Random Library

Documentation and Implementation Status

`Note:` The new random library does not work in the same way as the old one. This means that some applications that used the random library provided with the 1.0.2 release will be `broken`. However, porting these applications to the new random library will be fairly easy since large efforts were made to adhere to the standard set with the last version and some backwards compatibility hooks were incorporated.

Porting to Random 0.7

Overview

First, a few thoughts on pseudorandom number generation. It's hard to do right. There are many problems: the root cause, of course, is that computer algorithms themselves are not random. But there are also problems with defining "random", coming up with good tests for generators, and implementing algorithms correctly and efficiently. The history of pseudorandom number generation in simulation work is mostly embarassing. This library attempts to do a decent job of generating random numbers, as well as documenting how things work and what shortcomings there are. If you want to learn more about random number generation, the bibliography [See: `$(SWARMHOME)/src/random/docs/SOURCES.for.0.7`] has useful notes. Knuth is the main reference in this realm, but too old to describe most of the particular generators used here.

```
The code here represents an effort to implement several efficient, reasonably safe
generators.  The algorithms come from reading the literature [See:  $(SWARMHOME)/src/random/docs/SOURCES.for.0.7]:
these algorithms have been implemented as accurately as possible and run through some
simple tests.  There is always a chance that some algorithm here is no good - there's
also the chance that it is implemented incorrectly.
```

```
For best results, library users should test these generators themselves in some domain-specific
way.  One easy way to do this is to run an experiment twice:  once with one class
of generator (say, PMMLCG), and once with another (say, SWB). If the results differ
```

Random Library: Documentation and Implementation Status

This is version 0.7 of *Random*. It was donated by Sven Thommesen. Version 0.6 was a reimplementation of most of Nelson Minar's original random with many changes and a new interface. This version adds many more generators and distributions and changes the interface somewhat.

The implementation may be a little unstable. It hasn't been fully tested. Sven has also contributed a test suite which we will make available on the web site. But, as with any pseudo-random number generation library, results obtained from it should be examined very closely. But we are reasonably sure the generators and distributions have been correctly implemented.

Documentation

More documentation is available within the Swarm source distribution, itself [Ref: $(SWARMHOME)/src/random/docs/. These docs consist of:

WHATS.NEW.in.0.7
: Additions to Random since v. 0.6

WHATS.CHANGED.in.0.7
: Changes to Random since v. 0.6

WHATS.LEFT.TODO
: Improvements that may make it into v. 0.8

README.Generators.v07
: Usage Guide for generator objects

README.Distributions.v07
: Usage Guide for distribution objects

CHOOSING.A.GENERATOR
: Notes on choosing a generator for your sim

DEFAULT.GENERATORS
: Default generators invoked by distributions

SOURCES.for.0.7

Random Library: Porting to Random 0.7

The changes from version 0.6 to 0.7 consist mostly of a restructuring of the implementation of the library. However, there are some interface changes, as well, and the porting of applications written to work with the 0.6 version might require changes to accomodate the new interface. Also, it is advised that you take a look at the new interface, regardless of whether your application compiles and runs with the new random because the old interface is deprecated and will eventually be removed and because some of the functionality of the library changed such that the execution of your application may differ.

Minimalist Porting Notes:

What is described here is the minimum a normal random user would have to change to get her apps running with the 0.7. For complete documentation of the changes made to the random library, check the files:

$(SWARMHOME)/src/random/docs/WHATS.CHANGED.in.0.7
and
$(SWARMHOME)/src/random/docs/WHATS.NEW.in.0.7

1. The 'magic numbers' used by setState/getState to identify generators and distributions have changed. The sizes of the records returned by getState have also changed. Therefore version 0.7 generators will refuse to setStatea generator based on version 0.6 data.
2. The setState method has been renamed to setStateFrom, and getState to putStateInto.
3. The verifySelf methods have been exorcised. Their code has been moved to a separate test program.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

The two files 'methods.SimpleGenerators.h' and 'methods.SplitGenerators.h' in directory /random/docs show the complete sets of methods implemented for the two most common types of generator.

Generators

- InternalState
(Prot.html#InternalState)
- SingleSeed
(Prot.html#SingleSeed)
- MultiSeed
(Prot.html#MultiSeed)
- Simple
(Prot.html#Simple)
- SimpleOut
(Prot.html#SimpleOut)
- SplitSingleSeed
(Prot.html#SplitSingleSeed)
- SplitMultiSeed
(Prot.html#SplitMultiSeed)
- Split
(Prot.html#Split)
- SplitOut
(Prot.html#SplitOut)
- SingleShortGenerator
(Prot.html#SingleShortGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed>
- SingleLongGenerator
(Prot.html#SingleLongGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>
- CombinedShortGenerator
(Prot.html#CombinedShortGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>
- CombinedLongGenerator
(Prot.html#CombinedLongGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>
- SingleShortSplitGenerator
(Prot.html#SingleShortSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed>
- SingleLongSplitGenerator

  - LCG2gen
    (LCG1-3gen.html#LCG2gen)
    : LCGgen <LCGgen, CREATABLE>
  - LCG3gen
    (LCG1-3gen.html#LCG3gen)
    : LCGgen <LCGgen, CREATABLE>

- MRGgen
  (MRG5-7gen.html)
  <SingleLongGenerator>
  - MRG5gen
    (MRG5-7gen.html#MRG5gen)
    : MRGgen <MRGgen, CREATABLE>
  - MRG6gen
    (MRG5-7gen.html#MRG6gen)
    : MRGgen <MRGgen, CREATABLE>
  - MRG7gen
    (MRG5-7gen.html#MRG7gen)
    : MRGgen <MRGgen, CREATABLE>
- MT19937gen
  (MT19937gen.html)
  <SingleLongGenerator, CREATABLE>
- MWCAgen
  (MWCA-Bgen.html)
  <SingleLongGenerator, CREATABLE>
- MWCBgen
  (MWCA-Bgen.html#MWCBgen)
  <SingleLongGenerator, CREATABLE>
- PMMLCGgen
  (PMMLCG1-9gen.html)
  <SingleShortGenerator>
  - PMMLCG1gen
    (PMMLCG1-9gen.html#PMMLCG1gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG2gen
    (PMMLCG1-9gen.html#PMMLCG2gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG3gen
    (PMMLCG1-9gen.html#PMMLCG3gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG4gen

- TGFSRgen
(TGFSRgen.html)
<SingleLongGenerator>

  - TT403gen
  (TGFSRgen.html#TT403gen)
  : TGFSRgen <TGFSRgen, CREATABLE>
  - TT775gen
  (TGFSRgen.html#TT775gen)
  : TGFSRgen <TGFSRgen, CREATABLE>
  - TT800gen
  (TGFSRgen.html#TT800gen)
  : TGFSRgen <TGFSRgen, CREATABLE>

Distributions

- ProbabilityDistribution
(Prot.html#ProbabilityDist)
<Create, Drop, InternalState>
- BooleanDistribution
(Prot.html#BooleanDistribution)
<ProbabilityDistribution>
- IntegerDistribution
(Prot.html#IntegerDistribution)
<ProbabilityDistribution>
- UnsignedDistribution
(Prot.html#UnsignedDistribution)
<ProbabilityDistribution>
- DoubleDistribution
(Prot.html#DoubleDistribution)
<ProbabilityDistribution>
- Normal
(Prot.html#Normal)
<DoubleDistribution>
-
  RandomBitDist
  (RandomBitDist.html)
  <BooleanDistribution, CREATABLE>
- BernoulliDist
(BernoulliDist.html)
<BooleanDistribution, CREATABLE>

Random Library: Protocol Reference

InternalState

```
//
// InternalState --
//   Methods to save the internal state of an object (genera-
tor, distribution)
//   to a memory buffer allocated by the calling pro-
gram, and to set the state
//   of an object from previously saved state data, provided in a mem-
ory buffer.
//   Method -getStateSize specifies the minimum buffer size needed.
//
@protocol InternalState
- (unsigned) getStateSize; // size of buffer needed
- (void) putStateInto: (void *) buffer; // save state data for later use
- (void) setStateFrom: (void *) buffer; // set state from saved data
- (void) describe: (id) outStream; // prints ascii data to stream
- (char *)      getName; // returns name of object
- (unsigned) getMagic; // object's 'magic number'
@end
```

The InternalState protocol provides attributes common to the generators and distributions provided in the random library. This protocol could be used for any class of objects that might contain an internal state, including things unrelated to pseudo random number generation.

The "state" of a given object depends on the type of object it is. In the case of random, both the bit generators and the distributions are capable of carrying state. The state is, in all cases, a structure that is understood by the object with which it is associated. The details of this structure can be found in the code for the given generator or distribution; but, in general, the user is not expected to understand the structure. Where the state must be saved and used to restart, the user is expected only to read and write the block of memory from and to the disk and pass the pointer to and from the object with which the state is associated. [See the Usage Guide (./UsageGuide.html)
for examples of how to do this.] Please note that different computer architectures store binary data in memory in different ways (using different byte orders). Since this version of the InternalState protocol stores data in binary form, you cannot expect to be able to use these methods to transfer data between different computers. We hope to lift this restriction in a future version.

(unsigned) -getStateSize returns an unsigned integer indicating how many bytes of memory are necessary to save the state of the object.

```
USING
- setStateFromSeeds: (unsigned *) seeds;
- (unsigned)  lengthOfSeedVector;
- (unsigned *) getMaxSeedValues;
- (unsigned *)  getInitialSeeds;
@end
```

Simple

```
@deftype Simple
CREATING
+ createWithDefaults: aZone;
USING
- setAntithetic: (BOOL) antiT;
- (BOOL) getAntithetic;
- (unsigned long long int) getCurrentCount;
@end
```

SimpleOut

```
@deftype SimpleOut
USING
- (unsigned)    getUnsignedMax;
- (unsigned) getUnsignedSample;
- (float)       getFloatSample; // using 1 unsigned
- (double)      getThinDoubleSample; // using 1 unsigned
- (double)      getDoubleSample; // using 2 unsigneds
- (long double) getLongDoubleSample; // using 2 unsigneds
@end
```

SplitSingleSeed

```
@deftype SplitSingleSeed
CREATING
+ create: aZone
   setA: (unsigned) A  // # of virtual generators
   setv: (unsigned) v  // log2(#segments/generator)
   setw: (unsigned) w // log2(segment length)
setStateFromSeed: (unsigned) seed;
USING
- setStateFromSeed: (unsigned) seed;
- (unsigned) getMaxSeedValue;
- (unsigned) getInitialSeed;
@end
```

SplitMultiSeed

```
@deftype SplitMultiSeed
CREATING
+ create: aZone
   setA: (unsigned) A  // # of virtual generators
   setv: (unsigned) v  // log2(#segments/generator)
```

No extra messages added.

CombinedShortGenerator
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>

No extra messages added.

CombinedLongGenerator
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>

No extra messages added.

SingleShortSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed>

No extra messages added.

SingleLongSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

No extra messages added.

CombinedShortSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

No extra messages added.

CombinedLongSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

No extra messages added.

Distribution Protocols

ProbabilityDistribution
<Create, Drop, InternalState>

```
// ProbabilityDistribution --
//
// A process for generating a sequence of random numbers matching the
//   frequencies defined by a specific distribution function.  The
//   process is driven by input from a supplied uniform random generator.
//
@deftype ProbabilityDistribution <Create, Drop, InternalState>
CREATING
+ createWithDefaults: aZone;
+ create: aZone setGenerator: (id) generator;
+ create: aZone setGenerator: (id) generator
setVirtualGenerator: (unsigned) vGen;
+ createBegin;
USING
```

```
//   values as represented by double-precision floating point values.
//
@deftype DoubleDistribution <ProbabilityDistribution>
- (double) getDoubleSample;
@end
```

Normal
<DoubleDistribution>

```
//
// NormalDistribution -- a well-known continuous probability distribution
// LogNormalDistribution -- also well-known ...
//
@deftype Normal <DoubleDistribution>
CREATING
+ create: aZone setGenerator: generator
                setMean: (double) mean setVariance: (double) variance;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setMean: (double) mean setVariance: (double) variance;
USING
- setMean: (double) mean setVariance: (double) variance;
- (double) getMean;
- (double) getVariance;
- (double) getStdDev;
- (double) getSampleWithMean: (double) mean
withVariance: (double) variance;
@end
```

(./Interface.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

Parameters: | unsigned int r // long lag
| unsigned int s // short lag

State: | unsigned int state[r] // state vector

Output types: | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range: | Any number in the range [0,2^32-1] may be returned
| (i.e. [0,4294967295].)

Valid seeds: | Using create:setStateFromSeed: [1,2^32-1]
| Using create:setStateFromSeeds: [1,2^32-2]

Cycles: | With properly chosen parameters, these generators
| have a cycle of length about 2^r.

Output Quality: | Low order bits are poor, and there are other problems.
| SWB is a better generator in the same class, and it is
| only slightly slower.

Reference for | Knuth, op.cit.
output quality: |

ACGgen | On a 486/66, the following approximate measures were obtained:
speed: | -getUnsignedSample: 2.702 uS
| -getFloatSample, getDoubleSample: 4.482 uS
| -getFatDoubleSample, getLongDoubleSample: 8.299 uS

Relative speed:
| Speed 1.369 (time 0.731) relative to MT19937 getUnsignedSample
--------------- | ---

Parameters: | r = 55
| s = 24

Reference for | These values recommended by Knuth.
parameters: |

Cycles: | With the chosen parameters, this generator
| has a single full cycle of length ~ 2^r.

Output Quality: | Low-order bits are poor.

C2LCGXgen Class

```
//
// C2LCGX --
//   short component based generator with splitting facilities. Recommended.
//
@deftype C2LCGXgen  <CombinedShortSplitGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C2LCGX

Description:  | Combined random generator using 2 (PMM)LGC generators

Reference:  | Pierre L'Ecuyer and Serge Cote,
| "Implementing a Random Number Package with Splitting
| Facilities." ACM Transactions on Mathematical Software
| vol. 17 no. 1, March 1991, pp. 98-111.
| Their name for the generator: 'split.c'.
| See also <http://www.iro.umontreal.ca/~lecuyer>.

Comments:  | This portable generator is based on a backbone generator
| which is a combination of 2 (PMM)LCG generators. It has a
| period length of almost 2^61 (2.3e18).

| The backbone generator's period can be split up into a
| number of 'virtual generators' (A), each of which can be set
| to access a number of 'segments' (V) of length W, subject to
| the constraint that A * V * W <= 2^60.

| This is possible because the backbone generator has
| 'jump-ahead facilities' -- i.e. a quick method of calculating
| the state of the generator at a given point in its cycle
| without having to actually draw a large number of variates.

Algorithm | state(j) = seed(j)
for each | newstate(j) = (a(j) * oldstate(j)) mod m(j)
generator:  | output(j) = state(j)

Algorithm | output = (state(1) - state(2)) mod m(0)
for combination:|
```

```
| straightforward. Unfortunately, on x86 at least, there is
| a significant speed penalty for doing so.
| The output, however, would be identical. YMMV.

C2LCGXgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        7.029 uS
|   -getFloatSample, getDoubleSample:          9.049 uS
|   -getFatDoubleSample, getLongDoubleSample: 16.725 uS

Relative speed:
| Speed 0.526 (time 1.900) relative to MT19937 getUnsignedSample
--------------- . ----------------------------------------------------------
--

Parameters:  | a[COMPONENTS] = {      40014,      40692 }
| m[COMPONENTS] = { 2147483563, 2147483399 }

Reference for | L'Ecuyer and Cote, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length close to 2^60 (2.3e18).

Output Quality:  | The authors performed an exhaustive search for component
| generators that would yield the best statistical quality.

Reference for | L'Ecuyer and Cote, op. cit.
output quality:  |
```

(./ACGgen.html)

(./Interface.html)

(./C2MRG3gen.html)

C2MRG3gen Class

```
//
// C2MRG3 --
//   long component based generator recommended for use.
//
@deftype C2MRG3gen <CombinedLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C2MRG3

Description:  | Combination of 2 Multiple Recursive LCG Generators

Reference:  | Pierre L'Ecuyer,
| "Combined Multiple Recursive Random Number Generators."
| See <http://www.iro.umontreal.ca/~lecuyer>.

| For a discussion of MRG generators, see MRGgen.h.

Algorithm:  | For each generator:
| x_n = (a_1*x_n-1 + a_2*x_n-2 ... + a_k*x_n-k) mod m
| Combining them:
| x = (x1 - x2) mod m1

Comments:  | Combinations of like generators are shown to have better
| statistical properties than single generators.

Implementation:  | The components of this generator each has two nonzero
| multipliers (and one that's zero). They use different
| moduli (2^31-1, 2145483479.)

Parameters:  | unsigned int m[2] // modulus
| unsigned int a1[2], ak[2] // multipliers
| unsigned int k[2] // the order of the generator

State:  | unsigned int x[k][2] // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
```

```
Output Quality:  | Good. Parameters chosen on the basis of theory
| and empirical testing.

Reference for | L'Ecuyer et al, op. cit.
output quality:  |
```

(./C2LCGXgen.html)

(./Interface.html)

(./C2TAUS1-3gen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
| unsigned int Mask[2] // 2^P - 1

State:  | unsigned int state[2] // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,2^31-1] may be returned
| (i.e. [0,2147483647].)

Valid seeds:  | Using create:setStateFromSeed: in [1, 2^32-1]
| Using create:setStateFromSeeds:
|  Seed[0] must be in [1,Mask[0]];
|  Seed[1] must be in [1,Mask[1]];

Cycles:  | These generators have a single full cycle
| of length Mask1 * Mask2 ~= 2^60 (1.15e18).

Output Quality:  | The authors conducted an exhaustive search among candidate
| Tausworthe generators to find the combination that had the
| best lattice structure. (Single Tausworthe generators do not
| have acceptable properties.) The 3 generators implemented
| here were retained as the best found.

Reference for | Tezuka and L'Ecuyer, op. cit.
output quality:  |

C2TAUSgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        4.078 uS
|   -getFloatSample, getDoubleSample:          5.932 uS
|   -getFatDoubleSample, getLongDoubleSample: 10.920 uS

Relative speed:
| Speed 0.907 (time 1.103) relative to MT19937 getUnsignedSample
```

C2TAUS1gen Documentation

```
Parameters:  | P = 31 so Mask = 2^31-1
(component 1) | S = 12 so PmS = 19
| Q = 13

| P = 29 so Mask = 2^29-1
(component 2) | S = 17 so PmS = 12
```

```
parameters:  | op. cit., table III p. 107.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length ~ 2^60.

Output Quality:  | Good.

Reference for | Tezuka and L'Ecuyer, op.cit.
output quality:  |
```

(./C2MRG3gen.html)

(./Interface.html)

(./C3MWCgen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,2^32-1].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds: each component may have
| a 'state' <= 0xffff and a carry <= (multiplier-1), so that
| for the first component the largest seed allowed is
| (a-1)<<16 + 65535.

Cycles:  | This generator is claimed to be strictly periodic, with a
| period > 2^118. There may be more than one cycle.

Output Quality:  | Good.

Reference for | Marsaglia, op. cit.
output quality:  |

C3MWCgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        6.437 uS
|   -getFloatSample, getDoubleSample:          8.200 uS
|   -getFatDoubleSample, getLongDoubleSample: 15.562 uS

Relative speed:
| Speed 0.574 (time 1.741) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

(./C2TAUS1-3gen.html)

(./Interface.html)

(./C4LCGXgen.html)

C4LCGXgen Class

```
//
// C4LCGXgen --
//   short component based generator with splitting facilities. Recommended.
//
@deftype C4LCGXgen  <CombinedShortSplitGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C4LCGX

Description:  | Combined random generator using 4 (PMM)LGC generators

Reference:  | Pierre L'Ecuyer and Terry H. Andres,
| "A Random Number Generator Based on the Combination
|  of Four LCGs", 17 December 1996.
| To appear in Mathematics and Computers in Simulation. See:
| <http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clcg4.ps>.
| Their name for the generator: 'clcg4'.

Comments:  | This portable generator is based on a backbone generator
| which is a combination of 4 (PMM)LCG generators. It has a
| period length of (m1-1)(m2-1)(m3-1)(m4-1) / 8, or almost
| 2^121 (2.6e36).

| The backbone generator's period can be split up into a
| number of 'virtual generators' (A), each of which can be set
| to access a number of 'segments' (V) of length W, subject to
| the constraint that A * V * W <= 2^120.

| This is possible because the backbone generator has
| 'jump-ahead facilities' -- i.e. a quick method of calculating
| the state of the generator at a given point in its cycle
| without having to actually draw a large number of variates.

Algorithm | state(j) = seed(j)
for each | newstate(j) = (a(j) * oldstate(j)) mod m(j)
generator:  | output(j) = state(j)

Algorithm | output = (state(1) - state(2) + state(3) - state(4)) mod m(0)
for combination:|
```

```
comment:        | Schrage to avoid 32-bit overflow when calculating the
| next state (i.e. using q=m/a and r=m%a as auxiliary
| values). Using 64-bit math (long long integers) is more
| straightforward. Unfortunately, on x86 at least, there is
| a significant speed penalty for doing so.
| The output, however, would be identical. YMMV.

C4LCGXgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       11.688 uS
|   -getFloatSample, getDoubleSample:         13.644 uS
|   -getFatDoubleSample, getLongDoubleSample: 26.154 uS

Relative speed:
| Speed 0.316 (time 3.160) relative to MT19937 getUnsignedSample
--------------- | ------------------------------------------------------------
---

Parameters:  | a[COMPONENTS] = {      45991,     207707,
|      138556,      49689 }
| m[COMPONENTS] = { 2147483647, 2147483543,
|  2147483423, 2147483323 }

Reference for | L'Ecuyer and Andres, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length close to 2^121 (2.6e36).

Output Quality:  | The authors performed an exhaustive search for component
| generators that would yield the best statistical quality.

Reference for | L'Ecuyer and Andres, op. cit.
output quality:  |
```

(./C3MWCgen.html)

(./Interface.html)

LCG1-3gen Classes

```
//
// LCG1, LCG2, LCG3 --
//   single short random number generators,
//   NOT recommended for serious use; these are included for
//   historical reasons (compatibility with earlier releases).
//
@deftype LCGgen      <SingleShortGenerator> @end
@deftype LCG1gen        <LCGgen, CREATABLE> @end
@deftype LCG2gen        <LCGgen, CREATABLE> @end
@deftype LCG3gen        <LCGgen, CREATABLE> @end
```

Use of these generators is NOT Recommended

General Documentation

```
Generator name:  | LCG

Description:  | Linear Congruential Generator

Reference:  | This is a classic generator.
| See for example Donald Knuth,
| The Art of Computer Programming,
| vol. II, "Seminumerical Algorithms."

| state = seed
Algorithm:  | newstate = ((a * oldstate) + c ) mod 2^32
| output = state

Comments:  | This generator relies on controlled overflow at 32 bits.
| This requires that unsigned be a 32bit value that follows
| ANSI C rules.
| Knuth claims that the adder c does not matter much, as
| long as it has no factors in common with the modulus 2^32.


Parameters:  | unsigned int c // adder
| unsigned int a // multiplier
```

LCG2gen Documentation

```
Parameters:  | a =        69,069
| c = 1,013,904,223

Reference for | The multiplier is from Knuth vol II (line 25 of table).
parameters:  | Adder is from Numerical Recipes, in turn from H.W.Lewis

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length m.

Output Quality:  | Low-order bits are poor.

Reference for | Knuth, op.cit. and Numerical Recipes.
output quality:  |
```

LCG3gen Documentation

```
Parameters:  | a =   1,664,525
| c = 152,193,325

Reference for | The multiplier is from Knuth vol II (line 26 of table).
parameters:  | Adder is from Numerical Recipes, in turn from H.W.Lewis

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length m.

Output Quality:  | Low-order bits are poor.

Reference for | Knuth, op.cit. and Numerical Recipes.
output quality:  |
```

(./C4LCGXgen.html)

(./Interface.html)

MRG5-7gen Classes

```
//
// MRG5, MRG6, MRG7 --
//   single long generators recommended for use.
//
@deftype MRGgen      <SingleLongGenerator>  @end
@deftype MRG5gen      <MRGgen, CREATABLE> @end
@deftype MRG6gen      <MRGgen, CREATABLE> @end
@deftype MRG7gen      <MRGgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | MRG

Description:  | Multiple Recursive LCG Generator

Reference:  | Pierre L'Ecuyer, Francois Blouin and Raymond Couture,
| "A Search for Good Multiple Recursive Random Generators."
| See <http://www.iro.umontreal.ca/~lecuyer>.

Algorithm:  | x_n = (a_1*x_n-1 + a_2*x_n-2 ... + a_k*x_n-k) mod m

Comments:  | These generators require k multipliers and k past values
| to be kept. In their paper, the authors investigate MRG's
| of order k from 1 to 7. They provide several sets of
| parameters which they recommend out of a large number that
| were tested. Generally, the quality of the generators
| increases with k.

Implementation:  | The authors discovered that most of the benefit of these
| generators can be had if only a_1 and a_k are nonzero and the
| rest zero; this yields faster execution since only 2 terms
| need be calculated. We implement 4 such generators here,
| all with modulus m close to 2^31.


Parameters:  | unsigned int m // modulus
| unsigned int a1, ak // multipliers
| unsigned int k // the order of the generator

State:  | unsigned int x[k] // state vector
```

```
|   -getFloatSample, getDoubleSample:         12.764 uS
|   -getFatDoubleSample, getLongDoubleSample: 24.627 uS

Relative speed:
| Speed 0.339 (time 2.951) relative to MT19937 getUnsignedSample
```

MRG5gen Documentation

```
Parameters:  | unsigned int k = 5 // lag order
| unsigned int m = 2^31 - 1 // modulus
| unsigned int a1 = 107374182 // multiplier
| unsigned int a5 = 104480 // multiplier

Reference for | L'Ecuyer et al, op. cit., Table 3.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (2^31-1)^5 - 1,
| i.e. 2^154 < cycle < 2^155.

Output Quality:  | Good. Parameters chosen on the basis of theory
| and empirical testing.

Reference for | L'Ecuyer et al, op. cit.
output quality:  |
```

MRG6gen Documentation

```
Parameters:  | unsigned int k = 6 // lag order
| unsigned int m = 2^31 - 1 // modulus
| unsigned int a1 = 177786 // multiplier
| unsigned int a6 = 64654 // multiplier

Reference for | L'Ecuyer et al, op. cit., Table 3.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (2^31-1)^6 - 1,
| i.e. 2^185 < cycle < 2^186.

Output Quality:  | Good. Parameters chosen on the basis of theory
| and empirical testing.

Reference for | L'Ecuyer et al, op. cit.
output quality:  |
```

MT19937gen Class

```
//
// MT19937 --
//   single *very* long generator recommended for use.
//
@deftype MT19937gen  <SingleLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | MT19937

Description:  | Twisted GFSR generator

Reference:  | Makoto Matsumoto and ??? Nishimura,
| code posted to the pLab web site, xx/xx/97.
| Site: <http://random.mat.sbg.ac.at>
| Code: <ftp://random.mat.sbg.ac.at/pub/data/mt19937b-int.c>
| Article reportedly submitted to ACM TOMACS.

|
Algorithm:  |
|

Comments:  |

Implementation:  |

Parameters:  | unsigned int N // size of state array
| unsigned int M // lag
| unsigned int a // "Magic numbers"
| unsigned int b,c //    "     "
| unsigned int u,s,t,l //    "     "

State:  | unsigned int state[N] // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)
```

(./MRG5-7gen.html)

(./Interface.html)

(./MWCA-Bgen.html)

---

*Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)*

Last modified: 1998-01-26

```
Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds: each component may have
| a 'state' <= 65535 and a 'carry' <= (multiplier-1). Thus
| the first component has a maximum seed of
| ( (a-1) << 16) + 65535.

Cycles:  | This generator is claimed to be strictly periodic, with a
| period > 2^59. (There's possibly two such cycles.)

Output Quality:  | Good.

Reference for | Marsaglia, op. cit.
output quality:  |

MWCgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       3.399 uS
|   -getFloatSample, getDoubleSample:         5.131 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.447 uS

Relative speed:
| Speed 1.088 (time 0.919) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

MWCBgen Documentation

```
Generator name:  | MWCB

Description:  | Multiply-With-Carry generator

Reference:  | Article posted to Usenet in 1994 by George Marsaglia,
| "The Mother of all Random Number Generators."
| Source found at <ftp://ftp.taygeta.com/pub/c/mother.c>
| Also document /papers/mwc1.ps on his Diehard CD-ROM,
| found at <http://www.hku.hk/internet/randomCD.html>.

Algorithm:  | K = a * (K & 65535) + (K>>16);
| J = b * (J & 65535) + (J>>16);
| return (K<<16) + (J&65535);

Comments:  | This generator is a combination of 2 16-bit
| Multiply-With-Carry generators.

Implementation:  | This generator implements an alternate manner of
| conjoining the two components (differs from MWCA).
```

(./Interface.html)

(./PMMLCG1-9gen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

| The choice of a good multiplier a is crucial.

Parameters: | unsigned int m // modulus
| unsigned int a // multiplier

State: | unsigned int state // a "state vector" of length 1

Output types: | unsigned int (we return the current state-1).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range: | The basic algorithm yields output in the range [1,m-1].
| We subtract 1 to return output in the range [0,m-2]
| (i.e. [0,2147483645].)

Valid seeds: | create:setStateFromSeed: [1, 2^32-1]
| create:setStateFromSeeds: [1, m-1]

Cycles: | With properly chosen parameters, these generators
| have a single full cycle of length (m-1).
| (State=0 is an absorbing state which will not normally
| occur; we avoid setting the starting state to 0.)

Output Quality: | All bits should be safe.
| (I.e. low order bits should not have small cycles.)
| With m prime, this generator is better than standard LCG.

Reference for | Park and Miller, op. cit.
output quality: |

Implementation | The code for -getUnsignedMax uses a trick devised by
comment: | Schrage to avoid 32-bit overflow when calculating the
| next state (i.e. using q=m/a and r=m%a as auxiliary
| values). Using 64-bit math (long long integers) is more
| straightforward. Unfortunagely, on x86 at least, there
| is a 35% speed penalty for doing so. The output, however,
| is identical. YMMV.

PMMLCGgen | On a 486/66, the following approximate measures were obtained:
speed: | -getUnsignedSample: 4.715 uS
| -getFloatSample, getDoubleSample: 6.500 uS
| -getFatDoubleSample, getLongDoubleSample: 12.152 uS

Relative speed:
| Speed 0.792 (time 1.263) relative to MT19937 getUnsignedSample

```
choice of  |
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.
| These parameters are not as well tested as PMMLCG1,
| but there's reason to believe they may be better.

Reference for | Park and Miller, op. cit.
output quality:  |
```

PMMLCG4gen Documentation

```
Parameters:  | a =        45,991
| m = 2,147,483,647 (2^31-1)

Reference for | Pierre L'Ecuyer and Terry H. Andres,
choice of  | "A Random Number Generator Based on the Combination
parameters:  | of Four LCGs", unpubl.

| This is one of the component generators of their CLCG4.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.

Reference for | L'Ecuyer and Andres, op.cit.
output quality:  |
```

PMMLCG5gen Documentation

```
Parameters:  | a =       207,707
| m = 2,147,483,543 (2^31-105)

Reference for | Pierre L'Ecuyer and Terry H. Andres,
choice of  | "A Random Number Generator Based on the Combination
parameters:  | of Four LCGs", unpubl.

| This is one of the component generators of their CLCG4.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).
```

PMMLCG8gen Documentation

```
Parameters:  | a =        40,014
| m = 2,147,483,563 (2^31-85)

Reference for | Pierre L'Ecuyer and Serge Cote,
choice of  | "Implementing a Random Number Package with Splitting
parameters:  | Facilities," ACM Trans. Math. Softw. 17 no 1 (March 1991).

| This is one of the component generators of their generator.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.

Reference for | L'Ecuyer and Cote, op.cit.
output quality:  |
```

PMMLCG9gen Documentation

```
Parameters:  | a =        40,692
| m = 2,147,483,399 (2^31-249)

Reference for | Pierre L'Ecuyer and Serge Cote,
choice of  | "Implementing a Random Number Package with Splitting
parameters:  | Facilities," ACM Trans. Math. Softw. 17 no 1 (March 1991).

| This is one of the component generators of their generator.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.

Reference for | L'Ecuyer and Cote, op.cit.
output quality:  |
```

(./MWCA-Bgen.html)

PSWBgen Class

```
//
// PSWB --
//   single long generator recommended for use.
//
@deftype PSWBgen   <SingleLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | PSWB

Description:  | Subtract-with-Borrow Congruential Generator
|    with Prime Modulus

Reference:  | George Marsaglia and Arif Zaman,
| "A New Class of Random Number Generators",
| Annals of Applied Probability vol 1 no 3 (1991), 462-480.
| SWB is an improvement on the Lagged Fibonacci generators.

| PSWB is an improvement on SWB in that the use of a prime
| modulus guarantees a single full cycle.
| It's slower, of course.

| state vector = f(seed)
Algorithm:  | state_n = (state_(n-s) - state_(n-r) - carry) mod m; r > s
| output = state_n

| These generators use a basic algorithm of the form
| X_n = f(X_(n-r),X_(n-s)) mod m; r>s
| The function f is typically xor, addition, subtraction,
| multiplication or subtraction with carry. It uses
| simpler math than a basic LCG, but keeps a larger state.

Implementation:  | The state vector contains the generator's r most recent
| outputs (X_n .. X_(n-r)). This vector is initialized from
| an LCG generator using one 32-bit seed value. After
| initialization we throw away r*5 numbers to 'prime' the
| generator (i.e. make sure we have entered a closed cycle.)

Parameters:  | unsigned int r // long lag
| unsigned int s // short lag
```

Cycles: | With the chosen parameters, this generator
| has a single cycle of length close to 10^414.

Output Quality: | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality: |

(./PMMLCG1-9gen.html)

(./Interface.html)

(./RWC28gen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
| Using create:setStateFromSeeds: x1 and x2: [1, 2^32-1]
| The carry can only be [1,a-1].

Cycles:  | This generator is claimed to have a period > 2^92.
| It may have rho sequences up to length r=2.
| It may have multiple cycles of the stated period length.

Output Quality:  | Good.

Reference for | Marsaglia, op. cit.
output quality:  |

RWC2gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        7.445 uS
|   -getFloatSample, getDoubleSample:          8.845 uS
|   -getFatDoubleSample, getLongDoubleSample: 17.338 uS

Relative speed:
| Speed 0.497 (time 2.013) relative to MT19937 getUnsignedSample
```

RWC8gen Documentation

```
Generator name:  | RWC8

Description:  | 8-lag Recursion-With-Carry generator
| ("The Mother of all RNG's")

Reference:  | Article posted to Usenet in 1994 by George Marsaglia,
| "The Mother of all Random Number Generators."
| Source found at <ftp://ftp.taygeta.com/pub/c/mother.c>
| Also document /papers/mwc1.ps on his Diehard CD-ROM,
| found at <http://www.hku.hk/internet/randomCD.html>.

Algorithm:  | x = sum(ai*xi) + carry mod 2^16
| y = sum(bi*yi) + carry mod 2^16
| return ((x<<16)+y)

Comments:  | This generator is a combination of 2 16-bit
| 8-lag Recursion-With-Carry generators.

Parameters:  | unsigned int a[8] =
| { 1941, 1860, 1812, 1776, 1492, 1215, 1066, 12013 }

| unsigned int b[8] =
| { 1111, 2222, 3333, 4444, 5555, 6666, 7777,  9272 }
```

(./SCGgen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
| unsigned int s // short lag
| unsigned int m // modulus

State:  | unsigned int state[r] // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,10^9-1] may be returned
| (i.e. [0,999999999].)

Valid seeds:  | Using create:setStateFromSeed:  [1,2^32-1]
| Using create:setStateFromSeeds: [1,999999998]

Cycles:  | With properly chosen parameters, these generators
| have a cycle of length about 2^r.

Output Quality:  | Low order bits are poor, and there are other problems.
| SWB is a better generator in the same class, and it is
| only slightly slower.

Reference for | Knuth, op.cit.
output quality:  |

SCGgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       3.328 uS
|   -getFloatSample, getDoubleSample:         5.058 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.274 uS

Relative speed:
| Speed 1.111 (time 0.900) relative to MT19937 getUnsignedSample
--------------- | -----------------------------------------------------------
---

Parameters:  | r = 55
| s = 24
| m = 10^9

Reference for |
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length ~ 2^r.
```

SWB1-3gen Classes

```
//
// SWB1, SWB2, SWB3 --
//   single long generators recommended for use.
//
@deftype SWBgen   <SingleLongGenerator> @end
@deftype SWB1gen    <SWBgen, CREATABLE> @end
@deftype SWB2gen    <SWBgen, CREATABLE> @end
@deftype SWB3gen    <SWBgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | SWB

Description:  | Subtract-with-Borrow Congruential Generator

Reference:  | George Marsaglia and Arif Zaman,
| "A New Class of Random Number Generators",
| Annals of Applied Probability vol 1 no 3 (1991), 462-480.
| SWB is an improvement on the Lagged Fibonacci generators.

| state vector = f(seed)
Algorithm:  | state_n = (state_(n-s) - state_(n-r) - carry) mod 2^32; r > s
| output = state_n

| These generators use a basic algorithm of the form
| X_n = f(X_(n-r),X_(n-s)) mod m; r>s
| The function f is typically xor, addition, subtraction,
| multiplication or subtraction with carry. It uses
| simpler math than a basic LCG, but keeps a larger state.

Note:  | This generator relies on controlled overflow at 32 bits.
| This requires that unsigned be a 32bit value that follows
| ANSI C rules.

Implementation:  | The state vector contains the generator's r most recent
| outputs (X_n .. X_(n-r)). This vector is initialized from
| an LCG generator using one 32-bit seed value. After
| initialization we throw away r*5 numbers to 'prime' the
| generator (i.e. make sure we have entered a closed cycle.)
```

```
| s = 24

Reference for | Marsaglia, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has 64 cycles of length 10^354

Output Quality:  | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality:  |
```

SWB2gen Documentation

```
Parameters:  | r = 24
| s = 19

Reference for | Marsaglia, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has 1536 cycles of length 10^228

Output Quality:  | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality:  |
```

SWB3gen Documentation

```
Parameters:  | r = 21
| s =  6

Reference for | Marsaglia, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has 192 cycles of length 10^200

Output Quality:  | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality:  |
```

TGFSRgen Classes

```
//
// TT403, TT775, TT800 --
//   single long generators recommended for use.
//
@deftype TGFSRgen    <SingleLongGenerator>  @end
@deftype TT403gen     <TGFSRgen, CREATABLE> @end
@deftype TT775gen     <TGFSRgen, CREATABLE> @end
@deftype TT800gen     <TGFSRgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | TGFSR

Description:  | Twisted GFSR generator

Reference:  | Makoto Matsumoto and Yoshiharu Kurita,
| "Twisted GFSR Generators II".
| ACM TOMACS, vol 4 no 3 July 1994 pp. 254-266.
| Amended by M. Matsumoto, July 8 1996.

|
Algorithm:  |
|

Comments:  |

Implementation:  |

Parameters:  | unsigned int w // word length
| unsigned int N // size of state array
| unsigned int M // lag
| unsigned int s // 'magic vector'
| unsigned int t //    "     "
| unsigned int b //    "     "
| unsigned int c //    "     "
| unsigned int a //    "     "

State:  | unsigned int state[N] // state vector

Output types:  | unsigned int (we return the current state).
```

```
| and empirical testing.

Reference for | Matsumoto and Kurita, op. cit.
output quality:  |
```

TT775gen Documentation

```
Parameters:  | unsigned int w=31 // word length
| unsigned int N=25 // size of state array
| unsigned int M=8 // lag
| unsigned int s=6 // 'magic vector'
| unsigned int t=14 //    "     "
| unsigned int b=0x1abd5900 //    "     "
| unsigned int c=0x776a0000 //    "     "
| unsigned int a=0x6c6cb38c //    "     "

Reference for | Matsumoto and Kurita, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^775 -1.

Output Quality:  | Good. Parameters chosen on the basis of theory
| and empirical testing.

Reference for | Matsumoto and Kurita, op. cit.
output quality:  |
```

TT800gen Documentation

```
Parameters:  | unsigned int w=32 // word length
| unsigned int N=25 // size of state array
| unsigned int M=7 // lag
| unsigned int s=7 // 'magic vector'
| unsigned int t=15 //    "     "
| unsigned int b=0x2b5b2500 //    "     "
| unsigned int c=0xdb8b0000 //    "     "
| unsigned int a=0x8ebfd028 //    "     "

Reference for | Matsumoto and Kurita, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^800 -1.
```

RandomBitDistribution Class

```
//
// RandomBitDistribution --
//   A generator that returns uniformly distributed single bit values
//   (i.e. fair coin tosses).
//
@deftype RandomBitDist <BooleanDistribution, CREATABLE>
USING
- (BOOL) getCoinToss;
@end
```

Documentation

```
Name:  | RandomBitDistribution

Description:  | Distribution returning YES and NO with equal probability

Algorithm:  | Extract a middle bit from an unsigned random number

Reference:  | (none)

Parameters:  | (none)

State:  | unsigned long long int currentCount

RandomBitDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getBooleanSample:                         5.656 uS
|  -average # of generator calls per variate: 1.000 (U)
|  -distribution net time:                    1.958 uS

Relative speed:
| Speed 0.654 (time 1.529) relative to MT19937 getUnsignedSample
--------------- . -------------------------------------------------------------
--
```

BernoulliDistribution Class

```
//
// BernoulliDistribution --
distribution returning YES with given probability p
//
@deftype BernoulliDist <BooleanDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setProbability: (double) p;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setProbability: (double) p;
USING
- setProbability: (double) p;
- (double) getProbability;
- (BOOL)        getSampleWithProbability: (double) p;
@end
```

Documentation

```
Name:  | BernoulliDistribution

Description:  | Distribution returning binary value YES with probability p

Algorithm:  | Rejection method

Reference:  | Code contributed by Barry McMullin

Parameters:  | double Probability

State:  | unsigned long long int currentCount

BernoulliDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getBooleanSample:                         8.156 uS
|  -average # of generator calls per variate: 1.000 (tD)
|  -distribution net time:                    2.654 uS

Relative speed:
| Speed 0.453 (time 2.206) relative to MT19937 getUnsignedSample
```

UniformIntegerDist Class

```
//
// UniformInteger --
//   A generator of integral values uniformly distributed across a closed
//   interval [min,max]. (The interval includes both its endpoints.)
//   Setting minValue == maxValue is allowed (and returns minValue).
//
@deftype UniformIntegerDist <IntegerDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setIntegerMin: (int) minValue setMax: (int) maxValue;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setIntegerMin: (int) minValue setMax: (int) maxValue;
USING
- setIntegerMin: (int) minValue setMax: (int) maxValue;
- (int) getIntegerMin;
- (int) getIntegerMax;
- (int) getIntegerWithMin: (int) minValue withMax: (int) maxValue;
@end
```

Documentation

```
Name:  | UniformInteger

Description:  | Distribution returning integers over an interval [min,max]

Algorithm:  | Transformation of uniform unsigneds [0,unsignedMax]

Reference:  | (none)

Parameters:  | integer minValue
| integer maxValue

State:  | unsigned long long int currentCount

Implementation  | Setting minValue=maxValue is allowed.
comment:        | Then minValue is retured "with probability 1".

UniformIntegerDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:
```

UniformUnsignedDist Class

```
//
// UniformUnsigned --
//   A generator of non-negative integral values uniformly distributed across
//   a closed interval [min,max].  (The interval in-
cludes both its endpoints.)
//   Setting minValue == maxValue is allowed (and returns minValue).
//
@deftype UniformUnsignedDist <UnsignedDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setUnsignedMin: (unsigned) minValue setMax: (un-
signed) maxValue;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setUnsignedMin: (unsigned) minValue setMax: (un-
signed) maxValue;
USING
- setUnsignedMin: (unsigned) minValue setMax: (unsigned) maxValue;
- (unsigned) getUnsignedMin;
- (unsigned) getUnsignedMax;
- (unsigned) getUnsignedWithMin: (unsigned) minVal
           withMax: (unsigned) maxVal;
@end
```

Documentation

```
Name:  | UniformUnsigned

Description:  | Distribution returning unsigneds over an interval [min,max]

Algorithm:  | Transformation of uniform unsigned [0,unsignedMax]

Reference:  | (none)

Parameters:  | unsigned minValue
| unsigned maxValue

State:  | unsigned long long int currentCount

Implementation  | Setting minValue=maxValue is allowed.
```

UniformDoubleDist Class

```
//
// UniformDouble --
//   A generator of floating point values uniformly distributed
//   across a half-open interval [min,max). (The interval includes
//   the lower endpoint but excludes the upper endpoint.)
//   NOTE: Setting minValue == maxValue is allowed (and returns minValue).
//
@deftype UniformDoubleDist <DoubleDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setDoubleMin: (double) minValue setMax: (double) maxValue;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setDoubleMin: (double) minValue setMax: (double) maxValue;
USING
- setDoubleMin: (double) minValue setMax: (double) maxValue;
- (double) getDoubleMin;
- (double) getDoubleMax;
- (double) getDoubleWithMin: (double) minValue withMax: (double) maxValue;
@end
```

Documentation

```
Name:  | UniformDouble

Description:  | Distribution returning doubles over an interval [min,max)

Algorithm:  | Transformation of uniform double [0.0,1.0)

Reference:  | (none)

Parameters:  | double minValue
| double maxValue

State:  | unsigned long long int currentCount

Implementation | Setting minValue=maxValue is allowed.
comment:  | Then minValue is returned "with probability 1".

UniformDoubleDist | On a 486/66, the following measures were obtained,
```

ExponentialDistribution Class

```
//
// ExponentialDistribution -- a well-
known continuous probability distribution
//
@deftype ExponentialDist <DoubleDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setMean: (double) mean;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setMean: (double) mean;
USING
- setMean: (double) mean;
- (double) getMean;
- (double) getSampleWithMean: (double) mean ;
@end
```

Documentation

```
Name:  | ExponentialDistribution

Description:  | Exponential distribution returning double values

Algorithm:  | Transformation method

Reference:  | Russell 1992

Parameters:  | double Mean

State:  | unsigned long long int currentCount

ExponentialDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          20.905 uS
|  -average # of generator calls per variate:  1.000 (D)
|  -distribution net time:                    10.746 uS

Relative speed:
| Speed 0.177 (time 5.653) relative to MT19937 getUnsignedSample
```

GammaDistribution Class

```
//
// GammaDistribution -- a well-known continuous probability distribution
//
@deftype GammaDist <DoubleDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
setAlpha: (double) alpha setBeta: (double) beta;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
setAlpha: (double) alpha setBeta: (double) beta;
USING
- setAlpha: (double) alpha setBeta: (double) beta;
- (double) getAlpha;
- (double) getBeta;
- (double) getSampleWithAlpha: (double) alpha
withBeta: (double) beta;
@end
```

Documentation

```
Name:  | GammaDistribution

Description:  | Gamma distribution returning double values

Algorithm:  | Transformation, rejection and xxx methods.

Reference:  | Watkins 1994.

Parameters:  | double Alpha
| double Beta

State:  | unsigned long long int currentCount

GammaDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          119.424 uS
|  -average # of generator calls per variate:   6.224 (D)
|  -distribution net time:                     56.194 uS
```

Normal and LogNormalDistribution Classes

```
//
// NormalDistribution -- a well-known continuous probability distribution
// LogNormalDistribution -- also well-known ...
//
@deftype NormalDist    <Normal, CREATABLE> @end
@deftype LogNormalDist <Normal, CREATABLE> @end
```

NormalDist Documentation

```
Name:  | NormalDistribution

Description:  | Normal (Gaussian) distribution returning double values

Algorithm:  | The Box-Muller method

Reference:  | W.H. Press et al, "Numerical Recipes in C" (2nd ed.).
| Cambridge University Press, 1992.

Parameters:  | double Mean
| double Variance

State:  | double stored_value
| BOOL stored
| unsigned long long int currentCount

NormalDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          24.729 uS
|  -average # of generator calls per variate:  1.273 (D)
|  -distribution net time:                    11.797 uS

Relative speed:
| Speed 0.150 (time 6.687) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

LogNormalDist Documentation

(RandomBitDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

We then configure the generator as having a number (A) of 'vir-
tual generators',
each of which can address a number (2^v) of sub-segments of length 2^w.
These parameters (A,v,w) are user selectable when the generator is created.
(As an example, for C4LCGXgen the default values are A=128, v=31, w=41.)
The advantage is that the subsegments act as statistically independent
streams of random numbers.

In addition to the -getUnsignedSample method, generators now also supply
floating point output in the range [0.0,1.0), in the form of these methods:

```
   -(float)       getFloatSample; // using 1 unsigned value
   -(double)      getThinDoubleSample; // using 1 unsigned value
   -(double)      getDoubleSample; // using 2 unsigned values
   -(long double) getLongDoubleSample; // using 2 unsigned values
```

The last method is not portable across architectures, since the length of a
long double varies between machines.

Generators may now be started with a single seed, *or* with a vector of seeds
whose length is generator dependent. (PMMLCG requires 1 integer for a
seed, while MT19937 needs 624 of them.)

Generators now remember what seed values they were started with. They also
count how many variates they have delivered (i.e., how many calls to
-getUnsignedSample they have serviced.)

There are a few arbitrary seed values, DEFAULTSEED, DEFAULT-
SEED1, DEFAULTSEED2,
DEFAULTSEED3, DEFAULTSEED4 defined. There is the value RANDOMSEED, which will
be different every time it is invoked. And there is value START-
SEED, which will
by default equal DEFAULTSEED, but will instead be equal to RANDOMSEED if you
start your program with the '-varySeed' command line parameter.

The generators have gained a new creation method, '+createWithDe-
faults: aZone',
which creates the generator and initializes it with STARTSEED. Split
generators get default values for A,v,w.

CHANGES since v. 0.6:

The generator classes have changed names to where they all end in '-gen'.
A simple search-and-replace in your code will get you up and running again.

(Or perhaps you'll want to try one of the new generators?)

This allocates the object and initializes it with your seed value. If the object actually requires a vector of seed values to fill the state, this method generates the rest of the values needed using an inline PMMLCG generator.

You can find out later what seed value was used to initialize the generator:

```
myUnsigned = [ myGenerator getInitialSeed ];
```

And you can find out what the largest valid seed value is by calling

```
myUnsigned = [ myGenerator getMaxSeedValue ];
```

(For the generators defined in v. 0.7, this value is 2^32-1 for all of them. The seed may not be 0.)

You may reset the generator's state at any time using this method:

```
[ myGenerator setStateFromSeed: mySeedValue ];
```

(c) using a vector of seed values:

Assume we have defined a fixed array at compile time:

```
unsigned int mySeedVector [vectorLength];
```

Then we can do this:

```
   myGenerator = [ PSWBgen create: [self getZone]
setStateFromSeeds: mySeedVector ];
```

You can find out how many seed values are required by asking

```
myUnsigned = [ myGenerator lengthOfSeedVector ];
```

(Obviously, you must first successfully have created the object to do this, for example using createWithDefaults!)

And we allocate the seed vector dynamically this way:

```
unsigned int *mySeedVector;
mySeedVector = [[self getZone] alloc: [ myGenerator lengthOfSeedVector]];
```

You can find out what vector of seed values was used to initial-ize the object:

```
unsigned int *myVector;
```

(e) generator output:

You obtain successive pseudorandom numbers from a generator by calling

```
   myUnsignedValue = [ myGenerator getUnsignedSample ];
```

The largest value thus returned can be found by asking

```
   myUnsignedValue = [ myGenerator getUnsignedMax ];
```

(The smallest value returned is always 0.)

If you would rather have floating point output in the range [0.0,1.0),
you call one of these:

```
// Using 1 unsigned value to fill the mantissa:

   myFloatValue  = [ myGenerator getFloatSample ];
   myDoubleValue = [ myGenerator getThinDoubleSample ];

// Using 2 unsigned values to fill the mantissa:

   myDoubleValue     = [ myGenerator getDoubleSample ];
   myLongDoubleValue = [ myGenerator getLongDoubleSample ];
```

NOTE that the last method is not portable, because the size of a long double
varies and hence the precision varies between architectures.

Finally, you can obtain a count of how many variates have been generated:

```
   myLongLongInt = [ myGenerator getCurrentCount ];
```

(currentCount is an unsigned long long int, which counts up to 2^64.)

SPLIT generators:

A 'split' generator requires us to specify the configuration (A,v,w)
at create time:

```
   myGenerator = [ C4LCGXgen create: [self getZone]
setA: 64 setv: 20 setw: 76
setStateFromSeed: mySeedValue ];
```

```
   [ myGenerator restartGenerator: myVgen ];
   [ myGenerator advanceGenerator: myVgen ];
   [ myGenerator jumpGenerator:    myVgen    toSegment: myLongLongIntValue ];
```

InternalState methods common to simple and split generators:

```
   // Print (most of) the object's state data to a stream:
   [ myNormalDist describe: myStream ];
```

The stream myStream may be created thus:

```
id myStream = [ OutStream create: [self getZone] setFileStream: stdout ]; or
id myStream = [ OutStream create: [self getZone] setFileStream: stderr ];

   // Get the (class) name of the object:
   myString = [ myNormalDist getName ];

   // Get the object's 'magic number', used by putStateInto / setStateFrom:
   myUnsigned = [ myNormalDist getMagic ];
```

SAVING AND RESETTING STATE:

You may save, and later restore, the internal state of a generator using these methods:

```
   // Get the size of the memory buffer needed by put-
StateInto / setStateFrom:
   myUnsigned = [ myGenerator getStateSize ];

   // Extract the generator's state data into your memory buffer:
   [ myGenerator putStateInto: myBuffer ];

   // Set the generator's state from data in a memory buffer:
   [ myGenerator setStateFrom: myBuffer ];
```

To illustrate, assume the following data definitions:

```
   FILE *myFile;
   const char *myFileName = "MyGenFile.bin"; // or whatever
   int stateSizeG;
   id stateBufG;
   int status;
```

```
    // Close the file:
    status = fclose(myFile);
    if (status) { }; // error on close

    // Ask the generator set its state from the buffer data:
    [ myGenerator setStateFrom: (void *) stateBufG ];

    // Free the memory allocated to the buffer:
    [[self getZone] free: stateBufG];
```

TESTING GENERATORS:

Since v. 0.6 we have done some rudimentary statistical testing of the implemented generators, using Marsaglia's Diehard tests and the ENT tests.

The results of these tests are summarized in document /random/docs/doc.quality.generators, where test results as well as period length, state size and execution times are listed. You can use these data to select a generator that suits your simulation.

Some brief comments:

a) the tests show that old generators SCG and LCG are of poor quality and should be avoided.

b) the lagged-Fibonacci based generators (ACG, SWB, PSWB) all fail Diehard's 'Birthday spacings test', for reasons having to do with their lattice structure.
These generators are only conditionally recommended.

c) The rest of the 32-bit generators (i.e. generators that fill all 32 bits of an unsigned int) pass all tests, and are recommended at this time.

(Note that while a test may show that a generator is bad, passing a number of tests does not prove that a generator is good!)

d) The 31-bit generators all fail the same set of tests. Some of these cannot be passed by a generator whose output has a 'stuck' bit.
Until I clear up with Prof. Marsaglia how to interpret these results, I believe all the 31-bit generators are in the 'recommended' category.

However, a cautionary note: while the PMMLCG generators pass the tests, they have a very short period (less than 2^31) and should only be used for 'toy' simulations. You don't want your generator(s) to 'go around' and start repeating themselves !

UniformIntegerDist and UniformUnsignedDist now allow you to set parameter minValue equal to maxValue. In this case that value is returned every time.

UniformDoubleDist also allows this, even if the set [x,x) is mathematically suspect ...

NormalDist and LogNormalDist now allow you to specify zero Variance, in which case the values returned are the Mean and exp(Mean) respectively.

CHANGES since v. 0.6:

The distribution classes have changed names to where they all end in 'Dist'. A simple search-and-replace in your code will get you back up and running.

The strong distinction between 'frozen' and 'un-frozen' distribution objects in v. 0.6 has been softened considerably. You may now set and reset the default parameters as often as you wish, and you may make calls for variates with given parameters even if different default parameters have been set.

The generation of uniform(0,1) floating point values has been moved from the distribution objects into the generator objects. Thus, if all you need is a uniform(0,1) double, you have no need of a distribution but can get what you desire from a generator.

Note that the generators fill the mantissa of a double from two 32-bit unsigned values in a different manner from v. 0.6 distributions, so output will be a bit different in the new version.

A bug in LogNormalDist has been fixed.

The 'getState:' method has been named 'putStateInto: (void *) buffer', and the 'setState:' method is now 'setStateFrom: (void *) buffer. A quick search-and-replace fixes things in your code.

But note: these methods have also changed somewhat, as has the size of the data being saved. As a result, v. 0.7 distributions will refuse to 'setStateFrom' data saved by v. 0.6 objects.

There should be fewer changes like this in the next release.

USAGE notes:

You can create the generator at the same time as the distribution:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: [TT800gen create: [self getZone]
setStateFromSeed: 34453]         ];
```

(c) Without default parameters, using a split generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySplitGenerator
setVirtualGenerator: 7 ];
```

or perhaps

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: [C4LCGXgen createWithDefaults: [self getZone]]
setVirtualGenerator: 99 ];
```

A split generator can be thought of as comprising a set of virtual generators (streams of random numbers), and a distribution object must be 'connected' to one of these streams. You cannot re-assign the generator or the virtual generator after a distribution object has been created.

In all these cases (a) - (c), when we want to obtain a random variate from this distribution object we need to specify the parameters:

```
   myDouble = [ myNormalDist getSampleWithMean: 3.3 withVariance: 1.7];
```

You can use different parameters for every call. (And you can use this call even if default parameters have been set.)

(d) With default parameters, using a simple generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySimpleGenerator
setMean: 7.6 setVariance: 1.2 ];
```

(e) With default parameters, using a split generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySplitGenerator
setVirtualGenerator: 33
setMean: 3.2 setVariance: 2.1 ];
```

In these cases, we do not need to specify parameters to get a random number:

```
   // Print (most of) the object's state data to a stream:
   [ myNormalDist describe: myStream ];
```

The stream myStream may be created thus:

```
id myStream = [ OutStream create: [self getZone] setFileStream: stdout ]; or
id myStream = [ OutStream create: [self getZone] setFileStream: stderr ];

   // Get the (class) name of the object:
   myString = [ myNormalDist getName ];

   // Get the object's 'magic number', used by putStateInto / setStateFrom:
   myUnsigned = [ myNormalDist getMagic ];
```

You may save, and later restore, the internal state of a distribution object using these methods:

```
   // Get the size of the memory buffer needed by put-
StateInto / setStateFrom:
   myUnsigned = [ myNormalDist getStateSize ];

   // Extract the distribution's state data into your memory buffer:
   [ myNormalDist putStateInto: myBuffer ];

   // Set the distribution's state from data in a memory buffer:
   [ myNormalDist setStateFrom: myBuffer ];
```

To illustrate, assume the following data definitions:

```
   FILE *myFile;
   const char *myFileName = "MyDistFile.bin"; // or whatever
   int stateSizeD;
   id stateBufD;
   int status;
```

The following code shows how to save an object's state to disk:
(You should add your own code to deal with disk file errors,
either aborting or printing out error messages.)

```
   // Ask how big a buffer we need:
   stateSizeD = [ myNormalDist getStateSize ];

   // Allocate memory for the buffer:
   stateBufD  = [[self getZone] alloc: stateSizeD];

   // Ask the distribution object to put state data into the buffer:
```

UTILITY OBJECTS PROVIDED:

The following objects have been defined in <random/random.m>, and are immediately accessible from anywhere in your program:

```
id <MT19937>              randomGenerator;
id <UniformIntegerDist>    uniformIntRand;
id <UniformUnsignedDist>   uniformUnsRand;
id <UniformDoubleDist>     uniformDblRand;
```

The 3 distribution objects all draw their random numbers from the MT19937 generator, which has a period of 2^19937 (10^6001) and is quite fast.

(Interface.html)

(../random.html)

(AdvancedUsage.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Random Library: Interface Design Notes

Never has a library interface been debated as this one was (and still is). This document will attempt to provide some of the rational for the decisions made by echoing some of the controversial issues covered in the debates on the interface. (These discussions overlapped into implementation. But, the point, here, is to give people an idea of the complexity of the issues involved in useful pseudo-random number use in simulation.)

Issues:

- Different classes for generators or variants? Ref: http://www.sela.co.il:8080/swtrain/new/shai/tgp.ps
- Include non-recommended generators?
- Seed vs. State for generators?
- What is "state" and how do we handle it?
- Larger issue of debugging support via an object reporting its own configuration via a special method and how this relates to object state reporting and object saving.
- Separate classes for separate distribution types (e.g. double, integer, unsigned :: continuous/discrete)?
- Problems with the transformation from generator output to distribution output.
- What precision should be supported?
- Should the generator underlying a distribution be visible through the distribution?
- Efficiency!
- What distributions to provide?
- What generators to provide?
- Open vs. Closed intervals?
- Programmed default generators, seeds, and states?
- Programming by committee? (a.k.a. Software process!)
- Shorthand create methods?
- Potential phase optimizations?
- Random number streams, syncing limit cycles, and proper simulation vs. code debugging. (This is a HOT ONE.)
- Library interchangeability.

These are just the questions and issues. Some addition should follow (when there's time) addressing each issue and giving rationale for what we chose. But, that's a big job.

Random Library: Implementation Notes

The generators supplied with this release have been subjected to statistical testing using George Marsaglia's Diehard tests as well as John Walker's entropy tests (ENT). The results of these tests are summarized in the document $(SWARMHOME)/src/random/docs/doc.quality.generators. Other properties of the generators are summarized in $(SWARMHOME)/src/random/docs/generators.table. Some notes on how to choose a generator for your simulation are found in $(SWARMHOME)/src/random/docs/CHOOSING.A.GENERATOR.

The ENT test is included in the tarball of test programs found at the Swarm web site. The Diehard tests are copyright and hence are not, but they can be downloaded from the web at http://www.hku.hk/internet/randomCD.html (http://www.hku.hk/internet/randomCD.html)
.

The distribution objects have not been statistically tested.

(./InterfaceDesign.html)

(../random.html)

(../random.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10