations of n taken k at a time $= \binom{n}{k} = \frac{n!}{(n-k)!}$. COMB(n, k)

The Greek Alphabet

A	α	alpha	I	ι	iota	P	ρ	rho
B	β	beta	K	κ	kappa	Σ	σ	sigma
Γ	γ	gamma	Λ	λ	lambda	T	τ	tau
Δ	δ	delta	M	μ	mu	Y	υ	upsilon
E	ϵ	epsilon	N	ν	nu	Φ	ϕ	phi
Z	ζ	zeta	Ξ	ξ	xi	X	χ	chi
H	η	eta	O	\omicron	omikron	Ψ	ψ	psi
Θ	θ	theta	Π	π	pi	Ω	ω	omega

Metric Prefixes

10	deka (da)	10^{-1}	deci (d)
10^2	hecto (h)	10^{-2}	centi (c)
10^3	kilo (k)	10^{-3}	milli (m)
10^6	mega (M)	10^{-6}	micro (μ)
10^9	giga (G)	10^{-9}	nano (n)
10^{12}	tera (T)	10^{-12}	pico (p)
10^{15}	peta (P)	10^{-15}	femto (f)
10^{18}	exa (E)	10^{-18}	atto (a)
10^{21}	zetta (Z)	10^{-21}	zepto (z)
10^{24}	yetta (Y)	10^{-24}	yocto (y)

International Electrotechnical Commission Prefixes for Binary Multiples

2^{10}	kibi (Ki)
2^{20}	mebi (Mi)
2^{30}	gibi (Gi)
2^{40}	tebi (Ti)
2^{50}	pebi (Pi)
2^{60}	exbi (Ei)

Selected Syst me International d'Unit s

quantity	symbol	name	derivation	other units
absorbed dose	Gy	gray	J/kg	1 rad = 0.01 Gy
acceleration			m/s ²	
activity (radionuclide)	Bq	becquerel	s ⁻¹	1 curie (Ci) = 3.7×10 ¹⁰ Bq
amount of substance	mol	mole	base unit	
angular acceleration			rad/s ²	
angular velocity			rad/s	
area		square meter	m ²	
capacitance (electrical)	F	farad	A s/V	
charge (electrical)	C	coulomb	A s	1 electrostatic units (esu) = 3 ⁻¹ ×10 ⁻⁹ C; 1 Franklin (Fr) = 3.335641×10 ⁻¹⁰ C
conductance (electrical)	S	siemens	A/V	1 mho = 1 S
density			kg/m ³	
dose equivalent	Sv	sievert	J/kg	1 rem = 0.01 Sv
electric current	A	ampere	base unit	1 biot (Bi) = 10 A; 1 gilbert (Gi) = 0.7957747 A
energy, work, quantity of heat	J	joule	N m	1 calorie (cal) = 4.184 J; 1 erg = 10 ⁻⁷ J; 1 British thermal unit (BTU) = 1055.87 J; 1 foot-pound (ft-lb) = 1.35582 J; 1 electronvolt (eV) = 1.602177×10 ⁻¹⁹ J;
field strength (electrical)			V/m	
field strength (magnetic)			A/m	1 oersted (Oe) = 4 ⁻¹ ×10 ³ A/m
force	N	newton	kg m/s ²	1 dyne (dyn) = 10 ⁻⁵ N
frequency	Hz	hertz	s ⁻¹	1 cycle per second (cps) = 1 Hz
illuminance	lx	lux	lm/m ²	1 footcandle = 1.076391 lx
inductance	H	henry	V s/A	
length	m	meter	base unit	1 astronomical unit (AU) = 1.495979×10 ¹¹ m; 1 light year (l.y.) = 9.46073×10 ¹⁵ m; 1 angstrom (�) = 10 ⁻¹⁰ m
luminous intensity	cd	candela	base unit	
luminance			cd/m ²	1 lambert = 3.183099 cd/m ²
luminous flux	lm	lumen	cd sr	
magnetic flux	Wb	weber	V/s	1 maxwell (Mx) = 10 ⁻⁸ Wb
magnetic flux density	T	tesla	Wb/m ²	1 gauss (Gs) = 10 ⁻⁴ T
magnetomotive force	A	ampere		
mass	kg	kilogram	base unit	1 metric ton = 10 ³ kg; 1 metric carat = 2×10 ⁻⁴ kg
mass density			kg/m ³	
plane angle	rad	radian	m/m	degree (�); minute ('); second ("); 1 grad (gon) = 1.570796×10 ⁻² rad
power	W	watt	J/s	1 horsepower (hp) = 745.7 W
pressure	Pa	pascal	N/m ²	atmosphere, atm = 1.01325×10 ⁵ N/m ² ; 1 bar = 10 ⁵ N/m ² ; 1 psi = 6.894757×10 ³ Pa; 1 Torr = 133.3224 Pa
resistance (electrical)	�	ohm	V/A	
solid angle	sr	steradian	m ² /m ²	degree (�); minute ('); second (")
specific volume			m ³ /kg	
temperature	K	kelvin	base unit	degree Celsius �C, degree Fahrenheit �F
time	s	second	base unit	day (d); hour (h); minute (min); 1 shake = 1 ns
velocity, speed			m/s	
voltage, electromotive force, electrical potential	V	volt	W/A	
volume		cubic meter	m ³	1 liter (L) = 10 ⁻³ m ³

Angles

	"	'	°	radians	grads (gons)
"	1	0.0166667	0.000277778	0.0159155	0.0176839
'	60	1	0.0166667	0.954930	1.06103
°	3600	60	1	57.2958	63.6620
radians	62.8319	1.04720	0.0174533	1	1.11111
grads (gons)	56.5487	0.942478	0.0157080	0.9	1

Time

	second	minute	hour	day	week	year
second	1	0.0166667	0.000277778	1.15741E-05	1.65344E-06	3.17969E-08
minute	60	1	0.0166667	0.000694444	9.92063E-05	1.90781E-06
hour	3600	60	1	0.0416667	0.00595238	0.000114469
day	86400	1440	24	1	0.142857	0.00274725
week	604800	10080	168	7	1	0.0192308
year	31449600	524160	8736	364	52	1

Temperature Conversions

	a °C (Celsius)	b K (Kelvin)	c °F (Fahrenheit)	d °R (Rankine)
a °C	a	$a = b - 273.15$	$a = (c - 32)/1.8$	$a = d/1.8 - 273.15$
b K	$b = a + 273.15$	b	$b = (c + 459.67)/1.8$	$b = d/1.8$
c °F	$c = 1.8a + 32$	$c = 1.8b - 459.67$	c	$c = d - 459.67$
d °R	$d = 1.8a + 491.67$	$d = 1.8b$	$d = c + 459.67$	d

Chapter 12

Error and warning messages

A number of error and warning messages are produced by *mle*. This chapter describes the most common error and warning messages. Messages are categorized in broad categories

- Help messages and other messages that occur in response to improper command line options.
- Warning messages are given for models, parameters and other iterative functions that might not completely converge.
- Error messages cause *mle* to stop running. They can be roughly subcategorized as coming from the following sources:
 - ⇒ run-time routines
 - ⇒ Parsing routines
 - ⇒ Data routines.
 - ⇒ Mathematical library
 - ⇒ Symbol table routines

Messages from Command Line Options

The following message is printed when command line options are not recognized and at least one of the options is taken as an input file. For example, typing `mle xxx` yields:

```
Error: Couldn't find program file xxx

Usage: mle -v -p -i -I <path> -rr -rw -t -d<X> mlefile args
  -v      Iteration histories and other messages are written to the screen
  -p      Only parses the mle file
  -i      Runs mle interactively
  -I <path> Searches path for INCLUDE files
  -sr      Read START values from restart files <name><m#>.<r#>
  -sw      Writes START values to restart files <name><m#>.<r#>
  -t      Terminate model when file <name>.TRM exists
  -d<X>   Debugging: x execution, e echo, i integration
              l likelihood, p parser, s symbol table
mlefile is the name of the program file
args are optional arguments to the program

Usage: mle -h [name1 name2 . . . .]
  help for PDFs, functions, symbols, parameter transforms
  -h matches words exactly, -H searches within words

Usage: mle -pn n1 n2 . . . .
  parses n's and returns values and type
```

◇ *File <name> does not exist. Try again.*

Typing `mle` on the command line will result in the message

```
mle Program file to run?
```

Should you type a file name that does not exist, the following message appears:

```
File asd does not exist. Try again.
mle Program file to run?
```

Ensure the proper directory and file extension is being used. Note that *mle* does not automatically append `.mle` to the input file.

Warning Messages

Warning messages come from routines that iteratively attempt to find a solution of parameters or other functions. Warning messages will not result in termination of the *mle* run.

◇ *Warning FINDMIN reached maximum iterations*

◇ *Warning FINDZERO reached maximum iterations*

The FINDMIN or FINDZERO function was not evaluated to the specified tolerance in the specified number of iterations. You can increase the number of iterations to the function (one way is by increasing the value of FIND_MAXITS) or decrease the convergence criterion (one way is by decreasing the value of FIND_EPS).

◇ *Warning: gamma SDF (by continued fractions) did not completely converge*

◇ *Warning: gamma SDF (by series) did not completely converge*

These two messages suggest that some evaluation of the Euler's incomplete gamma function was not very precise.

◇ *Warning: beta CDF did not converge*

This message suggests that some evaluation of the incomplete Beta function was not very precise.

- ◇ *Warning: Upper [Lower] Interval for param x did not converge to x.xxx in yyyy iterations*

This message arises when **mle** has troubles finding the upper or lower limit of a likelihood confidence interval. The number of iterations should be increased (set CI_MAXITS to a higher value), or the convergence criterion should be relaxed (set CI_CONVERGE to a larger value).

- ◇ *Warning: Upper [Lower] Interval for param x not bound between xxx and yyy.*

This message arises when a confidence interval is larger than the upper and lower limits of the parameter. The HIGH or LOW limits for the parameter should be changed so that the confidence limit is within the limits of the parameters.

- ◇ *Warning: the matrix is singular*

This message indicates that the variance-covariance matrix could not be computed because the observed Fisher's information matrix was singular. This occurs when one or more parameters have very large standard errors, or changes in the parameter do not affect the likelihood. Some suggestions are: reduce the number of parameters, ensure all parameters affect the likelihood, solve the likelihood to a higher precision, transform one or more parameters so that the parameter estimate is not near a mathematical limit. This last situation occurs, for example, when a probability is modeled untransformed (between 0 and 1) and the parameter estimate is near 0 or 1. Using a logistic specification for the parameter will sometimes fix the problem.

Run-time Errors

- ◇ *Error (run time): <name> can't be called with subscripts*
- ◇ *Error (run time): Missing subscript, dimension <n> for <name>*
- ◇ *Error (run time): Subscript out of range for <name>. Found <n>. Valid range is [<n1> to <n2>]*
- ◇ *Error (run time): Calling boolean func <name> + with <type> args*
- ◇ *Error (run time): assigning a STRING with length <> 1 to a CHAR*
- ◇ *Error (run time): CHAR function got a string of length <> 1*
- ◇ *Error (run time): Can't assign a file type variable*
- ◇ *Error (run time): Bad <type> identifier type found*
- ◇ *Error (run time): calling ROUND(<r>): out of range of integer*
- ◇ *Error (run time): Tried assigning null string to char var <name>*

A null string (i.e. a string of length zero specified by "") cannot be assigned to a character variable.

- ◇ *Error (run time): Unimplemented or unknown METHOD: <name>*

The requested maximization method (set by METHOD=<name>) is not recognized.

- ◇ *Error (run time): Unimplemented integration method: <name>*

The requested integration method (set by INTEGRATE_METHOD=<name>) is not recognized.

- ◇ *Error (run time): Bad string in STRING2REAL*

A string argument to the STRING2REAL function could not be converted into a real number.

- ◇ *Error (run time): Bad string in STRING2INT*

A string argument to the STRING2INT function could not be converted into an integer.

- ◇ *Error (run time): Bad real ident. <name> is type <type>*

- ◇ *Error (run time): Bad integer identifier type found*

An argument to an integer function was not an integer.

- ◇ *Error (run time): Bad string identifier type found*

An argument to a string function was not a string or character.

- ◇ *Error (run time): Calling boolean func <name> with [real, integer, string/char, boolean] args*

- ◇ *Error (run time): Type mismatch for arg n calling func <name>*

The type (integer, real, boolean, string, character) for argument *n* of function <name> was incorrect.

- ◇ *Error (run time): Opening [INFILE, DATAFILE, MLERC] file xxxx: <message>*

An error occurred while opening a file. The <message> can be one of the following:

File was not found — a bad file name was specified

Path was not found — a bad path name was specified

Too many open files — more files were opened than are allowed by the operating system

File access denied — the user does not have permission to access the file (unix) or the file is locked (DOS)

Invalid file reference — the file name is not valid

Not enough memory — memory was unavailable for some operation

Invalid environment — an environment variable is bad

Invalid drive letter — an invalid drive letter was specified

Can't remove current directory — an remove-directory operation failed. The directory has files in it or is protected.

Can't rename files across drives — tried to rename a file across logical or physical disk drives

Disk read error — an error occurred reading a disk. This is usually a hardware problem.

Disk write error — an error occurred trying to write to a disk. This is usually a hardware problem.

Disk is write-protected — an error occurred trying to write to a write-protected disk.

Unknown device — a device name was not recognized.

Disk drive is not ready

Disk seek or sector error — an error occurred trying to read or write a disk. This is usually a hardware problem.

Unknown media — a bad disk was put into a disk drive or a hardware error occurred.

The printer is out of paper — A printer device ran out of paper.

Error trying to write to the output device

Error trying to read an input device

A hardware failure occurred

Errors from the Parser

Error messages from the parser always contain information on the line and column where the error occurred.

- ◇ *Error found while parsing <id> at line <line#> column <column#>
Expected a positive constant instead of "<text>"*

A positive constant is expected in the FIELD and LINE specifications of the DATA statement.

- ◇ *Boolean expression was expected*

A boolean expression was expected but not found. For example, as the first expression in the IF...THEN...ELSE...END function must be a boolean expression.

- ◇ *"<name>" must be previously declared for use here*

A variable used in an expression had not been previously declared. All variables must be declared, either as a predeclared variable, in a PARAM function, in the DATA statement, or in an ASSIGN statement before being used in an expression.

- ◇ *"<name>" exists and cannot be declared as a PARAM*

An attempt was made to declare <name> as a parameter, but it was previously declared.

- ◇ *<name> doesn't exist, so it can't be reduced.*

The parameter <name> found in a REDUCE statement does not exist.

- ◇ *Bad argument type to [function] <func>. Expected <type> but found <type>*

An argument to the named function is not correct.

- ◇ *<variable> already exists. It cannot be a DATA variable.*

A variable defined in the DATA statement already exists.

◇ *Bad type in assign statement*

The resulting type on the right-hand side of the assignment is incompatible with the left-hand side.

◇ *Can't assign value of type <type> to variable of type <type>.*

The two types are incompatible.

◇ *Bad number format found while scanning "<text>"*

The text was supposed to be converted into a number, but could not be properly converted. Usually there is an invalid character in the sequence of digits/letters that form the number.

◇ *Character constant is too long*

A character constant had more than one character in it. Usually this is because a single quote (') was inadvertently used instead of double quotes (") to denote a string constant.

◇ *Unclosed comment at end of file*

This results when a comment is not properly closed.

◇ *<name> is not an array and cannot be subscripted*

◇ *Non-integer subscripts found for variable <name>*

◇ *Wrong number of dimensions assigning <name> an array/variable function. Expected <n1> found <n2>.*

◇ *Wrong number of dimensions assigning <name1> passed as a VAR parameter to <name2>. Expected <n1> found <n2>.*

◇ *Bad subscripts assigning <name> an array/variable function. Dimension <n>. <name> range: <n1> to <n2>. Found: <n3> to <n4>.*

◇ *Bad subscripts assigning <name> passed as a VAR parameter to <name2>. Dimension <n>. <name> range: <n1> to <n2>. Found: <n3> to <n4>.*

◇ *Subscript out of range. Found <n1>. Valid range is <n2> to <n3>.*

◇ *Cannot assign an array variable/function to scalar <name>.*

◇ *Can't assign to constant <name>.*

◇ *Arg passed to VAR parameter <name> is the wrong type. It must be <type>.*

◇ *Arg passed to VAR parameter <name> is a constant.*

◇ *Incompatible arg (type <type>).*

◇ *Incorrect number of dimensions in DATAARRAY. Found <n1> expected <n2>.*

◇ *Subscripted variable <name> does not exist.*

◇ *FILE arguments must be VAR parameters*

A subscript was either higher than the highest declared value or lower than the lowest declared value.

◇ *Wrong number of arguments passed to <function>: <n1> expected, <n2> found.*

The wrong number of arguments were passed to the simple function specified.

◇ *Error: bad syntax at: line <line#> column <column#>*

The bad token is:<id>

One of the following was expected:

This error occurs when improper syntax is found.

Error Messages from Data Routines

◇ *Error (data): Data transformation, bad function type*

The function defined for a data transformation does not return a real or integer type.

◇ *Error (data): Unexpected end of file <name> reading observation <n> line <n> of <n>, field <n> of <n>*

The file ended when an observation was only partially read.

◇ *Error (data): Unexpected end of line <name> reading observation <n> line <n> of <n>, field <n> of <n>*

The line ended when an observation was only partially read.

◇ *Bad value <n1> line <n2> field <n3>. Can't convert: <text> to a number.*

The value for observation *n1*, on line *n2* in field *n3* couldn't be properly converted to a number.

◇ *Error (data): No data file assigned. Use DATAFILE procedure'*

The data file was not assigned using DATAFILE procedure. Place the statement DATAFILE (" <name> ") before the DATA statement.

◇ *Can't have <n1> observations per line. Max is <n2>*

Too many lines per observation was specified in a LINES_PER_OBS = <n1> statement.

◇ *Can't read in <n1> fields. Max is <n2>*

Too many fields were specified for the FIELD <n1> part of a DATA statement.

Error Messages from Function Calls:

A number of errors arise from improper values being passed to function calls. Frequently the remedies are to place proper constraints on parameter values, clean the input data, transform data, or add a small positive number to all failure times. The following messages arise from these difficulties

- ◇ *Error (math): Attempted division by zero*
- ◇ *Error (math): Attempted to take the square root of a negative number: <-nnnn>*
- ◇ *Error (math): Bad BIVNORMAL param rho. Expected: $-1 \leq r \leq 1$ but = <nnn>*
- ◇ *Error (math): Bad <name> should be > 0 and < 1 but = <nnn>*
- ◇ *Error (math): Bad <name> should be ≥ 1 but = <nnn>*
- ◇ *Error (math): Bad <name> should be ≥ 0 but = <nnn>*
- ◇ *Error (math): Bad <name> should be > 0 but = <nnn>*
- ◇ *Error (math): Bad <name> cannot be $= 0$ but is*
- ◇ *Error (math): Bad <name> should be > -1 and < 1 but = <nnn>*
- ◇ *Error (math): Bad <name> should be ≥ -1 and ≤ 1 but = <nnn>*
- ◇ *Error (math): Attempted log of negative number: <nnn>*
- ◇ *Error (math): Bad Logit(<nnn>)*
- ◇ *Error (math): Bad arg to: POWER(<nnn>, <nnn>)*
- ◇ *Error (math): Integer overflow in FACT*
- ◇ *Error (math): Integer overflow in COMBINATION*
- ◇ *Error (math): Integer overflow in PERMUTE*
- ◇ *Error (math): IBETA: arg is not $0 \leq t \leq 1$, = <nnn>*
- ◇ *Error (math): Bad random seed: <nnn>*

Error Messages from Symbol Table Routines

- ◇ *Error (sym table): Wrong type: can't assign <name> (<type>) to <name> (<type>).*

An attempt was made to assign incompatible variables.

◇ *Error (sym table): Variable of type <type> is too large*

There was not enough memory to allocate a variable.

References

- Abramowitz M and Stegun IA, eds. (1972) *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, 9th printing. New York: Dover.
- Agresti A (1990) *Categorical Data Analysis*. New York: John Wiley and Sons.
- Ahuja JC and Nash SW (1967) The generalized Gompertz-Verhulst family of distributions. *Sankhya series A* **29**:141-56.
- Akaike H (1973) Information theory and an extension of the maximum likelihood principle. In *Second International Symposium on Information Theory*, ed. B.N. Petrov and F. Csaki, pp. 268-281. Budapest: Hungarian Academy of Sciences. [Reprinted in Akaike (1992) and Akaike (1998)].
- Akaike H (1992) Information theory and an extension of the maximum likelihood principle. In *Breakthroughs in Statistics*, Volume III ed. S. Kotz and N. Johnson, pp. 610-624. New York: Springer Verlag.
- Akaike H (1998) *Selected Papers of Hirotugu Akaike*. New York: Springer Verlag.
- Bernoulli J (1713) *Ars Conjectandi*. Basel.
- Birnbaum ZW and Saunders SC (1969) A new family of life distributions. *Journal of Applied Probability* **6**:319-27.
- Borel E (1925) *Principes et formules classiques du Calcul des Probabilités*. Paris.
- Borwein P (1995) An efficient algorithm for the Riemann zeta function. Working paper. <http://www.cecm.sfu.ca/~pborwein/PAPERS/P155.pdf>, <http://citeseer.nj.nec.com/9477.html>.
- Box GE, Hunter WG, Hunter JS (1978) *Statistics for Experimenters*. New York: John Wiley & Sons.
- Bratley P, Fox BL, Schrage LE (1983) *A Guide to Simulation*. New York: Springer-Verlag.
- Brent RP (1973) Algorithms for minimization without derivatives. *Englewood Cliffs, NJ*: Prentice-Hall.
- Chew V (1968) Some alternatives to the normal distribution. *The American Statistician* **22**:22-4.
- Chhikara RA and Folks JL (1989) *The Inverse Gaussian Distribution*. New York: Marcel Dekker.
- Christensen R (1984) *Data Distributions*. Lincoln, MA: Entropy Ltd.
- Cullen AC, Frey HC (1999) *Probabilistic Techniques in Exposure Assessment: A Handbook for Dealing with Variability and Uncertainty in Model and Inputs*. New York: Plenum Press.
- Cox DR, Oakes D (1984) *Analysis of Survival Data*. London: Chapman and Hall.
- Daniels HE (1945) *Proc Royal Soc London, Series A* **183**:405-35.
- Deevey ES Jr. (1947) Life tables for natural populations of animals. *Quarterly Review of Biology* **22**:283-314.

- Dobson AJ (1990) *An Introduction to Generalized Linear Models*. Boca Raton: Chapman & Hall/CRC.
- Edwards AWF (1972) *Likelihood*. Cambridge: Cambridge University Press.
- Efron B (1982) *The Jackknife, the Bootstrap and Other Resampling Plans*. Philadelphia: Society for Industrial and Applied Mathematics.
- Eggenberger F, and Pólya G (1923) Über die Statistik verketteter Vorgänge. *Zeitschrift für Angewandte Mathematik und Mechanik* **1**:279-289.
- Elandt-Johnson RC, Johnson NL (1980) *Survival Models and Data Analysis*. New York: John Wiley and Sons.
- Evans M, Hastings N, Peacock B (2000) *Statistical Distributions*. Third edition. New York: John Wiley and Sons.
- Fisher RA, Corbet AS, Williams CB (1943) The relation between the number of species and the number of individuals in a random sample from an animal population. *Journal of Animal Ecology* **12**:42-58.
- Fisher RA (1921) On the 'probable error' of a coefficient of correlation deduced from a small sample. *Metron* **1**:3-32.
- Folks JL and Chhikara RS (1978) The inverse Gaussian distribution and its statistical applications—A review. *Journal of the Royal Statistical Society, Series B* **40**:263-89.
- Forsythe G, Malcolm MA, Moler CB (1977) *Computer Methods for Mathematical Computations*. Englewood Cliffs, NJ: Prentice-Hall.
- Gage TB (1989) Bio-mathematical approaches to the study of human variation in mortality. *Yearbook of Physical Anthropology* **32**:185-214.
- Gillespie D (1989) *p2c: Pascal to C translator*.
- Geoffe WL, Ferrier GD, Rogers J (1994) Global optimization of statistical functions with simulated annealing. *Journal of Econometrics* **60**:65-99.
- Gompertz B (1825) On the nature of the function expressive of the law of human mortality. *Philosophical Transactions of the Royal Society of London, Series A* **115**:513-85.
- Gumbel EJ (1947) The distribution of the range. *Annals Mathematical Statistics* **18**:384-412.
- Guttorp P (1995) *Stochastic Modeling of Scientific Data*. London: Chapman and Hall.
- Harris JW, Stocker H (1998) *Handbook of Mathematics and Computational Science*. New York: Springer-Verlag.
- Hammes LM, Treloar AE (1970) Gestational interval from vital records. *American Journal of Public Health* **60**:1496-505.
- Hazelrig JB, Turner ME, Blackstone EH (1982) Parametric survival analysis combining longitudinal and cross-sectional censored and interval-censored data with concomitant information. *Biometrics* **38**:1-15.
- Hilborn R and Mangel M (1997) *The Ecological Detective Confronting Models with Data*. Monographs in Population Biology 28. Princeton, N.J.: Princeton University Press.
- Holman DJ (1996) *Total Fecundability and Fetal Loss in Rural Bangladesh*. Doctoral Dissertation, The Pennsylvania State University.
- Holman DJ and Jones RE (1998) Longitudinal analysis of deciduous tooth emergence II: Parametric survival analysis in Bangladeshi, Guatemalan, Japanese and Javanese children. *American Journal of Physical Anthropology* **105**(2):209-30.
- Johnson NL, Kotz S (1969) *Discrete Distributions*. New York: John Wiley and Sons.

- Johnson NL, Kotz S, Balakrishnan N (1994) *Continuous Univariate Distributions*, (Volume 1, 2nd edition). New York: John Wiley and Sons.
- Johnson NL, Kotz S, Balakrishnan N (1995) *Continuous Univariate Distributions*, (Volume 2, 2nd edition). New York: John Wiley and Sons.
- Jørgensen B (1982) Statistical Properties of the Generalized Inverse Gaussian Distribution. *Lecture Notes in Statistics*, No. 9. New York: Springer-Verlag.
- Kalbfleisch JD, Prentice RL (1980) *The Statistical Analysis of Failure Time Data*. New York: John Wiley & Sons.
- King G (1998) *Unifying Political Methodology: The Likelihood Theory of Statistical Inference*. Ann Arbor: The University of Michigan Press.
- Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. *Science* **220** (4598):671-80.
- Kishor ST (1982) Probability and Statistics with Reliability, Queuing, and Computer Science Applications. Englewood Cliffs, NJ: Prentice-Hall.
- Laplace PS (1774) Mémoire sur la probabilité des causes par les évènements *Mém. de Math et Phys., l'Acad. Roy. des Sci. par div. Savans* **6**:621-56.
- Lee ET (1992) *Statistical Methods for Survival Data Analysis*. New York: John Wiley and Sons.
- Levy P (1939) *Composita Mathematica* **7**:283-339.
- Maxwell JC (1860a) *Phil Mag* **19**:19
- Maxwell JC (1860b) *Phil Mag* **20**:21, 33.
- Metropolis N, Rosenbluth A, Rosenbluth M, Teller A, and Teller E (1953) Equation of state calculations by fast computing machines. *Journal of Chem. Phys.* **21**:1087-90.
- Mohr PJ and Taylor BN (1999) CODATA Recommended values of the fundamental physical constants:1998. *Journal of Physical and Chemical Reference Data* **28**(6):1-140.
- Morgan BJT (2000) *Applied Stochastic Modeling*. London:Arnold.
- Murie A (1944) *The Wolves of Mount McKinley*. (Fauna of the National Parks of the U.S.. Fauna Series No. 5 238 pp.) U.S. Dept. Int., National Park Service. Washington.
- Nelson W (1982) *Applied Life Data Analysis*. New York: John Wiley and Sons.
- Nelder JA, and Mead R (1965) A simplex method for function minimization. *Computer Journal* **7**:308-13.
- Pearson K (1895) *Phil. Trans. Roy. Soc. London, Series A* **186**:343-414.
- Pearson K (1900) *Phil Mag and J Sci*, 5th Series. **50**:157-75.
- Pickles A (1985) *An Introduction to Likelihood Analysis*. Norwich: Geobooks.
- Powell MJD (1964) An efficient method for finding the minimum of a function of several variables without calculating derivatives. *Comp. Journal* **7**:155-62.
- Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1989) *Numerical Recipes in Pascal: The Art of Scientific Programming*. Cambridge: Cambridge University Press.
- Rao CR (1973) *Linear Statistical Inference and Its Applications*. New York: John Wiley and Sons.
- Ridders CJF (1982) *Advances in Engineering Software* **4**(2):75-6.
- Salvia AA (1985) Reliability applications of the alpha distribution. *IEEE Transactions on Reliability* **34**:251-2.
- SAS Institute (1985) *SAS User's Guide: Statistics*. Version 5 edition. Cary, NC: SAS Institute, Inc.

- Schrödinger E (1915) Zür Theorie der Fall—und Steigversuche an Teilchenn mit Brownsche Bewegung. *Physikalische Zeitschrift* **16**:289-95.
- Shah BK, Dave PH (1963) A note on log-logistic distribution, *Journal of the M.S. University of Baroda (Science Number)* **12**:15-20.
- Subbotin MT (1923) On the law of frequency of errors. *Mathematicheskii Sbornik* **31**:296-301.
- Tanner MA (1996) *Tools for Statistical Inference: Methods for the Exploration of Posterior Distributions and Likelihood Functions*, 3rd edition. New York: Springer-Verlag.
- Taylor BN (1995) *Guide for the Use of the International System of Units (SI)*. National Institute of Standards and Technology, special publication 811, 1995 edition. Washington: US Government Printing Office.
- Thomas M (1949) A generalizaton of poisson's binomial limit for use in ecology *Biometrika* **36**:18-25.
- Tuma NB, Hannan MT (1984) *Social Dynamics: Models and Methods*. New York: Academic Press.
- Tweedie MCK (1947) Functions of a statistical variate with given means, with special reference to Laplacian distributions. *Proceedings of the Cambridge Philosophical Society* **43**:41-9
- Van Canneyt M (2000) *Free Pascal Programmers' Manual*. (for FPC version 1.0.2) version 1.8.
- Vaupel JW (1990) Relatives' risks: Frailty models of life history data. *Theoretical Population Biology* **37**:220-34.
- Vaupel JW, Yashin AI (1985) Heterogeneity's ruses: Some surprising effects of selection on population dynamics. *American Statistician* **39**:176-85.
- Wald A (1947) *Sequential Analysis* New York:John Wiley & Sons.
- Wise ME (1966) Tracer-dilution curves in cardiology and random walk and lognormal distributions. *Acta Physiologica Pharmacologica Neerlandica* **14**:175-204.
- Wood JW (1989) Fecundity and natural fertility in humans. *Oxford Reviews of Reproductive Biology* **11**:61-109.
- Wood JW (1994) *Dynamics of Human Reproduction: Biology, Biometry, Demography*. Hawthorne, NY: Aldine de Gruyter.
- Wood JW, Holman DJ, O'Connor KA and Ferrell RE. (2001) Models of human mortality. In Hoppa R and Vaupel J (eds) *Paleodemography: Age Distributions from Skeletal Samples*. Cambridge: Cambridge University Press.
- Wood JW, Holman DJ, Yasin A, Peterson RJ, Weinstein M, Chang M-c (1994) A multistate model of fecundability and sterility. *Demography* **31**:403-26.
- Wood JW, Holman DJ, Weiss KM, Buchanan AV, LeFor B (1992) Hazards models for human biology. *Yearbook of Physical Anthropology* **35**:43-87.
- Zipf GK (1949) *Human Behavior and the Principle of Least Effort*. Reading: Addison Wesley.

