Documentation for random2.h and random2.c

Steven Andrews, © 2008

Header, part I

The entire header file is quite long, so is not reproduced here. However, some of the earlier lines are important:

```
#ifndef __random2_h
#define __random2_h

// Comment out the following line if the Mersenne Twister is unavailable
#include "SFMT/SFMT.h"
```

The first portion is just the usual definition to prevent multiple inclusions of the header file. The SFMT line includes the Mersenne Twister (SFMT) header file. If the SFMT is unavailable, just comment out the line, which will cause the system-supplied random number generator to be used instead. No other modifications are required in the code. A test of a short program in which most of the time was spent in getting random numbers took 4.64 s with the SFMT and 4.05 s with the Macintosh system-supplied random number generator.

History

My library file random.c was started 5/12/95 and modified on a regular basis up to 11/13/06. It used the system-supplied random number generator exclusively. On 4/18/08, I rewrote it and saved the new version as random2.c, which fully supercedes the old version. This new version replaces the macro define statements with inline functions and allows the use of either the SFMT random number generator or the system-supplied random number generator. Changed `intrandp` to allow for unscaled cumulative probabilities 6/2/08. Added `randtableshuffleV` 4/4/11; edited shuffling functions 4/6/11 to use the exact Fisher-Yates algorithm.

12/~15/12 Minor change to sphererandCCF for C++ conformity.
  Changed radrandsphCCF declaration and code for trivial bug fix (type issue).
  Changed variable names in sphererandCCD and sphererandCCF to avoid name
    collisions with something else.

Compiling

To use a program that uses this library (on my computer, where the SFMT file is compiled separately), it's mostly the same as with any other library file: use a `#include` in the code and list the obect code in the Makefile. The only other addition is that the object file SFMT.o needs to be listed as well if the SFMT random number generator is used.

Either the SFMT random number generator or the system-supplied random number generator can be used. Also, the SFMT generator can be compiled in a variety of ways. On my computer, I enable the SSE2 option, for good speed, but can't use the AltiVec option.

Description

Most of these functions return random numbers, chosen from a variety of densities. Each function name contains a suffix that tells the type of the arguments and return value, such as 'F' for float, 'D' for double, and "ULI" for unsigned long int. Also, this suffix often contains either 'C' or 'O' letters to designate closed or open boundaries for the random number interval.

A lot of functions are defined in the header file as inline static functions, so that they will be copied over each time that they are referenced, which speeds execution.

If the Mersenne Twister is used, the random number generator needs to be initialized before it can be used. This is done with the `randomize` function.

Math

Math for various densities

Note that `1.0*rand()/RAND_MAX` returns a uniform density on [0,1] and `(rand()+1.0)/(RAND_MAX+1.0)` is uniform on (0,1]. The functions `randCCD` and `randOCD` also return random numbers with these respective densities. To convert these uniform densities to the density $\rho(x)$, first calculate the cumulative probability $P(x) = \int_{-\infty}^{x} \rho(x')dx'$, where it is seen that $P(x)$ is 0 at $x = -\infty$ and 1 at $x = \infty$. If the value for $y = P(x)$ is chosen with a uniform density, its value mapped onto x has the desired density. Thus, a function should return $x = P^{-1}(y)$.

If you want an event to happen with probability `x`, then make it happen if `randCOD()<x`.

Pseudo-random number issues

The CodeWarrior compiler on a Macintosh has `RAND_MAX` equal to $2^{15}-1$, whereas it is $2^{31}-1$ for the gcc compiler on Linux (I'm not sure which gcc version). The SFMT has a maximum value of $2^{32}-1$, which is a little better yet. It also allows 64-bit numbers, but I don't use them. Few numbers become a significant problem when rare events are required. For example, with the Macintosh numbers, it is possible to have an event happen an average of exactly 1 time per 32,767 iterations, or exactly 2 times, or so on, but not 1.5 times. If the random number generator isn't perfect, then the options are at least as sparse and with unknown intervals.

For the most part, I have not had any trouble with the randomness quality on a Macintosh, although in one instance I found net diffusion of randomly moving particles towards the center of the volume; this undoubtedly arose from an imperfect random number generator, although the precise problem is unclear. I solved it by shuffling the lookup table that I was using.

Math for `trianglerandD`

The "easy" problem is to find a random point in a 2-dimensional triangle which has been arranged so that point numbers 0, 1, and 2 have x values that increase from 0 to 1 and from 1 to 2. Suppose this is the case. The triangle point coordinates are (x_0,y_0), (x_1,y_1), and (x_2,y_2). From these, one can calculate the slopes from one point to another:

$$m_{01} = \frac{y_1 - y_0}{x_1 - x_0} \qquad m_{02} = \frac{y_2 - y_0}{x_2 - x_0} \qquad m_{12} = \frac{y_2 - y_1}{x_2 - x_1}$$

Also, one can calculate the triangle area as the distance that y_1 is above the 0-2 line, measured parallel to the y-axis, times the x distance between points 0 and 2, divided by two:

$$A = \frac{1}{2}\left[(y_1 - y_0) - m_{02}(x_1 - x_0)\right](x_2 - x_0)$$

This area is positive if y_1 is above the 0-2 line and negative if it is below. The integral of the triangle area, starting from point 0 and then normalized with respect to the triangle area, is found to be

$$Y(x) = \left\{ \begin{array}{ll} \frac{1}{2A}(m_{01} - m_{02})(x - x_0)^2 & x_0 \le x \le x_1 \\ 1 - \frac{1}{2A}(m_{02} - m_{12})(x - x_2)^2 & x_1 \le x \le x_2 \end{array} \right\}$$

This function is 0 at $x = x_0$, 1 at $x = x_2$, and increases monotonically in between. At x_1, the value can be calculated from either function to be

$$\begin{aligned} Y(x_1) &= \frac{1}{2A}(m_{01} - m_{02})(x_1 - x_0)^2 \\ &= 1 - \frac{1}{2A}(m_{02} - m_{12})(x_1 - x_2)^2 \end{aligned}$$

The inverse of the function is used to find a random x value given a uniformly distributed random Y value:

$$x = \left\{ \begin{array}{ll} x_0 + \sqrt{\frac{2AY}{m_{01} - m_{02}}} & 0 \le Y \le Y(x_1) \\ x_2 - \sqrt{\frac{2A(1-Y)}{m_{02} - m_{12}}} & Y(x_1) \le Y \le 1 \end{array} \right\}$$

This solves the problem of finding a random x-coordinate within the triangle. Next, a random y-coordinate is found. The y range at position x is

$$y \in \left[y_0 + m_{02}\left(x - x_0\right), \begin{Bmatrix} y_0 + m_{01}\left(x - x_0\right) & x_0 \le x \le x_1 \\ y_1 + m_{12}\left(x - x_1\right) & x_1 \le x \le x_2 \end{Bmatrix} \right]$$

A random y value is chosen within this range using a uniform density. Thus, the problem of finding a random set of coordinates within a triangle is solved for the simple case. Additional complexity arises from having to order the points.

For three-dimensions, the function calculates the unknown coordinate value by first finding the equation for the plane that includes the 3 triangle points, and then using it to find the unknown. The equation of a plane is

$$c_x x + c_y y + c_z z + c_k = 0$$

From the website local.wasp.uwa.edu.au/~pbourke/geometry/planeeq/, with minor notational changes, the equations for the coefficients c_x, c_y, c_z, and c_k, from the three triangle points, are

$$\begin{aligned} c_x &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\ c_y &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\ c_z &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\ -c_k &= x_1(y_2 z_3 - y_3 z_2) + x_2(y_3 z_1 - y_1 z_3) + x_3(y_1 z_2 - y_2 z_1) \end{aligned}$$

From the plane equation, the unknown z value is calculated for the random point from the known x and y values.

Function summary

The table below shows the domain, range, and densities of the functions given here. The domains are the suggested domains, although larger domains can sometimes be used as well. For example, `coinrand` can accept an input anywhere between –∞ and ∞, although the function always returns 0 if `p` < 0 and 1 if `p` > 1. The densities are only strictly correct in the limit that `RAND2_MAX` approaches infinity. In regions where the density is small (where $\rho(x)\Delta x \approx 1/$`RAND2_MAX`, for some characteristic Δx), a small set of random numbers is mapped to a large output range, leading to relatively sparse coverage.

Name	Domain	Range	Density
`randCCD`		[0,1]	1
`randCOD`		[0,1)	1
`randOCD`		(0,1]	1
`randOOD`		(0,1)	1
`randULI`		{0,1,…,`RAND2_MAX`}	1/`RAND2_MAX`
`unirandCCD`	$(-\infty,\infty)^2$	[`lo`,`hi`]	1/|`hi`−`lo`|

`unirandCOD`	$(-\infty,\infty)^2$	[lo,hi)	$1/\lvert hi-lo \rvert$
`unirandOCD`	$(-\infty,\infty)^2$	(lo,hi]	$1/\lvert hi-lo \rvert$
`unirandOOD`	$(-\infty,\infty)^2$	(lo,hi)	$1/\lvert hi-lo \rvert$
`signrand`		{–1,1}	{0.5,0.5}
`coinrandD`	[0,1]	{0,1}	{1–p,p}
`intrand`	[1,∞)	{0,1,…,n–1}	{1/n,1/n,…,1/n}
`exprandCOD`	[0,∞)	[0,∞)	$1/a*\exp(-x/a)$
	(–∞,0]	(–∞,0]	$1/a*\exp(-x/a)$
`logscalerandCCD`	(0,∞),(min,∞)	[min,max]	uniform on log scale
`powrandCOD`	(–∞,∞),(–∞,–1)	[xmin,∞)	$(1-m)/x_{min}*(x/x_{min})^m$
`radrandcircCCD`	(–∞,∞)	[0,r]	$2x/r^2$
`radrandsphCCD`	(–∞,∞)	[0,r]	$3x^2/r^3$
`thetarandCCD`		[0,π]	$1/2*\sin(x)$
`unirandsumCCF`	n>0, all m,s	[m–s√(3n),m+s√(3n)]	≈Gaussian with mean m, std. dev. s
`intrandpD`	n>0, $0 \le p_i \le 1$	{0,1,…,n–1}	$\{p_0,p_1-p_0,\ldots,1-p_{n-2}\}$
`poisrandD`	(–∞,∞)	[0,∞)	Poisson with mean xm
`binomialrandF`	[0,1],[0,∞)	[0,n]	Binomial deviate, prob. p, n trials
`gaussrandD`		(–∞,∞)	Gaussian with mean 0, std. dev. 1
`sphererandCCD`	$[0,\infty)^2$	$[-rad2,rad2]^3$	Point in spherical shell
`trianglerandCD`	$[0,\infty)^2$	$[-rad2,rad2]^3$	Point in spherical shell

Defines

The following defines are from the SFMT portion of the header. The same ones, but sometimes with different replacement text are used in the system-supplied random number generator.

```
#define RAND_BITS 32
```

This is the number of bits in a pseudo-random number that is gotten with the function `randULI`. It's 32 for the SFMT and 30 for the system-supplied generator (if the system-supplied generator has a different value, multiple random numbers are concatenated or a random number is trimmed down to yield exactly 30 bits).

```
#define RAND_MAX_30 1073741823
```

This is $2^{30}-1$, which is the largest random integer possible if `rand30` is used.

```
#define rand30() (gen_rand32()&RAND_MAX_30)
```

Returns a random integer with exactly 30 bits. How it's done depends on whether the SFMT is used or the system-supplied generator.

Functions (some header, some main file)

```
inline static double randCCD(void);
inline static double randCOD(void);
inline static double randOCD(void);
inline static double randOOD(void);
inline static float randCCF(void);
inline static float randCOF(void);
inline static float randOCF(void);
inline static float randOOF(void);
```

All of these functions do basically the same thing, which is return a real-valued random number that is between 0 and 1. The last letter is ‘D’ or ‘F’ to indicate whether the number is returned as a double or a float, respectively. There is no speed benefit to using the float option if the SFMT is used, whereas there may be a benefit if the system-supplied random number generator is used. The underlying random number has 32 bits with SFMT, 30 with the system-supplied ‘D’ option, and whatever the system offers naturally with the system-supplied ‘F’ option. The preceding two letters tell whether the endpoints at 0 and 1, respectively, are closed or open (‘C’ or ‘O’, respectively). Thus, for example, `randCOD()` returns a double that is on the interval [0,1).

```
inline static unsigned long int randULI(void);
```

This returns a random unsigned long int with `RAND_BITS` bits, which is between 0 and `RAND2_MAX`, inclusive.

```
long int randomize(long int seed);
```

Sets the random number generator seed to `seed` if the value is ≥0 or to the current time value if `seed` is <0. The seed that was used is returned.

```
inline static double unirandCCD(double lo,double hi);
inline static double unirandOCD(double lo,double hi);
inline static double unirandCOD(double lo,double hi);
inline static double unirandOOD(double lo,double hi);
inline static float unirandCCD(float lo,float hi);
inline static float unirandOCD(float lo,float hi);
inline static float unirandCOD(float lo,float hi);
inline static float unirandOOD(float lo,float hi);
```

Returns a uniformly distributed random number between `lo` and `hi`. The same suffix designations apply here as for the `rand*()` functions. Usually `lo` is less than `hi`, but they can also be equal or swapped. The sequence of open or closed designations in the function correspond to the variables `lo` and `hi`, respectively, regardless of their input values.

```
inline static int signrand(void);
```

Returns ±1 with equal probability of each.

```
inline static int coinrandD(double p);
inline static int coinrandF(float p);
```

Returns 1 with probability `p`, and 0 otherwise.

```
inline static int intrand(int n);
```

Returns an integer between 0 and n–1, inclusive, each with uniform probability. The probability distribution is correct if `n` is an integer power of 2, quite good if `n` is a small integer, and poor if `n` is a significant fraction of $2^{\text{RAND\_BITS}}$ and not a power of 2.

```
inline static exprandCOD(double a);
```

Returns an exponentially distributed random number between (and including) 0 and ±∞ (same sign as `a`) with a characteristic value of `a`.

```
inline static double logscalerandCCD(double min,double max);
```

Returns a random number that is exponentially distributed between `min` and `max`, including both ends. Use this function for a uniformly distributed random variable for a variable that is plotted on a log axis.

Math. Suppose x' is a uniformly chosen random variable between ln `min` and ln `max` (this is easiest to envision with base 10 logs and a log x axis). Then, $e^{x'}$, is the desired value. Therefore, x' = `unirandCCD(`ln `min`,ln `max)` and x = exp(`unirandCCD(ln min`,ln `max)`. I'm not sure what the actual probability density is.

```
inline static powrandCOD(double xmin,double power);
```

Returns a random number chosen from a power law distribution with slope `power`, which needs to be <–1. `xmin` is typically positive, in which case it is the smallest number that can be returned; it can also be negative, which just switches the sign of the returned value.

```
inline static radrandcircCCD(double r);
```

This is intended for use in choosing a random radius within a circle of radius `r`. In combination with a random angle (uniform between 0 and 2π), this yields a random point uniformly distributed within the circle.

```
inline static radrandsphCCD(double r);
```

This is intended for use in choosing a random radius within a sphere of radius `r`. In combination with a random spherical angle, this yields a random point uniformly distributed within the sphere.

```
inline static double thetarandCCD(void);
```

This is intended for use in choosing a random θ direction in spherical coordinates. The answer is between 0 and π, inclusive, where 0 is parallel to the z-axis and π is antiparallel.

```
double unirandsumCCD(int n,double m,double s);
float unirandsumCCF(int n,float m,float s);
```

This adds together `n` random variables from a uniform density and then scales the sum to yield a mean of `m` and standard deviation `s`. It's a quick alternative for a Gaussian-like density, although not as fast or as well distributed as a look-up table and interpolation (see `randtable`), and also less good than `gaussrandD`. This function used to be misleadingly named `binomrand`.

```
int intrandpD(int n,double *p);
int intrandpF(int n,float *p);
```

This is similar to `intrand`, but allows non-uniform probabilities for each integer. `p` is sent in as a list of unscaled cumulative probabilities for each integer. Since they are cumulative, `p` is an increasing list of non-negative numbers. If they are scaled then p_{n-1} equals 1; they may also be unscaled, meaning that each cumulative probability is effectively divided by p_{n-1}. Results will always be between 0 and `n-1`, even with incorrect `p` values.

```
int poisrandD(float xm);
int poisrandF(float xm);
```

Returns an integer chosen from a Poisson density with mean `xm`, which will typically be in the range `xm±√xm`. This routine is copied almost verbatim from *Numerical Recipies*. A feature which the book routine has and which is kept here is that if the routine is called more than once with the same value of `xm`, it doesn't recalculate some variables, in order to speed up the routine. Negative values of `xm` are possible but always return a value of 0.

```
float binomialrandF(float p,int n);
```

Returns a random integer (as a float) chosen from a binomial distribution for `n` trials, each with probability `p`. It is the number of successes for these `n` trials. The routine was copied nearly verbatim from *Numerical Recipies*.

```
double gaussrandD();
float gaussrandF();
```

Returns a normal deviate with mean 0 and standard deviation 1 using the Box-Muller transformation described in *Numerical Recipies*.

```
void circlerandD(double *x,double radius);
```

Returns a 2-dimensional random point in `x` which is uniformly distributed on the circle that has radius equal to `radius`.

```
void sphererandCCD(double *x,double rad1,double rad2);
void sphererandCCF(float *x,float rad1,float rad2);
```

Returns a 3 dimensional point in `x` which is uniformly distributed within a spherical shell bounded on the inside by `rad1` and the outside by `rad2` (both inclusive). For a fixed radius, set both `rad1` and `rad2` to the radius. The input contents of `x` are ignored although it needs to be allocated to at least size 3.

```
void ballrandCCD(double *x,int dim,double radius);
```

Returns a `dim` dimensional point in `x` which is uniformly distributed within a ball of radius `radius`.

This algorithm, which is partly from the internet, is as follows. First, the `x` vector is set to Gaussian distributed random numbers because these create a symmetric distribution about the origin in `dim`-dimensional space. Dividing the `x` vector by the actual radius of the `x` vector about the origin (`rad`) leads to a value that is uniformly distributed on the surface of the unit ball. Then, a random radius is generated as $RU^{1/\mathtt{dim}}$, where R is the ball radius and U is a uniform random value between 0 and 1. These two scalings are combined into a single line here.

```
void trianglerandCD(double *pt1,double *pt2,double *pt3,int dim,double *ans);
```

Given a triangle defined by the three Cartesian coordinates `pt1`, `pt2`, and `pt3`, each of which has dimensionality `dim`, this returns in `ans` the Cartesian coordinates for a random point within the triangle using a uniform density. The triangle edges are included with the triangle. This function should work for all possible inputs with 2 or 3 dimensions. I'm fairly sure that it yields incorrect answers for `dim`>3.

If the system is two-dimensional, this function first copies over the x and y values for the points such that the new x values are non-decreasing from point 0 to point 1 to point 2; then it uses the math presented above to find a random location. It should be valid even if x and/or y values equal each other. If the system is more than two dimensional, this finds the dimension that the triangle has the smallest range on, puts that at the end, calls itself recursively to find a random point in the reduced dimensional space, calculates the remaining coordinate, and swaps back. The calculation of the remaining coordinate is also explained above in the math section. It should be reasonably easy to extend the unknown coordinate calculation for arbitrary dimensional space, but I haven't done that yet.

```
void randtableD(double *a,int n,int eq);
void randtableF(float *a,int n,int eq);
```

This fills in a lookup table with entries for quickly converting a uniform density to an alternate density, using `eq` to indicate which density is desired. `n` is the number of elements in the table. If `eq` is 1, the density is a normal density with mean 0 and standard deviation 1; returned values range from $-\mathrm{erf}^{-1}(0.5/n-1)$ to $\mathrm{erf}^{-1}(0.5/n-1)$. For example, if `rt` is a table with 1024 elements, the following expression would return a normally distributed random variable with mean `mu` and standard deviation `sd`: `x=mu+sd*rt[randULI()&1023]`, also the range is from –3.297 to 3.297. Clearly, there are only `n` possible outcomes in this expression, which could be corrected by linear interpolation and somewhat slower and lengthier code. If eq is 2, the density is from the part of a complementary error function for $x \geq 0$, scaled so that the standard deviation of the underlying Gaussian is 1. In this case, `x=sd*[randULI&1023]` yields a random number for corresponding standard deviation of `sd`. The range is 0.0004 to 3.186. Currently, no other `eq` values are recognized.

Here is the logic used in the algorithm. Ideally, the cumulative probability function has a range from 0 to 1, but, more generally, it goes from *min* to *max*. If there are to be n samplings, then $\Delta y = (max-min)/n$. Because sampling are wanted with equal spacings, but not at the exact ends, they are at $(i+0.5)\Delta y + min$. Also, ideally, the argument of the cumulative probability function is x. If instead, the argument is x/k, then the result from the inverse function needs to be multiplied by k (for example, the integral of a Gaussian is $\mathrm{erf}(x/\sqrt{2})$).

```
void randshuffletableD(double *a,int n);
void randshuffletableF(float *a,int n);
void randshuffletableI(int *a,int n);
void randshuffletableV(void **a,int n);
```

These shuffle a list of `n` numbers (or `void*`s) such that each item is equally likely to end up at any position in the list. These functions used a slightly imperfect algorithm up to 4/6/11, when I modified the function to use the exact Fisher-Yates algorithm.

`void showdist(int n,float low,float high,int bin);`

This is only intended for debugging other functions, so it is not a general routine. It plots a bar graph (`bin` bars that range from the first bar center at `low` to the last bar center at `high`) showing the distribution of `n` random variables from whatever function is hard-coded into it. This function also displays the actual mean and standard deviation.