Qlib: Technical documentation

jacob navia

Contents

1 Foreword

This is a public release of my version of the CEPHES Mathematical Library. This work is based on the work of Stephen L Moshier, the author of the CEPHES Mathematical Library.

The copyright notice of that library is as follows:

```
   Some software in this archive may be from the book _Methods and
Programs for Mathematical Functions_ (Prentice-Hall or Simon & Schuster
International, 1989) or from the Cephes Mathematical Library, a
commercial product. In either event, it is copyrighted by the author.
What you see here may be used freely but it comes with no support or
guarantee.

The two known misprints in the book are repaired here in the
source listings for the gamma function and the incomplete beta
integral.


Stephen L. Moshier
moshier@na-net.ornl.gov
```

Since the author allowed me to use his work freely, I have written a new version of it. The main differences from the original code are:

- It is a 64 bit library. The last version that I know of it, was a 32 bit library with 144 bits. This version is a 64 bit one, using 448 bits.
- The four operations have been rewritten in pure assembler, making this version very fast. There are two versions of them: One in 64 bit x86-64 assembler, and the other in 64 bit ARM-64 assembler, what allows it to run in the raspberry pi and the Apple Macintosh (with Apple's CPUs).
- The precision has been improved in many functions, and a new set of constants have been added with 448 bit precision.
- The documentation has been rewritten in the LATEX system.

This software is very old. I have found a comment in studt.c like this:

```
/*  STUDNT.C    24 NOV 83

C       STUDNT.FOR      LATEST REV: 31 AUG 77
C               SLM, 31 AUG 77
C
C   EVALUTATES INTEGRAL OF STUDENT'S T DISTRIBUTION FROM
C   MINUS INFINITY TO T
C
```

```
C       USAGE:
C   CALL STUDNT(K,T,P)
C
C   K = INTEGER NUMBER OF DEGREES OF FREEDOM
C   T = RANDOM VARIABLE ARGUMENT
C   P = OUTPUT AREA
C
C   THE DENSITY FUNCTION IS
C   A*Z**-(K+2)/2,
C   WHERE Z = 1 + (T**2)/K
C   AND A = GAMMA((K+1)/2)/(GAMMA(K/2) * SQRT(K*PI)).
C   THE INTEGRAL IS EVALUATED IN CLOSED FORM BY INTEGRATION BY
C   PARTS.  THE RESULT IS EXACT, TO WITHIN ROUNDOFF ERROR.
C
C   SUBROUTINE LGAM, LOG OF GAMMA FUNCTION, IS NEEDED.
*/
```

Mr Moshier has been working in this library since 1977. And it is a testament to the longevity of the software written in the C language that more than 40 years later it continues to run like new. C is not "the new language of the day", it is a tried and tested language where you can build software that will run for decades.

Almost all of these functions were written by Stephen L. Moshier. These files have a copyright notice that begins in 1984 or 1985.

```
* Cephes Math Library Release 2.3:  March, 1995
* Copyright 1985, 1995 by Stephen L. Moshier
```

I have revised the algorithms of some functions, added some (catalan, AGM, for instance) reformatted the code, but essentially this work is based on SLM's work.

```
/*
* Revision history:
*
* SLM,  5 Jan 84   PDP-11 assembly language version
* SLM,  2 Mar 86   fixed bug in asctoq()
* SLM,  6 Dec 86   C language version
* JN    1995-2011  Many improvements, modifications, etc
* JN    2017       Ported to ARM64
* JN    2021       Ported to Apple's M1
*/
```

I have tested this version in the following machines:

1. A PC with windows (16GB RAM, AMD Ryzen Windows 10)
2. A PC with linux (16GB RAM, Rtzen, Linux Mint)
3. A Macintosh (mac-mini Apple M1, 16GB RAM, Mac OS X)
4. A Macintosh (mac-pro Apple x86_64 32GB RAM Mac OS X)
5. A Raspberry pi (8GB RAM and ARM64 CPU, Linux)
6. A Rock64 single board computer from Pine64.org (ARM 64 processor 4GB RAM Linux Debian)

2 Data types

This library features 8 bytes numbers, with a mantissa of seven 64 bits numbers and a header of 64 bits where the sign and the exponent are stored. This is a design for maximal speed, where space considerations are mostly ignored. The four operations are written entirely in ARM64/x86_64 assembler. This allows to increase speed by a factor of 30 relative to a high level language like C. Built upon this assembler core functionality the module uses an adapted version of the CEPHES mathematical library written by Stephen L. Moshier. All higher order functions like logarithm, square root , exponential function etc, are written in C using the core functions.

2.1 Data types

Numbers are represented using the following structure:

```
typedef struct tagQfloat {
    int sign;                               // 32 bits sign...
    unsigned int exponent;                  // 32 bits exponent
    unsigned long long mantissa[MANTISSA_LENGTH]; // 7 * 64 = 448 bits mantissa
} Qfloat;
```

This differs completely of Mr Moshier representation as an undifferentiated table of integers. Access to exponent and sign is clearer. All the library has been rewritten to use this structure.

One problem in this transformation was that since numbers were a table of integers, they were always passed by reference, since tables are always passed as a reference to the first member in C. To keep this and avoid inefficiencies when passing numbers by value, all local variables and numbers are declared as tables of one element: instead of `Qfloat x;` all the functions in the library use `Qfloat x[1];` what is actually exactly the same, but since the second declaration will be understood as a table, it will be passed by reference in all calls.

All constants were recalculated for 64 bits and correctly rounded, and the precision in several functions was extended.

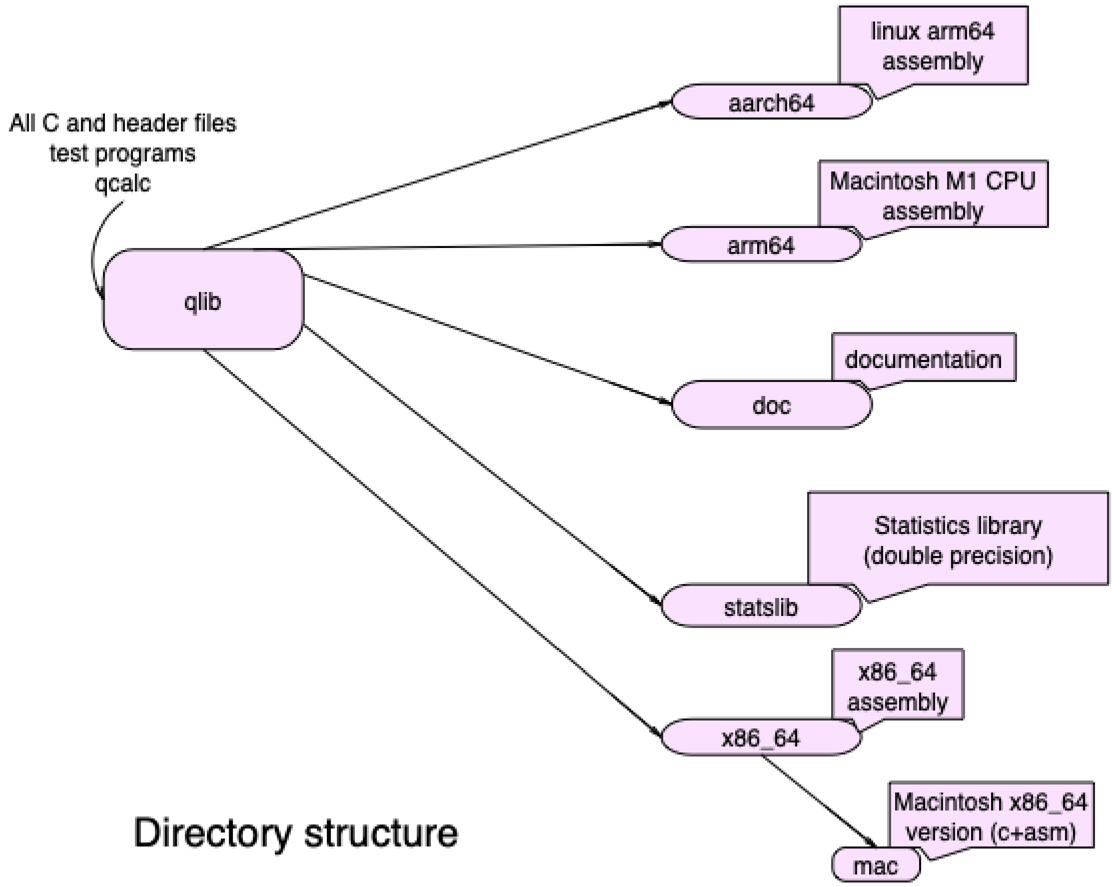
2.2 Implementations

Two assembler implementations exist: one for the x86_64 and another for the ARM64 architectures. They are implemented essentially in the `qasm-xxx` modules. There 3 of them: the `aarch64` module for linux with ARM64, `arm64` for Apple under ARM64, and x86_64 for linux under x86.

2.3 Software organization

Each function is implemented in his own file. For instance qsqrt.c implements the square root, etc. There are several sub-directories:

- `doc` Documentation in Latex
- `aarch64` Assembly programs for the linux aarch64 architecture (raspberry pi, rock64).
- `arm64` Apple's M1 series assembly programs.
- `x86-64` PC assembly programs
- `statslib` Statistics library in double precision



3 The functions

Table 3.1: Qlib documentation

File	Function description
agm.c	This file implements the arithmetic geometric mean, written by `Tom Van Baak (tvb) www.LeapSecond.com/tools` and used to test the implementation in qfloat precision.
cmplx.c	This is the complex arithmetic module in double precision, used to calculate starting approximations for some functions.

qacosh.c	**3.0 Inverse hyperbolic cosine (qacosh)**
	SYNOPSIS:
	`int qacosh(x, y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	$$acosh(x) = log\left(x + \sqrt{(x-1)\times(x+1)}\right)$$

qagm.c	**3.1 Arithmetic Geometric Mean of two numbers (qagm)**
	SYNOPSIS:
	`void qagm(x, y, r);`
	`qfloat *a; // input`
	`qfloat *g; // input`
	`qfloat *r; // output`
	x and y is:
	$x_0 = x$, $y_0 = y$
	$x_{n+1} = \frac{1}{2}(x_n + y_n)$
	$y_{n+1} = \sqrt{x_n - y_n}$

qairy.c	**3.2 Airy functions**
	SYNOPSIS:
	`int qairy( x, ai, aip, bi, bip );`
	`qfloat *x; // input`
	`qfloat *ai, *aip; // output`
	`qfloat *bi, *bip; // output`
	Solution of the differential equation
	$y''(x) = xy$.
	The function returns the two independent solutions Ai, Bi and their first derivatives Ai'(x), Bi'(x).
	Evaluation is by power series summation for small x, by asymptotic expansion for large x.
	ACCURACY:
	The asymptotic expansion is truncated at less than full working precision (only 105 digits).

Table 3.1: Qlib documentation

<table>
<tr><th>File</th><th>Function description</th></tr>
<tr><td>qasin.c</td><td>

3.3 Inverse sine (qasin)

SYNOPSIS:

```
int qasin( x, y ); int qasin( x, y );
qfloat *x; // input
qfloat *y; // output
```

This file implements `asin(x)` and `acos(x)`.
`qasin` returns radian angle between -pi/2 and +pi/2 whose sine is x.

$$asin(x) = arctan\left(\frac{1}{\sqrt{1-x^2}}\right)$$

If $|x| > 0.5$ it is transformed by the identity

$$asin(x) = \frac{\pi}{2} - 2 \times asin\left(\sqrt{\frac{(1-x)}{2}}\right)$$

`qacos`:

$$acos(x) = \frac{\pi}{2} - asin(x)$$

</td></tr>
<tr><td>qasinh.c</td><td>

3.4 Inverse Hyperbolic sine (qasinh)

SYNOPSIS:

```
int qasinh( x, y );
qfloat *x; // input
qfloat *y; // output
```

$$asinh(x) = log\left(x + \sqrt{1+x^2}\right)$$

For very large x, $asinh(x) = log(x) + log(2)$

</td></tr>
<tr><td>qatanh.c</td><td>

3.5 Inverse hyperbolic tangent (qatanh)

SYNOPSIS:

```
int qatanh( x, y );
qfloat *x; // input
qfloat *y; // output
```

Returns inverse hyperbolic tangent of argument.

$$atanh(x) = 0.5 \times log\left(\frac{(1+x)}{(1-x)}\right)$$

For very small x, the first few terms of the Taylor series are summed.

</td></tr>
<tr><td>qatn.c</td><td>

3.6 Inverse circular tangent qatn

SYNOPSIS:

```
int qatn( x, y );
qfloat *x; // input
qfloat *y; // output
```

Returns radian angle between $\frac{-\pi}{2}$ and $\frac{+\pi}{2}$ whose tangent is x.
Range reduction is from three intervals into the interval from zero to $\frac{\pi}{8}$.

$$arctan(x) = \frac{x}{1 - \frac{x^2}{3 - \frac{4x^2}{5 - \frac{9x^2}{7 -}}}} \ldots$$

</td></tr>
</table>

Table 3.1: Qlib documentation

File	Function description

qbeta.c	**3.7 Beta function (qbeta)**
	SYNOPSIS:
	`int qbeta( a, b, y );`
	`qfloat *a, *b; // inputs`
	`qfloat *y; // output`
	$$beta(a,b) = \frac{\Gamma(a) \times \Gamma(b)}{\Gamma(a+b)}$$

qcalc.c	This is a calculator that features all functions of the package in an interactive way.

qcatalan.c	**3.8 Catalan (qcatalan)**
	SYNOPSIS:
	`void qcatalan(n,result);`
	`qfloat *n; // input`
	`qfloat *result; // output`
	This returns the n_{th} catalan number
	$$C_n = \frac{(2n)!}{(n+1)! \times n!}$$
	These numbers will be calculated using 128 bit integers up to $n = 63$. For numbers above qfloat precision is used.

qcbrt.c	**3.9 Cubic root (qcbrt)**
	SYNOPSIS:
	`int qcbrt( x, y );`
	`qfloat *x; // input`
	`qfloat *y; // output`
	This calculates the cubic root of a number that can be negative. A first approximation is calculated in double precision, then the newton method is used for getting full precision.

Table 3.1: Qlib documentation

<table>
<tr><th>File</th><th>Function description</th></tr>
<tr><td>qconst.c</td><td>

3.10 Constants

A set of mathematical constants (π, e, $log(2)$, and several small numbers and fractions, all of them calculated to 132 digits precision. The constants were verified using the GP PARI calculator of Bordeaux's University. By a lucky coincidence the hexadecimal representation of floating point numbers is identical to qlib's. I have rounded all constant to 448 bits by using 140 decimal places in GP, what allowed me to see the next bits and round accordingly.

<table>
<tr><td>qminusone</td><td>-1</td><td>qzero</td><td>0</td><td>qhalf</td><td>$\frac{1}{2}$</td></tr>
<tr><td>qone</td><td>1</td><td>qtwo</td><td>2</td><td>qthree</td><td>3</td></tr>
<tr><td>qfive</td><td>5</td><td>qnine</td><td>9</td><td>q32</td><td>32</td></tr>
<tr><td>oneThird</td><td>$\frac{1}{3}$</td><td>qlog2</td><td>$log(2)$</td><td>qinv_log2</td><td>$\frac{1}{log(2)}$</td></tr>
<tr><td>qsqrt2</td><td>$\sqrt{2}$</td><td>qinv_sqrt2</td><td>$\frac{1}{\sqrt{2}}$</td><td>oneopi</td><td>$\frac{1}{\pi}$</td></tr>
<tr><td>qpi</td><td>π</td><td>qPi_Div_2</td><td>$\frac{\pi}{2}$</td><td>qinv_pi</td><td>$\frac{1}{\pi}$</td></tr>
<tr><td>qeul [1]</td><td>γ</td><td>qlog10c</td><td>$log(10)$</td><td>qinv_log10</td><td>$\frac{1}{log10}$</td></tr>
<tr><td>qmem1 [2]</td><td>$-e^{-1}$</td><td>qexp</td><td>e</td><td>invSqrt2pi</td><td>$\frac{1}{\sqrt{\pi}}$</td></tr>
<tr><td>qepsilon</td><td>2^{-448}</td><td></td><td></td><td></td><td></td></tr>
</table>

</td></tr>
<tr><td>qcos.c</td><td>

3.11 Cosinus (qcos)

SYNOPSIS:

<pre><code>int qfcos(x, y);
qfloat *x; // input
qfloat *y; // output</code></pre>

The cosinus is just

$$\cos(x) = \sin\left(\frac{\pi}{2} - x\right)$$

In this file the function $cos(x) - 1$ (`qcosm1`) is implemented using the Taylor series, useful for small x.

</td></tr>
<tr><td>qcosh.c</td><td>

3.12 Hyperbolic cosine (qcosh)

SYNOPSIS:

<pre><code>int qcosh(x, y);
qfloat *x; // input
qfloat *y; // output</code></pre>

$$cosh(x) = \frac{exp(x) + exp(-x)}{2}$$

If the number is smaller than 16, the Taylor series is used instead of the simple procedure proposed in the original code:

$$cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} \ldots$$

</td></tr>
</table>

[1]This is the Euler-Mascheroni constant: 0,5772156649...

[2]This is used in Lambert's W function

Table 3.1: Qlib documentation

File	Function description

qei.c	**3.13 The exponential integral (qei)**
	SYNOPSIS:
	`qei( x, y );`
	`qfloat *x; // input`
	`qfloat *y; // output`
	$$Ei(x) = \int_{-\infty}^{x} \frac{e^t}{t} dt$$
	For values smaller than 32 this integral will be approximated by:
	$$Ei(x) = \delta + ln(x) + \sum_{n=1}^{\infty} \frac{x^n}{n \times n!}$$
	where δ is the Euler–Mascheroni constant (0.577215664901...).
	For values $>$ 32 SLM used the asymptotic expansion:
	$$x \times e^x \times Ei(x) = 1 + \frac{2}{x^2} + \frac{6}{x^3} + ...\frac{n!}{x^n}$$

qellie.c	**3.14 Incomplete elliptic integral of the second kind (qellie)**
	SYNOPSIS:
	`int qellie( phi, m, y );`
	`qfloat *phi, *m; // inputs`
	`qfloat *y;       // output`
	Approximates the integral:
	$$E(\phi, m) = \int_{0}^{\phi} \sqrt{1 - m \times sin^2\, t}\, dt$$
	of amplitude ϕ and modulus m, using the arithmetic geometric mean algorithm.

qellik.c	**3.15 Incomplete elliptic integral of the first kind (qellik)**
	SYNOPSIS:
	`int qellik( phi, m, y );`
	`qfloat*phi, *m; // inputs`
	`qfloat *y;      // output`
	Approximates the integral:
	$$F(\phi, m) = \int_{0}^{\phi} \frac{dt}{\sqrt{1 - m \times sin^2\, t}}$$

Table 3.1: Qlib documentation

File	Function description

qellpe.c	**3.16 Complete elliptic integral of the second kind (qellpe)**
	SYNOPSIS:
	`int qellpe(x, y);`
	`qfloat *x; //input`
	`qfloat *y; //output`
	Approximates the integral
	$$E(m) = \int_0^{\frac{\pi}{2}} \sqrt{1 - m\ sin^2 t}\ dt$$

qellpj.c	**3.17 Jacobian Elliptic Functions (qellpi)**
	SYNOPSIS:
	`int qellpj( u, m, sn, cn, dn, ph );`
	`qfloat *u, *m;           // inputs`
	`qfloat *sn, *cn, *dn, *ph; // outputs`
	Evaluates the Jacobian elliptic functions $sn(u\|m)$, $cn(u\|m)$, and $dn(u\|m)$ of parameter m between 0 and 1, and real argument u.
	These functions are periodic, with quarter-period on the real axis equal to the complete elliptic integral $ellpk(1.0 - m)$.
	Relation to incomplete elliptic integral: If $u = ellik(\phi, m)$, then $sn(u\|m) = \sin(\phi)$, and $cn(u\|m) = \cos(\phi)$. ϕ is called the amplitude of u.
	Computation is by means of the arithmetic-geometric mean algorithm, except when m is within 1e-9 of 0 or 1. In the latter case with m close to 1, the approximation applies only for $\phi < \pi/2$.
	ACCURACY: Truncated at 70 bits.

qellpk.c	**3.18 Complete elliptic integral of the first kind (qellpk)**
	SYNOPSIS:
	`int qellpk(x,y);`
	`qfloat *x; // input`
	`qfloat *y; //output`
	This approximates the integral:
	$$K(m) = \int_0^{\frac{\pi}{2}} \frac{dt}{\sqrt{1 - m \times sin^2 t}}$$
	where $m = 1 - m1$, using the arithmetic-geometric mean method.
	The argument $m1$ is used rather than m so that the logarithmic singularity at $m = 1$ will be shifted to the origin; this preserves maximum accuracy.
	$K(0) = \pi/2$.
	ACCURACY: Truncated at NBITS

qerf.c	**3.19 Error function (qerf)**
	SYNOPSIS:
	`int qerf(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	Calculates the error function.
	$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x exp(-t^2)\ dt$$

Table 3.1: Qlib documentation

File	Function description

qerfc.c	**3.20 Complementary error function**
	SYNOPSIS:
	`int qerfc(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	This calculates:
	$$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} exp(-t^2)\, dt = 1 - erf(x)$$
	using two different continued fraction for arguments smaller than 4 and bigger than 4.

qexp.c	**3.21 The exponential function (qefexp)**
	SYNOPSIS:
	`int qfexp(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	The exponential function (e^z)is implemented using the series
	$$e^z = \sum_{n=0}^{\infty} \frac{z^n}{n!} = 1 + \frac{z}{1} + \frac{z^2}{2!} + \frac{z^3}{3!} \cdots$$
	To achieve full precision this needs approx 30-40 iterations, depending on the argument. Arguments close to 1 need more iterations. This method is different from the one used by Mr Moshier:
	$$exp(x) = \frac{1 + tanh(x)}{1 - tanh(x)}$$
	The speed of the calculation was improved by 60%. On the downside a loss of 1 ulps has been seen due to the long summation of the pre-calculated inverse factorials table.

qexp10.c	**3.22 Base 10 exponential function (qexp10)**
	SYNOPSIS:
	`int qexp10(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	(Common antilogarithm).
	$$10^x = e^{x \times log(10)}$$

qexp2.c	**3.23 Base 2 exponential function (qexp2)**
	SYNOPSIS:
	`int qexp2(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	Base 2 exponential function
	$$2^(x) = e^{x \times log(2)}$$

Table 3.1: Qlib documentation

File	Function description

qexpm1.c	$e^x - 1$ **3.24 expm1 (qexpm1)**
	SYNOPSIS:
	`int qexpm1(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	Returns e (2.71828...) raised to the x power, minus 1.
	If x is nearly zero, then the common expression exp(x) - 1.0 will suffer from catastrophic cancellation and the result will have little or no precision. The `expm1` function provides an alternative means to do this calculation without the risk of significant loss of precision.

qexpn.c	**3.25 Exponential integral (qexpn)**
	SYNOPSIS:
	`int qexpn(n,x,y);`
	`qfloat *n; // input`
	`qfloat *x; // input`
	`qfloat *y; // output`
	Evaluates the exponential integral:
	$$E_n(x) = \int_1^{\infty} \frac{e^{-xt}}{t^n}\, dt$$

qfac.c	**3.26 Factorial (qfact)**
	SYNOPSIS:
	`int qfact(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	Calculates $n!$.

qfloor.c	**3.27 Floor (qfloor)**
	SYNOPSIS:
	`int qfloor(x,y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	Computes the largest integer not greater than x.

qfltbi.c	**3.28 The qfltbi file**
	This file contains all the low level routines that were rewritten in assembler. The main routines are:
	`addm( x, y )      add significand of x to that of y`
	`shdn1( x )        shift significand of x down 1 bit`
	`shdn8( x )        shift significand of x down 8 bits`
	`shdn16( x )       shift significand of x down 16 bits`
	`shup1( x )        shift significand of x up 1 bit`
	`shup8( x )        shift significand of x up 8 bits`
	`shup16( x )       shift significand of x up 16 bits`
	`divm( a, b )      divide significand of a into b`
	`mulm( a, b )      multiply significands, result in b`
	`mdnorm( x )       normalize and round off`
	None of this routines are directly usable. They are part of the library and are used through their higher level ones in `qflti.c`.

Table 3.1: Qlib documentation

File	Function description

qflti.c	**3.29 Utilities**
	qfloat precision utilities.
	`asctoq( string, q) ascii string to q type`
	`etoq( d, q )      IEEE double precision to q type`
	`e24toq( d, q )    IEEE single precision to q type`
	`itoq( &l, q )     long integer to q type`
	`qabs(q)           absolute value`
	`qadd( a, b, c )   c = b + a`
	`qclear(q)         q = 0`
	`qcmp( a, b )      compare a to b`
	`qdiv( a, b, c )   c = b / a`
	`qifrac(x,&l,frac) x to integer part l and q type fraction`
	`qfrexp( x, l, y ) find exponent l and fraction y between .5`
	`    and 1`
	`qldexp( x, l, y ) multiply x by 2^l`
	`qinfin( x )       set x to infinity, leaving its sign alone`
	`qmov( a, b )      b = a`
	`qmul( a, b, c )   c = b * a`
	`qmuli( a, b, c )  c = b * a, a has only 16 significant bits`
	`qisneg(q)         returns sign of q`
	`qneg(q)           q = -q`
	`qnrmlz(q)         adjust exponent and mantissa`
	`qsub( a, b, c )   c = b - a`
	`qtoasc( a, s, n ) q to ASCII string, n digits after decimal`
	`double qtoe(q,doround) convert q type to IEEE double precision`
	`qtoe24( q, &d )   convert q type to IEEE single precision`
	`qinv(src,result)  Inverse: result = 1/src.`

	Conversions
qflti.c	**3.30 Convert text to qfloat (asctoq)** SYNOPSIS: `int asctoq(const char *text, qfloat *output, char **pend);` The character string `text` will be converted into qfloat, and the `pend` pointer will point to the character right after the last digit. Scientific notation is supported.
qflti.c	**3.31 Convert qfloat to text (qtoasc)** SYNOPSIS: `int qtoasc(Qfloatp q,char *string,int width,int ndigs,int flags );` Formats the given number using the given width and number of digits after the decimal point. Numbers are rounded to the given precision.

qflti.c	**3.32 Conversion of double, float to qfloat (e2q e24toq)**
	SYNOPSIS:
	`int etoq(double a,qfloat *b);`
	`int e24toq(float d,qfloat *b);`
	`etoq` converts the double precision number a into a qfloat. `e24toq` converts a single precision one into a qfloat.

Table 3.1: Qlib documentation

File	Function description
qflti.c	**3.33 Conversion of qfloat to double, float, and long double** SYNOPSIS: `double qtoe(Qfloatp x,int roundflag);` `qtoe` converts the qfloat number x into a double precision one. If `roundflag` is not zero, rounding will be performed, otherwise the number is truncated.
qfltbi.c	**3.34 64 bit integer to qfloat** SYNOPSIS: `void lltoq(long long a,qfloat *b);` This function is written in assembler.
qflti.c	**3.35 Compare numbers** SYNOPSIS: `int qcmp(qfloat *a, qfloat*b);` Returns 1 if $a > b$, zero if $a = b$ or -1 if $a < b$.
qflti.c	**3.36 Integer part and fraction** SYNOPSIS: `void qifrac(Qfloatp x,long long *i,Qfloatp frac);` `qifrac` will write into the location pointed by i the integer part of x and in $frac$ the fractional part. If the integer part doesn't fit into a 64 bit integer the result is `0x7fffffffffffffff`.
	The four operations
qflti.c	**3.37 Addition** SYNOPSIS: `int qadd(qfloat *const a,qfloat *const b,qfloat *c);` $c = b + a$. Returns 1 if the operation was done, zero if the exponent difference between the two numbers was beyond accuracy, -1 if an underflow occurred, and -2 if overflow was detected. In case of underflow the result (c) is zero, in case of overflow the result is the biggest possible number. In case the result would be beyond accuracy, the larger number is copied to the result. This procedure is written in assembly language.
qfltbi.c	**3.38 Subtraction** SYNOPSIS: `int qsub(qfloat *const a,qfloat *const b,qfloat *c);` $c = b - a$ Returns the same integer codes as `qadd`. This procedure is written in assembly language.
qflbi.c	**3.39 Multiplication** SYNOPSIS: `void qmul(qfloat *const a,qfloat *const b,qfloat *c);` $c = b \times a$. If an overflow occurs, it returns the biggest possible number in c. This procedure is written in assembly language.
qflbi.c	**3.40 Multiplication by a 64 bit integer** SYNOPSIS: `void qmuli(long long a,qfloat *const b,qfloat *c);` $c = b \times a$. If an overflow occurs, it returns the biggest possible number in c. This procedure is written in assembly language.

Table 3.1: Qlib documentation

File	Function description
qfltbi.c	**3.41 Division** SYNOPSIS: `int qdiv(qfloat *const a,qfloat *const b,qfloat *c);` $c = b/a.$ This procedure is written in assembly language.
qfltbi.c	**3.42 Assignment** SYNOPSIS: `int qmov(qfloat *const a,qfloat *b);` $b = a.$ This procedure is written in assembly language.
qflti.c	**3.43 Inverse** SYNOPSIS: `int qinv(qfloat *const x,qfloat *y);` $$y = \frac{1}{x}$$

qfresf.c	**3.44 Fresnel integral**
	SYNOPSIS:
	`int qfresnl( x, s, c );` `int qfresng(x,f,g)`
	`qfloat *x; /* input */` `qfloat *x; // input`
	`qfloat *s; /* output */` `qfloat *f; // output`
	`qfloat *c; /* output */` `qfloat *g; // output`
	Evaluates the Fresnel integrals:
	$$C(x) = \int_0^x cos\left(\pi/2 \times t^2\right)\ dt$$
	$$S(x) = \int_0^x sin\left(\pi/2 \times t^2\right)\ dt$$
	The integrals are evaluated by a power series for x < 1. For large x auxiliary functions $f(x)$ and $g(x)$ are employed such that:
	$$C(x) = 0.5 + f(x) \times sin\left(\frac{\pi}{2}\,x^2\right) - g(x)\,cos\left(\frac{\pi}{2}\,x^2\right)$$
	$$S(x) = 0.5 - f(x) \times cos\left(\frac{\pi}{2}\,x^2\right) - g(x)\,sin\left(\frac{\pi}{2}\,x^2\right)$$
	Routine `qfresfg` computes f and g.
	ACCURACY: Series expansions are truncated at less than full working precision.

3.45 ldexp and frexp
qfrexp.c
	SYNOPSIS:
	`void qldexp(x,n, y);` `void qfrexp(x,n,y)`
	`qfloat *x; /* input */` `qfloat *x; // input`
	`long n;    /* input */` `int *n;    // output`
	`qfloat *y; /* output */` `qfloat *y; // output`
	The `qldexp` function multiplies x by 2^n The `qfrexp` function breaks the input x into a normalized fraction and an integral power of 2.

Table 3.1: Qlib documentation

<table>
<tr><th>File</th><th>Function description</th></tr>
<tr><td>qgamma.c</td><td>

3.46 log of Gamma function

SYNOPSIS:

```
int qlgam(x,y);               int qgamma(x,y)
qfloat *x; /* input */        qfloat *x; // input
qfloat *y; /* output */       qfloat *y; // output
```

The function `qlgam` calculates the natural logarithm of gamma function. The `qgamma` function returns the value of the gamma function.

</td></tr>
<tr><td>qflti.c</td><td>

3.47 Hexadecimal representation

SYNOPSIS:

```
int qhex(Qfloatp const number,size_t outbufLength,char *outbuf);
qhypot.c& \TOC{Hypotenuse}
```

Representation of a number in hexadecimal. This produces a character string containing 3 parts:

- Sign. A single character containing either `'+'` or `'-'`.
- `0.` In a normalized number the first bit is always 1. This leads that in some representations this bit is implicit, hence this `'1.'`. In the `qfloat` representation, with 448 bits, this isn't done actually, so the traditional '1' is replaced by a '0'.
- The 448 bits of the mantissa will be shown as 112 hexadecimal numbers.
- The letter `'p'` that introduces the exponent
- The exponent (using base 10). This number is the number of times you have to multiply the mantissa by 2 to arrive at the number's value.
- If the length of the output buffer is less than the length required by the full format, the mantissa will be truncated accordingly.
- if the length of the buffer is less than 15, the field will be filled with the `'*'` character.

EXAMPLE:

```
qhex(pi,128,buf);
+0x0.c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b2\
     2514a08798e3404ddef9519b3cd3a431b302b0a6df25f1437p1
qhex(pi,30,buf);
+0x0.c90fdaa22168c234c4c6628p1
qhex(pi,10,buf);
**********
```

</td></tr>
<tr><td>qhy2f1.c</td><td>

3.48 Hypergeometric $_2F_1$

SYNOPSIS:

```
int qhy2f1( a, b, c, x, y );
qfloat *a,*b,*c,*x; // input
qfloat *y; // output
```

$$hy_2f_1 = 1 + \sum_{k=0}^{\infty} \frac{a(a+1)(a+2)\ldots(a+k)\, b(b+1)(b+2)\ldots(b+k)}{c(c+1)(c+2)\ldots(c+k) \times (k+1)!}\; x^{k+1}$$

</td></tr>
</table>

Table 3.1: Qlib documentation

File	Function description
qhypot.c	**3.49 Hypotenuse** SYNOPSIS: `void qhypot(qfloat *const x, qfloat *const y, qfloat *z);` Calculates $\sqrt{x^2+y^2}$ without overflow [3] using the following algorithm: • $Max \leftarrow Maximum(\|x\|,\|y\|)$ • $Min \leftarrow Minimum(\|x\|,\|y\|)$ • $r = \frac{Min}{Max}$ • return $Max \times \sqrt{1 + r \times r}$ The term under the square root is always between 1 and 2.

qhyperg.c	**3.50 Hypergeometric function**
	SYNOPSIS:
	`int qhyp( a, b, x, y );`
	`qfloat *a,*b,*x; //inputs`
	`qfloat *y; //output`
	Confluent hypergeometric function.
	$${}_1F_1(a,b;x) = 1 + \frac{a\,x^1}{b\;1!} + \frac{a\,x^2}{b(b+1)\;2!} + \frac{a\,x^3}{b(b+1)(b+2)\;3!} + \ldots$$

qigam.c	**3.51 The incomplete gamma integral**
	SYNOPSIS:
	`int qigam( a, x, y );`
	`qfloat *a,*x; // inputs`
	`qfloat *x; // output`
	$$igam(a,x) = \frac{1}{\Gamma(a)} \int_0^x e^t\, t^{a-1}\, dt$$
	In this implementation both arguments must be positive. The integral is evaluated by either a power series or continued fraction expansion, depending on the relative values of a and x.
	ACCURACY: Expansions terminate at less than full working precision.

qigami.c	**3.52 Inverse of complemented incomplete gamma integral**
	SYNOPSIS:
	`int qigami( a, p, x );`
	`qfloat *a,p; // inputs`
	`qfloat *x; // output`
	Refines an initial estimate generated by the double precision routine `igami` to find the root of:
	$igamc(a,x) - p = 0.$
	ACCURACY: Set to do just one Newton-Raphson iteration.

[3] This will be difficult to achieve since the biggest number is $10^{1048576}$. A 1 followed by more than a million zeroes!

Table 3.1: Qlib documentation

File	Function description

qin.c	**3.53 Modified Bessel function I of noninteger order**
	SYNOPSIS:
	`int bessel_I( v, x, y );`
	`qfloat *v,*x; // inputs`
	`qfloat *y; // output`
	Returns modified Bessel function of order v of the argument. Also known as $Y_v(x)$, formerly `qin` in CEPHES.
	The power series is:
	$$I_v(z) = \left(\frac{z}{2}\right)^v \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\,\Gamma(v+k+1)}$$
	For large x:
	$$I_v(z) = \frac{e^z}{\sqrt{2\,\pi\,z}} \left\{1 - \frac{(u-1)^2)}{1!\,(8z)} + \frac{(u-1^2)(u-3^2)}{2!\,(8z)^2} + \ldots\right\}$$
	asymptotically where $u = 4v^2$.
	EXAMPLE:
	`* digits(40)`
	`40`
	`* besseli(1,1)`
	`0.5651591039924850272076960276098633073289`

qincb.c	**3.54 Incomplete beta integral**
	SYNOPSIS:
	`int qincb( a, b, x, y );`
	`qfloat *a, *b, *x; // inputs`
	`qfloat *y; // output`
	$$IncB(a,b,x) = \frac{\Gamma(a+b)}{\Gamma(a)\,\Gamma(b)} \int_0^x t^{(a-1)} \times (1-t)^{b-1}\,dt$$
	ACCURACY: Expansions terminate at less than full working precision.
	EXAMPLE:
	`* digits(60)`
	`60`
	`* incbet(1,3,0.4) // Incomplete beta integral`
	`0.784`
	`* incbetinv(1,3,0.784) - 0.4`
	`1.00698658183186357318073814936563507193850885218223178580568 7E-103`

Table 3.1: Qlib documentation

File	Function description

qincbi.c

3.55 Inverse of incomplete beta integral

SYNOPSIS:

```
void beta_distribution_invQ(a,b, y, x);
qfloat *a, *b, *y; // inputs
qfloat *x; // output
```

Given y, the function finds x such that

$$qincb(a, b, x) = y$$

The routine performs up to 10 Newton iterations to find the root of $qincb(a, b, x) - y = 0$.

EXAMPLE:

```
* digits(60)
60
* incbet(1,3,0.4) // Incomplete beta integral
0.784
* incbetinv(1,3,0.784) - 0.4 // Inverse of incomplete beta integral
1.006986581831863573180738149365635071938508852182231785805687E-103
```

qjn.c

3.56 Bessel function of non-integer order

SYNOPSIS:

```
void bessel_J(v, x, y);
qfloat *v, *x; // inputs
qfloat *y; // output
```

Returns Bessel function of order v of the argument, where v is real. Negative x is allowed if v is an integer.
Two expansions are used: the ascending power series and the Hankel expansion for large v. If v is not too large, it is reduced by recurrence to a region of better accuracy.

EXAMPLE:

```
 * digits(60)
60
* besselj(0,3)
-0.260051954901933437624154695977331436819608653511293277055987
```

NOTE: In the original library this function was called qjn.

qjypn.c

3.57 Auxiliary function for Hankel's asymptotic expansion

SYNOPSIS:

```
void qjypn(n, x, y);
qfloat *n, *x; // inputs
qfloat *y; // output
```

$$J_n(x) = \sqrt{\frac{2}{\pi x}}[P(n, x)cos(X) - Q(n, x)sin(X)]$$

$$Y_n(x) = \sqrt{\frac{2}{\pi x}}[P(n, x)sin(X) + Q(n, x)cos(X)]$$

where arg of sine and cosine = X = $x - (0.5n + 0.25) * \pi$. We solve this for $Pn(x)$:

$$J_n(x)cos(X) + Y_n(x)sin(X) = \sqrt{\frac{2}{\pi x}}P_n(x)$$

Series expansions are set to terminate at less than full working precision.

Table 3.1: Qlib documentation

File	Function description
qkn.c	**3.58 Modified Bessel function K of order n**

SYNOPSIS:

```
void bessel_K(n, x, y);
qfloat *n, *x; // inputs
qfloat *y; // output
```

$$x^2\, y'' + xy' + (x^2 + v^2)y = 0$$

Algorithm for k_n:

$$K_n(x) = 0.5\frac{x}{2}^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!}(-\frac{x^2}{4})^k +$$

$$(-1)^n\, 0.5(\frac{x}{2})^n \sum_{k=0}^{\infty} \left\{p(k+1) + p(n+k+1) - 2log(\frac{x}{2})\right\} \frac{\frac{(x^2)}{4}^k}{k!(n+k)!}$$

where $p(m)$ is the psi function $p(1) = -EUL$ and

$$p(m) = -EUL + \sum_{k=1}^{m-1} \frac{1}{k}$$

For large x:

$$k_v(z) = \sqrt{\frac{\pi}{2z}} e^{-z} \left\{1 + \frac{u^2-1}{1!(8z)^1} + \frac{(u^2-1)}{2!(8z)^2} + ...\right\}$$

asymptotically where $u = 4v^2$. Converges to 1.4e-17 [4].

EXAMPLE:

```
* digits(55)
55
* besselk(0,0.53)
0.8765603804164856831291372854543554152943086526727639605
```

[4] The asymptotic series was starting to be used at 24. I have modified it to be 40. This allows calculating $k_0(34)$ with 120 digits precision To verify the results I used Bordeaux University GP calculator with precision of 132 digits. I also used the inline calculator `bc`.

Table 3.1: Qlib documentation

File	Function description

qlog.c	**3.59 Logarithm**
	SYNOPSIS:
	`void qlog(x, y);`
	`qfloat *x; // input`
	`qfloat *y; // output`
	After reducing the range to $[\frac{1}{\sqrt{2}}, \sqrt{2}]$ the logarithm is calculated with
	$$w = \frac{(x-1)}{(x+1)}$$
	$$\frac{ln(x)}{2} = w + \frac{w^3}{3} + \frac{w^5}{5} + ...$$
	EXAMPLE:
	`* exp(23)`
	`9744803446.2489026000346326848229752776493877640360069764`
	`* log(exp(23))`
	`23`

qmtst.c	**3.60 Verification program**
	This program calls several functions several thousand times with random inputs and displays statistics about the error ranges and accuracy.
	`Consistency test of math functions: Fri Oct 28 15:43:51 2022`
	`Max and rms errors for 10000 random arguments.`
	`A = absolute error criterion (but relative if >1):`
	`Otherwise, estimate is of relative error`
	`x=sqrt(square(x) ):  max = 8.09761E-0135  rms = 1.44359E-0135`
	`x=atan(tan(x) )    :  max = 5.50143E-0135  rms = 1.32752E-0135`
	`x=cbrt(cube(x))    :  max = 1.90569E-0135  rms = 4.31512E-0137`
	`x=sin(asin(x))     :  max = 1.62595E-0134  rms = 3.659E-0135`
	`x=log(exp(x))      :  max = 2.00952E-0133  rms = 4.43468E-0135`
	`x=log2(exp2(x) )   :  max = 1.38372E-0133A rms = 4.7460E-0135 A`
	`x=log10(exp10(x) ):  max = 1.82018E-0133  rms = 3.41053E-0135`
	`x=acosh(cosh(x) ) :  max = 1.68925E-0135  rms = 3.08459E-0137`
	`x=pow(pow(x,a),1/a): max = 1.89962E-0132  rms = 2.66437E-0134`
	`x=tanh( atanh(x) ):  max = 2.70618E-0134  rms = 2.4614E-0135`
	`x=asinh(sinh(x) ) :  max = 2.39223E-0135  rms = 5.13637E-0137`
	`x=cos(acos(x) )    :  max = 9.63075E-0135A rms = 1.8656E-0135 A`
	`Absolute error and only 2000 trials:`
	`x =ndtri(ndtr(x)) :  max = 2.31827E-0122  rms = 8.36731E-0124`
	`Legendre ellpk, ellpe:max= 4.03599E-0132  rms = 1.4728E-0133`
	`lgam(x)=log(gamma(x)):max= 1.37582E-0134A rms = 2.4302E-0135 A`

Table 3.1: Qlib documentation

File | **Function description**

qnthroot.c

3.61 Nth root

SYNOPSIS:

```
void qnthroot(x, N, y);
qfloat *x, *N; // input
qfloat *y; // output
```

Calculates the nth root of x with a newton iteration.

$$delta = \frac{1}{n}\left(\frac{x}{x_{k-1}^{n-1}}\right)$$

$$x_{k+1} = x_k + delta; x_0 = x^{1/n}$$

EXAMPLE:

```
* digits(45)
45
* nthroot(81.1,4)
3.000925497564966531056368548723519042392334468
* nthroot(81,4)
3
```

qpow.c

3.62 Power: x^y

SYNOPSIS:

```
void qfpow(x, p, y);
qfloat *x, *p; // input
qfloat *y;    // output
```

Calculates x^p. It uses the trivial identity: $x^y = e^{y\ log(x)}$.

qprob.c

3.63 Binomial Probability density

SYNOPSIS:

```
void qbdtr( k, n,p,y)
int k,n; // input
qfloat *p; // input
qfloat *y; // output
```

Returns in y the sum of the terms 0 through k of the binomial probability density:

$$\sum_{j=0}^{k} \binom{n}{k} p^j (1-p)^{n-j}$$

The terms aren't summed directly ; instead, the incomplete beta integral is employed according to the formula: $y = bdtr(k, n, p) = incbet(n-k, k+1, 1-p)$ The arguments must be positive, with p ranging from 0 to 1.

EXAMPLE:

```
* binomialdist(25,36,0.7)
0.533655157929798456675154200072751884
```

Table 3.1: Qlib documentation

File	Function description

qprob.c	**3.64 Complemented Binomial Probability density**
	SYNOPSIS:
	`void qbdtrc(k,n,p,y)`
	`int k,n; // input`
	`qfloat *p; // input`
	`qfloat *y; // output`
	Returns the sum of the terms k+1 through n of the Binomial probability density:
	$$\sum_{j=k+1}^{n} \binom{n}{j} p^j (1-p)^{n-j}$$

qprob.c	**3.65 Inverse binomial distribution**
	SYNOPSIS:
	`void qbdtri(int k,int n,qfloat *y,qfloat *p)`
	`int k,n; // inputs`
	`qfloat *y; // input`
	`qfloat *p; // output`
	Finds the event probability p such that the sum of the terms 0 through k of the binomial probability density is equal to the given cumulative probability y.
	This is accomplished using the inverse beta integral function and the relation
	$$1 - p = incbi(n-k, k+1, y)$$

qprob.c	**3.66 Chi-square distribution** SYNOPSIS: `void qchdtr( const qfloat *df,const qfloat *x, qfloat *y )` Returns the area under the left hand tail (from 0 to x) of the Chi square probability density function with v degrees of freedom. $$P(x\|v) = \frac{1}{2^{v/2}\,\Gamma(v/2)} \int_0^x t^{\frac{v}{2}-1} e^{-t/2}\, dt$$ where x is the Chi-square variable. The incomplete gamma integral is used according to the formula: $$y = chdtr(v,x) = igam(v/2.0, x/2.0)$$ The arguments must be both positive.

Table 3.1: Qlib documentation

File	Function description
qprob.c	**3.67 Complemented Chi-square distribution** SYNOPSIS: `void qchdtc(qfloat * const df, qfloat *const x, qfloat *y);` Returns the area under the right hand tail (from x to infinity) of the Chi square probability density function with v degrees of freedom: $$P(x\|v) = \frac{1}{2^{v/2}\,\Gamma(v/2)} \int_x^\infty t^{\frac{v}{2}-1} e^{-t/2}\,dt$$ where x is the Chi-square variable. The incomplete gamma integral is used according to the formula: $$y = chdtc(v, x) = igamc(v/2.0, x/2.0)$$ The arguments must be both positive.
qprob.c	**3.68 Inverse of complemented Chi-square distribution** SYNOPSIS: `void qchdti(qfloat *df,qfloat *y, qfloat *x );` Finds the Chi-square argument x such that the integral from x to infinity of the Chi-square density is equal to the given cumulative probability y. This is accomplished using the inverse gamma integral function and the relation $$x/2 = igami(df/2, y);$$
qprob.c	**3.69 F distribution** SYNOPSIS: `void qfdtr(int ia,int ib,qfloat *x,qfloat *y );` Returns the area from zero to x under the F density function (also known as Snedcor's density or the variance ratio density). This is the density of $x = (u1/df1)/(u2/df2)$, where u1 and u2 are random variables having Chi square distributions with df1 and df2 degrees of freedom, respectively. The incomplete beta integral is used, according to the formula $$P(x) = incbet(df1/2, df2/2, (df1 \times x/(df2 + df1 \times x))$$ The arguments a and b are greater than zero, and x is nonnegative.
qprob.c	**3.70 Gamma distribution function** SYNOPSIS: `void qgdtr(qfloat *a,qfloat *b,qfloat *x,qfloat *y )` Returns the integral from zero to x of the gamma probability density function: $$y = \frac{a^b}{\Gamma b} \int_0^x t^{b-1}\, e^{-at}\,dt$$

Table 3.1: Qlib documentation

File	Function description
qprob.c	**3.71 Complemented F distribution** SYNOPSIS: `void qfdtrc(const int ia,const int ib,qfloat *const x, qfloat *y)` Returns the area from x to infinity under the F density function (also known as Snedcor's density or the variance ratio density). $$1 - P(x) = \frac{1}{B(a,b)} \int_x^\infty t^{a-1}\,(1-t)^{b-1}\,dt$$ The incomplete beta integral is used, according to the formula $$P(x) = incbet(df2/2, df1/2, (df2/(df2 + df1 * x))$$
qprob.c	**3.72 Inverse of complemented F distribution** SYNOPSIS: `void qfdtri(int ia,int ib,qfloat *y,qfloat *x );` Finds the F density argument x such that the integral from x to infinity of the F density is equal to the given probability p. This is accomplished using the inverse beta integral function and the relations $$z = incbi(df2/2, df1/2, p)$$ and $$x = df2(1-z)/(df1z)$$ Note: the following relations hold for the inverse of the uncomplemented F distribution: $$z = incbi(df1/2, df2/2, p)$$ $$x = df2z/(df1(1-z))$$
qprob.c	**3.73 Complemented gamma distribution function** SYNOPSIS: `void qgdtrc(qfloat *const a,qfloat *const b, qfloat *x, qfloat *y );` Returns the integral from x to infinity of the gamma probability density function: $$y = \frac{a^b}{\Gamma b} \int_x^\infty t^{b-1}\,e^{-at}\,dt$$ The incomplete gamma integral is used, according to the relation $$y = igamc(b, ax)$$

Table 3.1: Qlib documentation

<table>
<tr><th>File</th><th>Function description</th></tr>
<tr><td>qprob.c</td><td>3.74 Negative binomial distribution
SYNOPSIS:
<code>void qnbdtr(const int k,const int n,const qfloat *p,qfloat *y);</code>
Returns the sum of the terms 0 through k of the negative binomial distribution:
$$\sum_{j=0}^{k}\binom{n+j+1}{j}p^n(1-p)^j$$
In a sequence of Bernoulli trials, this is the probability that k or fewer failures precede the nth success. The terms are not computed individually; instead the incomplete beta integral is employed, according to the formula
$$y = nbdtr(k, n, p) = incbet(n, k+1, p)$$
The arguments must be positive, with p ranging from 0 to 1</td></tr>
<tr><td>qprob.c</td><td>3.75 Complemented negative binomial distribution
SYNOPSIS:
<code>void qnbdtc(const int k, const int n,qfloat *const p,qfloat *y);</code>
Returns the sum of the terms k+1 to infinity of the negative binomial distribution:
$$\sum_{j=k+1}^{\infty}\binom{n+j+1}{j}p^n(1-p)^j$$
The terms are not computed individually; instead the incomplete beta integral is employed, according to the formula
$y = nbdtrc(k, n, p) = incbet(k+1, n, 1-p)$
The arguments must be positive, with p ranging from 0 to 1.</td></tr>
<tr><td>qhyperg.c</td><td>3.76 Pochhammer symbol (Falling factorial)
SYNOPSIS:
<code>void qPochhammerDown(Qfloatp const x,Qfloatp const n,Qfloatp y);</code>
Calculates the falling factorial:
$$(m_n) = \frac{m!}{(m-n)!} = x(x-1)(x-2)...(x-m+1)$$</td></tr>
<tr><td>qhyperg.c</td><td>3.77 Pochhammer symbol (Rising factorial)
SYNOPSIS:
<code>void qPochhammerUp(Qfloatp const x,Qfloatp const nn,Qfloatp y);</code>
Caldculates the rising factorial:
$$(m^n) = \frac{(m+n-1)!}{(m-1)!} = x(x+1)(x+2)...(x+m-1)$$</td></tr>
</table>

Table 3.1: Qlib documentation

File	Function description
qprob.c	**3.78 Poisson distribution** SYNOPSIS: `int qPoissonDistribution(const int k,qfloat *const m, qfloat *y)` Returns the sum of the first k terms of the Poisson distribution: $$\sum_{j=0}^{k} e^{-m}\frac{m^j}{j!}$$ The terms are not summed directly; instead the incomplete gamma integral is employed, according to the relation $y = pdtr(k, m) = igamc(k + 1, m)$ The arguments must both be positive.
qprob.c	**3.79 Complemented poisson distribution** SYNOPSIS: `int qPoissonDistributionComp(const int k,const qfloat *m,qfloat *y)` Returns the sum of the terms k+1 to infinity of the Poisson distribution: $$\sum_{j=k+1}^{\infty} e^{-m}\frac{m^j}{j!}$$ The terms are not summed directly; instead the incomplete gamma integral is employed, according to the formula: $y = pdtrc(k, m) = igam(k + 1, m)$ The arguments must both be positive.
qprob.c	**3.80 Inverse Poisson distribution** SYNOPSIS: `int qPoissondistributionInv(const int k,qfloat *const y,qfloat *m);` Finds the Poisson variable x such that the integral from 0 to x of the Poisson density is equal to the given probability y. This is accomplished using the inverse gamma integral function and the relation $m = igami(k + 1, y)$
qpsi.c	**3.81 Psi (digamma) function** SYNOPSIS: `void qpsi(qfloat *const x,qfloat *y)` $$\psi(x) = \frac{d}{dx} ln\Gamma(x)$$ is the logarithmic derivative of the gamma function. For general positive x, the argument is made greater than 16 using the recurrence $\psi(x + 1) = \psi(x) + 1/x$. Then the following asymptotic expansion is applied: $$\psi(x) = log(x) - \frac{1}{2x} - \sum_{k=1}^{\infty}\frac{B_{2k}}{2kx^{2k}}$$ where the B_{2k} are the Bernoulli numbers. For negative inputs the relation holds: $$\psi(-x) = \psi(x + 1) + \pi/tan(\pi(x + 1))$$ The accuracy is the full 132 digits [5].

[5] The accuracy has been improved from the original 22 digits by adding much more terms to the Bernoulli

Table 3.1: Qlib documentation

File	Function description
qrand.c	**3.82 Pseudo-random number generator** SYNOPSIS: `int qfrand(qfloat *q );` Yields a random number 1.0 <= q < 2.0. A three-generator congruential algorithm adapted from Brian Wichmann and David Hill (BYTE magazine, March, 1987, pp 127-8) is used to generate random 16-bit integers. These are copied into the significand area to produce a pseudorandom bit pattern.
qremain.c	**3.83 Floating point remainder** SYNOPSIS: `void qremain(qfloat *qa,qfloat *qb,qfloat *qc );` c = remainder after dividing b by a. If n = integer part of b/a, rounded toward zero, then qremain(a,b,c) gives c = b - n * a.
qremquo.c	**3.84 Floating point remainder according to C99** SYNOPSIS: `int qremquo(qfloat *const a,qfloat *const b,qfloat *c);` c = remainder after dividing b by a. If n = integer part of b/a, rounded toward zero, then qremain(a,b,c) gives c = b - n * a. Integer return value contains low order bits of the integer quotient n. According to the C99 standard, when y != 0, the remainder r = x REM y is defined regardless of the rounding mode by the mathematical relation r = x - ny, where n is the integer nearest the exact value of x / y; whenever | n - x / y | = 1/2, then n is even. Thus, the remainder is always exact. If r = 0, its sign shall be that of x. This definition is applicable for all implementations.
qshici.c	**3.85 Hyperbolic sine integral** SYNOPSIS: `void qshi(qfloat *const x,qfloat *y);` $$shi(x) = \int_0^x \frac{cosh(t) - 1}{t} dt$$ The power series used: $$\sum_0^{\infty} \frac{z^{2n+1}}{(2n+1)\ (2n+1)!}$$ ACCURACY: The series gives only 126 decimal digits
qshici.c	**3.86 Hyperbolic cosinus Integral** SYNOPSIS: `int qchi(qfloat *x,qfloat *y)` $$chi(x) = eul + ln(x) + \int_0^x \frac{cosh(t) - 1}{t} dt$$ The power series used is: $$chi(z) = eul + ln(z) + \sum_{n=1}^{\infty} \frac{z^{2n}}{2n\ (2n)!}$$ ACCURACY: The series gives only 126 decimal digits

numbers table (59 now)

Table 3.1: Qlib documentation

File	Function description
qsin.c	**3.87 Sine** SYNOPSIS: `int qfsin(qfloat *const x,qfloat *y)` Range reduction is into intervals of pi/2. Then $$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} ...$$
qsinh.c	**3.88 Hyperbolic sine** SYNOPSIS: `int qsinh(qfloat *const x,qfloat *y)` The range is partitioned into two segments. If |x| <= 1/4, $$sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} ...$$ Otherwise the calculation is $$sinh(x) = \frac{(e^x - e^{-x})}{2}$$
qspenc.c	**3.89 Dilogarithm** SYNOPSIS: `int qspenc(qfloat *const x,qfloat *y)` Computes the integral $$spence(x) = -\int_1^x \frac{log(t)}{t-1}\, dt$$ for $x >= 0$. A power series gives the integral in the interval (0.5, 1.5). Transformation formulas for 1/x and 1-x are employed outside the basic expansion range.
qsqrt.c	**3.90 Square root** SYNOPSIS: `int qfsqrt(qfloat *const x,qfloat *y)` If the input is between the range of long (128 bit) double, the long double calculation is used for seeding the Newton iteration. Otherwise range reduction involves isolating the power of two of the argument and using a polynomial approximation to obtain a rough value for the square root. Then Heron's iteration is used to converge to an accurate value.

Table 3.1: Qlib documentation

File	Function description
qstudt.c	**3.91 Student's distribution** SYNOPSIS: `void qstudt(int k,Qfloatp t,Qfloatp y)` Computes the integral from minus infinity to t of the Student t distribution with integer $k > 0$ degrees of freedom: $$\frac{\Gamma(\frac{k+1}{2})}{\sqrt{k \times \pi}\,\Gamma(k/2)} \int_{-\infty}^{t} \left(1 + \frac{x^2}{k}\right)^{-\frac{(k+1)}{2}} dt$$ Relation to incomplete beta integral: $$1 - stdtr(k,t) = 0.5 \times incbet(k/2, 0.5, z)$$ where $z = k/(k+t^2)$. For $t < -2$, this is the method of computation. For higher t, a direct method is derived from integration by parts. Since the function is symmetric about $t = 0$, the area under the right tail of the density is found by calling the function with $-t$ instead of t.
qstudt.c	**3.92 Inverse of Student's t distribution** SYNOPSIS: `void qsdtri(int k,qfloat *t,qfloat *y);` Given probability p, finds the argument t such that $stdtr(k,t)$ is equal to p.
qtan.c	**3.93 Circular tangent** SYNOPSIS: `void qftan(qfloat *const x,qfloat *y);` Domain of approximation is reduced by the transformation $x \rightarrow x - \pi floor((x+\pi/2)/\pi)$ then tan(x) is the continued fraction $$tan(x) = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \cfrac{x2}{7 - \cdots}}}}$$
qtan.c	**3.94 Circular cotangent** SYNOPSIS: `void qcot(qfloat *const x,qfloat *y);` $$cot(x) = \frac{1}{tan(x)}$$

Table 3.1: Qlib documentation

File	Function description
qtanh.c	**3.95 Hyperbolic tangent** SYNOPSIS: `void qtanh(qfloat *const x,qfloat *y);` For $x \geq 1$ the program uses the definition $$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$ For x < 1 the method is a continued fraction $$tanh(x) = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{x^2}{5 + \cfrac{x^2}{7 + \cdots}}}}$$
qtime.c	**3.96 Timing the qlib functions** This will go through some of the functions in the library and iterate them several thousand times to see how many mili-seconds they use. The source code is very straightforward, nothing sophisticated here. The code has three parts: 1. Very fast functions like `qmov` or `qclear`. Written in assembler they need a lot of iterations to be able to measure them. 2. The four operations. Adding, subtracting, inverse, and others. 3. High level functions like square root, the psi function, cosinus, etc. A smaller number of iterations is used to avoid very long waiting times in `qtime`.

Table 3.1: Qlib documentation

File	Function description
qyn.c	**3.97 Real bessel function of second kind and general order** SYNOPSIS: `void neumann_N(qfloat *const n,qfloat *const x,qfloat *y);` Returns Bessel function of order v. If v is not an integer, the result is $$Y_v(z) = \frac{\cos(\pi v) \times J_v(x) - J_{-v)}(x)}{\sin(\pi v)}$$ Hankel's expansion is used for large x: $$Y_v(z) = \sqrt{\frac{2}{(\pi z)}}(P \sin w + Q \cos w)$$ where $w = z - (.5v + .25)\pi$ $$P = 1 - \frac{(u-1)(u-9)}{2!(8z)^2} + \frac{(u-1)(u-9)(u-25)(u-49)}{4!(8z)^4} - ...$$ $$Q = \frac{(u-1)}{8z} - \frac{(u-1)(u-9)(u-25)}{3!(8z)^3} + ...$$ $$u = 4v^2$$ $$Y_n(z) = \frac{-(z/2)^{-n}}{\pi} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!}(z^2/r)^k + (2/pi)ln(z/2)J_n(z) -$$ $$-\frac{(z/2)^n}{\pi} - \sum_{k=0}^{n-1}(\psi(k+1) + \psi(n+k+1)\frac{(-z^2/4)^k}{k!(n+k)!}$$
qzetac.c	**3.98 Riemann zeta function** SYNOPSIS: `void qzetac(qfloat *const x,qfloat *y);` $$zetac(x) = \sum_{k=2}^{\infty} k^{-x}, x > 1$$ is related to the Riemann zeta function by $$Riemannzeta(x) = zetac(x) + 1$$ Extension of the function definition for x < 1 is implemented.

Index