Documentation for math2.h and math2.c

Steven Andrews, © 2004-2012

```
#ifndef __math2_h
#define __math2_h

/******** useful constants and macros ***********/

#define PI 3.14159265358979323846
#define SQRT2 1.41421356237
#define SQRTPI 1.7724538509
#define SQRT2PI 2.50662827462
#define SQUARE(X) ((X)*(X))
#define QUADPLUS(A,B,C) ((-(B)+sqrt((B)*(B)-4*(A)*(C)))/(2*(A)))
#define QUADMINUS(A,B,C) ((-(B)-sqrt((B)*(B)-4*(A)*(C)))/(2*(A)))

/******** integer stuff ***********/

int iseven(int x);
int is2ton(int x);
int next2ton(int x);
int isintegerD(double x);
int isinteger(float x);
int signD(double x);
int sign(float x);
int minus1to(int x);
int intpower(int n,int p);
double factorialD(int n);
float factorial(int n);
double chooseD(int n,int m);
float choose(int n,int m);
int gcomdiv(int m,int n);

/********* special functions ***********/

double sincD(double x);
float sinc(float x);
double boxD(double x);
float box(float x);
double bessj0D(double x);
float bessj0(float x);
double bessj1D(double x);
float bessj1(float x);
double gaussD(double x,double mean,double sd);
float gauss(float x,float mean,float sd);
double gammalnD(double x);
float gammaln(float x);
double gammpD(double a,double x);
float gammp(float a,float x);
double erfnD(double x);
float erfn(float x);
```

```
double erfncD(double x);
float erfnc(float x);
double erfccD(double x);
float erfccD(float x);
double experfcD(double x);
double experfcm1D(double x);
double erfcintegralD(double x);
float erfcintegral(float x);
double betalnD(double x1,double x2);
float betaln(float x1,float x2);
double hermiteD(double x,int n);
float hermite(float x,int n);

/********* interval stuff ***********/

float constrain(float x,float lo,float hi);
double reflectD(double x,double lo,double hi);
float reflect(float x,float lo,float hi);

/********* inverse functions ***********/

double inversefnD(double (*fn)(double),double y,double x1,double x2,int n);
float inversefn(float (*fn)(float),float y,float x1,float x2,int n);

/********* various utilities ***********/

double quadpts2paramsD(double *x,double *y,double *abc);
double fouriersumD(double *a,double *b,int n,double l,double x);
void radialftD(double *r,double *a,double *k,double *c,int nr,int nk);
void linefitD(double *x,double *y,int n,double *m,double *b);

/********* Hill functions ***********/

void SetHillParamD(double *hp,double a,double e,double n);
double HillFnD(double *hp,double x);
void HillFnComposeD(double *hp1,double *hp2,double *hp12);
void HillFnComposeNF1D(double *hp1,double *hp2,double *hpf1,double *hpf12);

#endif
```

Requires: `<math.h>`,`<stdlib.h>`, `<float.h>`, `"math2.h"`
Example program: `LibTest.c`

History: Modified 9/2/98. Modified again 1/00 and again 8/00. Works with Metrowerks C. Documentation updated 10/19/01. Updated 1/8/02. Fully tested. Added `isinteger` 2/20/02. Modified `next2ton` slightly 10/28/03. Added `QUADPLUS` and `QAUDMINUS` and renamed sqr macro to `SQUARE` 4/16/04. Added `quadpts2params` 9/20/05. Added `signD` 10/19/05. Added `fouriersumD` 12/16/05. Added the Hill function functions 11/28/06. Added `constrain` 4/3/07. General clean up, added several double precision functions, and removed a couple of single precision functions 4/21/08. Modified `SetHillParamD` and `HillFnD` 4/21/09. Added `bessy0D` and `bessy1D` functions 11/13/13. Added `bessi0D` function 2/6/17. Added diffgreen2D 2/8/17. Added `experfcm1D` 7/11/23.

This library file complements the standard math.h library with many useful functions. No effort is made to check for overflow problems (i.e. routines may return `Inf`, or comparable error codes). A couple routines are copied directly from *Numerical Recipes*. A routine called `bessj1int` was removed 10/18/01 since it was inferior to `bessj1`; the inputs and outputs are identical.

Summary of math functions

Name	Domain	Output
`SQUARE`	(–∞,∞)	X^2
`QUADPLUS`	$(-\infty,\infty)^3$	positive real root of quadratic eqn. if exists, or `nan` if not
`QUADMINUS`	$(-\infty,\infty)^3$	negative real root of quadratic eqn. if exists, or `nan` if not
`iseven`	(–∞,∞)	1 if even, 0 if odd
`is2ton`	(–∞,∞)	1 if `x` is an integer power of 2, 0 if not
`next2ton`	(–∞,∞)	Next higher power of 2, 1 if `x`=0, or 0 if `x`<0
`isinteger`	(–∞,∞)	1 if `x` is an integer, 0 if not
`sign`	(–∞,∞)	+1, 0, or –1 for a positive, zero, or negative argument
`minus1to`	(–∞,∞)	$(-1)^x$
`intpower`	$(-\infty,\infty)^2$	n^p, as an integer, `p`<0 returns 0
`factorial`	[0,∞)	`n`!, by computation of products; returns 1 for `n`<0
`choose`	[0,∞),[0,`n`]	`n` choose `m`
`gcomdiv`	$(-\infty,\infty)^2$	greatest common divisor using a varient of Euler's method
`sinc`	(–∞,∞)	sin(x)/x
`box`	(–∞,∞)	1 for $-1 \le x \le 1$ and 0 elsewhere
`bessj0`	(–∞,∞)	J_0 Bessel function using *Numerical Recipes* routine
`bessj1`	(–∞,∞)	J_1 Bessel function using *Numerical Recipes* routine
`bessy0`	(0,∞)	Y_0 Bessel function using *Numerical Recipes* routine
`bessy1`	(0,∞)	Y_1 Bessel function using *Numerical Recipes* routine
`bessi0`	(-∞,∞)	I_0 modified Bessel function using *Num. Recipes* routine
`gauss`	(–∞,∞)	Normalized Gaussian
`gammaln`	(–∞,∞)	Natural log of absolute value of gamma function
`gammp`	(0,∞),[0,∞)	Incomplete gamma fn., P(a,x); –1 for input out of domain
`erfn`	(–∞,∞)	Error function
`erfnc`	(–∞,∞)	Complementary error function
`erfcc`	(–∞,∞)	Complementary error function, with a different method
`experfc`	(–∞,∞)	$\exp(x^2)*\mathrm{erfc}(x)$
`experfcm1`	(–∞,∞)	$\exp(x^2)*\mathrm{erfc}(x)-1$
`erfcintegral`	(–∞,∞)	$\int_0^x \mathrm{erfc}(x')\, dx'$
`betaln`	$(-\infty,\infty)^2$	Natural log of the beta function
`hermite`	(–∞,∞),[0,∞)	Hermite polynomial by recursion; returns 0 for `n`<0

`diffgreen2D`	[0,∞),[0,∞)	Green's function for radially symmetric 2-D diffusion
`constrain`	(–∞,∞)	Puts *x* into range between `lo` and `hi`
`reflect`	(–∞,∞)	Reflects *x* into range between `lo` and `hi` using recursion
`HillFn`	[0,∞)	Hill function with variable `x` and parameters `hp`.

Defined constants

Name	Value	Meaning
`PI`	3.14159265358979323846	π
`SQRT2`	1.41421356237	√2
`SQRTPI`	1.7724538509	√π
`SQRT2PI`	2.50662827462	√(2π)

Macro functions

```
#define SQUARE(X) ((X)*(X))
```

Returns X^2.

```
#define QUADPLUS(A,B,C) ((-(B)+sqrt((B)*(B)-4*(A)*(C)))/(2*(A)))
```

`QUADPLUS` returns the positive real root of the quadratic equation for coefficients `A`, `B`, and `C`. If `A` and `C` have the opposite sign, then a positive root exists and is greater than 0.

```
#define QUADMINUS(A,B,C) ((-(B)-sqrt((B)*(B)-4*(A)*(C)))/(2*(A)))
```

`QUADMINUS` returns the negative real root of the quadratic equation for coefficients `A`, `B`, and `C`. If `A` and `C` have the opposite sign, then a negative root exists and is less than 0.

Integer functions

```
int iseven(int x);
```

Returns 1 if `x` is even and 0 if `x` is odd.

```
int is2ton(int x);
```

Returns 1 if `x` is an integer power of 2 (or if `x` is 0), and 0 if not.

```
int next2ton(int x);
```

Returns the next higher integer power of two.

```
int isintegerD(double x);
int isinteger(float x);
```

Returns 1 if `x` is an integer and 0 if not.

```
int signD(double x);
int sign(float x);
```

Returns sign of `x` as –1, 0 , or 1.

```
int minus1to(int x);
```

Returns $(-1)^{x}$.

```
int intpower(int n,int p);
```

Returns n^{p}, performed with simple multiplication.

```
double factorialD(int n);
float factorial(int n);
```

Returns `n`!, performed with simple multiplication. If large numbers are wanted, use `gammaln` instead.

```
double choose(int n,int m);
float choose(int n,int m);
```

Returns `n` choose `m`, as a float or double. This function uses simple multiplication.

```
int gcomdiv(int m,int n);
```

Returns the greatest common divisor of `m` and `n`. This always returns a positive number. If either input is 0, then 1 is returned.

Special functions

```
double sincD(double x);
float sinc(float x);
```

Returns $1/\sin(x)$, which is 1 at 0.

```
double boxD(double x);
float box(float x);
```

Returns 1 for $|x| \leq 1$ and 0 for $|x| > 1$.

```
double bessj0D(double x);
float bessj0(float x);
```

Returns J_0 Bessel function.

```
double bessj1D(double x);
float bessj1(float x);
```

Returns J_1 Bessel function.

```
double bessy0D(double x);
```

Returns Y_0 Bessel function.

```
double bessY1D(double x);
```

Returns Y_1 Bessel function.

```
double bessi0D(double x);
```

Returns the I_0 modified Bessel function. From *Numerical Recipes*.

```
double gaussD(double x,double mean,double sd);
float gauss(float x,float mean,float sd);
```

Returns a Gaussian with area 1, mean `mean` and standard deviation `sd`. `sd` may not be 0 but a negative `sd` yields a negative Gaussian with area -1.

```
double gammalnD(double x);
float gammaln(float x);
```

Returns the natural log of the gamma function. This uses direct summation for integer and half integer values of `x`, recursion for other negative values, and a formula from *Numerical Recipes* elsewhere. There is one recursive step for each integer for negative values, so large negative values are not recommended. Result is 1/0 for `x` = 0 or negative integers. Gamma function is positive everywhere except the following open intervals: (–1,0), (–3,–2), (–5,–4), ….

```
double gammpD(double a,double x);
float gammp(float a,float x);
```

Returns incomplete gamma function. Partly from *Numerical Recipes*.

```
double erfnD(double x);
float erfn(float x);
```

Returns the error function, calculated using `gammp`.

```
double erfncD(double x);
float erfnc(float x);
```

Returns the complementary error function, calculated using `gammp`.

```
double erfccD(double x);
float erfcc(float x);
```

Returns the complementary error function, calculated with Chebyshev's method. This was copied verbatim from *Numerical Recipies in C*, by Press et al., Cambridge University Press, Cambridge, 1988. It works for all `x` and has fractional error everywhere less than 1.2e-7.

```
double erfD(double x);
```

Returns erf(x), calculated with Chebyshev's method. This just returns 1-`erfccD(x)`.

```
double experfcD(double x);
```

Returns exp(x^2)*erfc(x). This is computed directly (using the `erfccD` function) for $|x| < 20$ and with a series solution for larger x values. The series solution was found in Carslaw and Jaeger section 2.7 equation 3 (page 71) or Crank eq. 3.39 (page 37), and extended with Mathematica. This equation is the Faddeeva function with an imaginary argument, $w(ix)$.

```
double experfcm1D(double x);
```

Returns exp(x^2)*erfc(x)-1. This calls the `experfcD` function and then subtracts 1 if x>0.05 and uses a series solution otherwise. I found the series solution using Mathematica, which is accurate to within 10^{-10} over the entire range considered.

This function exists due to roundoff errors that would otherwise arise when x is small.

```
double erfcintegralD(double x);
float erfcintegral(float x);
```

Returns the integral of the complementary error function from 0 to x, where x may be positive or negative. The equation for this is

$$\int_0^x \mathrm{erfc}(x')\,dx' = \frac{1-e^{-x^2}}{\sqrt{\pi}} + x\,\mathrm{erfc}(x)$$

```
double betalnD(double x1,double x2);
float betaln(float x1,float x2);
```

Returns the natural log of the beta function.

```
double hermiteD(double x,int n);
float hermite(float x,int n);
```

Returns the `n`'th Hermite polynomial at `x`. This uses recursion.

Green's functions

```
double diffgreen2D(double r1,double r2);
```

Returns the Green's function for radially symmetric 2-dimensional diffusion. Enter `r1` and `r2` as reduced radii, meaning the radii divided by the rms step length. This computes the function

$$\mathrm{grn}(r,r') = G_1(r_1)G_1(r_2)I_0(r_1 r_2) = \frac{1}{2\pi}e^{-\frac{r_1^2+r_2^2}{2}}I_0(r_1 r_2)$$

where $G_\sigma(x)$ is the normalized Gaussian with standard deviation σ and $I_0(x)$ is a modified Bessel function. See the rxn2Dparam_doc document for more about this equation. It is computed in its own function because it is difficult to compute numerically due to the fact that the Gaussians become small exponentially quickly while the Bessel function gets big exponentially quickly. This function combines the Bessel function code from `bessi0D` with the Gaussian computations, leading to a numerically stable result.

Interval stuff

```
float constrain(float x,float lo,float hi);
```

If `x` is between `lo` and `hi`, returns `x`. Otherwise, returns closer of `lo` or `hi`.

```
double reflectD(double x,double lo,double hi);
float reflect(float x,float lo,float hi);
```

Reflects `x` off of `lo` and `hi` as many times as needed until `x` is within [`lo`,`hi`]. Returns the reflected value of x. This uses recursion which is slow, so the routine could be sped up some. It is especially slow if `x` starts far below `lo` or far above `hi`. The function will hang if `lo` is not less than `hi`.

Inverse functions

```
double inversefnD(double (*fn)(double),double y,double x1,double x2,int n);
float inversefn(float (*fn)(float),float y,float x1,float x2,int n);
```

Returns the x value at which `fn`(x) = `y`, where x is between `x1` and `x2`, which bracket the inverse value. It uses a bisection method with `n` steps and an algorithm similar to one described in *Numerical Recipes*. Returned value is on (`x1`,`x2`), which may be in either order, and has a maximum error of $2^{-(n+1)}$(`x2`–`x1`). If there are multiple solutions, the one that is found first will be returned. If there are no solutions, the returned value is nearly equal to the endpoint that is closer to the solution (with the same error as above).

Various utilities

```
double quadpts2paramsD(double *x,double *y,double *abc);
```

For the quadratic $y=ax^2+bx+c$, this solves for the a, b, and c parameters from three (x,y) pairs. `x` is three known x values and `y` are the respective known y values. The result is returned in the three element vector `abc`, with a first, then b, and then c. The function uses the matrix **M** where $M_{ij} = x_i^{(2-j)}$, along with its determinant. The determinant is returned; a return value equal to 0 means that the parameters cannot be determined and `abc` was not calculated, and very small return values (positive or negative) indicate that there may be large numerical errors in `abc`. A zero determinant occurs if, and I think only if, two or more x values are identical.

```
double fouriersumD(double *a,double *b,int n,double l,double x);
```

This calculates the sum of a Fourier series for a function which is periodic on $2l$ ($-l$ to l, or 0 to $2l$, or any other interval). The sum is defined by:

$$y=\frac{a_0}{2}+\sum_{j=1}^{n-1}\left[a_j\cos\left(\frac{j\pi x}{l}\right)+b_j\sin\left(\frac{j\pi x}{l}\right)\right]$$

The parameters in the equation are exactly as they are in the function call. Note that `b[0]` is completely ignored and that the sum goes to $n-1$ rather than to n, where the latter is the standard math convention.

```
void radialftD(double *r,double *a,double *k,double *c,int nr,int nk);
```

Computes the radial Fourier transform of the function $a(r)$, returning it as $c(k)$. `r`, which has `nr` values, is the vector of input radii, and `a`, which also has `nr` values, lists the values of $a(r)$. Output values are in the `k` vector, which has `nk` values. The output is in `c`. The following integral is performed here:

$$\hat{f}(k)=\frac{1}{k}\sqrt{\frac{2}{\pi}}\int_0^{\infty} r\sin(kr)f(r)\,dr$$

See Arfken and Weber page 860 (problem 15.3.20) or my Cambridge notes p. B-8.

```
void linefitD(double *x,double *y,int n,double *m,double *b);
```

Fits a straight line to the `n` points that are listed in `x` and `y`, returning the slope and intercept in `m` and `b`, respectively. No checks are made that values are reasonable. Input values of `m` and `b` are ignored. `m` and/or `b` may be set to `NULL`, if that result isn't wanted.

Hill functions

```
void SetHillParamD(double *hp,double a,double e,double n,double b);
```

Sets vector of Hill function parameters called `hp` to `a`, `e`, `n`, and `b`. No math or checking is done, just assignment of `a`, `e`, `n`, and `b` to the 4-element `hp` vector, which needs to be pre-allocated. Note that prior versions of this function did not include the `b` term and they also used a 3-element `hp` vector; all 4 elements are required now.

```
double HillFnD(double *hp,double x);
```

Hill function. `hp` is 4-element vector of Hill function parameters with values A, E, N, and B respectively, and `x` is the variable. Note that prior versions of this function used a 3-element vector, although all 4 elements are required now. The returned value follows the Hill equation:

$$y = A\frac{x^N}{E^N + x^N} + B$$

```
void HillFnComposeD(double *hp1,double *hp2,double *hp12)
```

Hill functions H_1 and H_2 are defined by parameters `hp1` and `hp2`. Their composition is $f(x) = H_2(H_1(\mathrm{x}))$, which is not a Hill function but is similar to one. The Hill function that shares the same maximum amplitude, x-value at half maximum, and log-slope at half-maximum as $f(x)$ has parameters `hp12`, which are calculated and returned by this function. This function ignores B values.

```
void HillFnComposeNF1D(double *hp1,double *hp2,double *hpf1,double *hpf12);
```

Composition of Hill functions `hp1` and `hp2`, including negative feedback from 2 to 1, to yield a "close" Hill function with parameters `hpf12`. The reaction mechanism is that the activated product of reaction 2 is a reactant for the inactivation of reaction 1. Either, both, or neither `hpf1` and `hpf2` are returned; for one to be returned send in a non-`NULL` address. It is assumed, but not checked, that both input n values (`hp1[2]` and `hp2[2]`) are equal to 1. This function could be generalized to other input n values, but that hasn't been done yet. This function ignores B values.