radically, then you can suspect the generator. If they don't, well, the generator still might be faulty.

There is also a tarball of test programs for the random number library on the Swarm ftp site:
SwarmTests-0.7.tar.gz
(ftp://ftp2.santafe.edu/pub/swarm/RandomTests-0.7.tar.gz)

Documentation

The *random* library is defined and documented according to the `Library Interface Conventions`

(defobj/library.html)
. The following standard sections of documentation are available:

- Interface Reference
(random/Interface.html)
- Usage Guide
(random/UsageGuide.html)
- Advanced Usage Guide
(random/AdvancedUsage.html)
- Interface Design Notes
(random/InterfaceDesign.html)
- Implementation Notes
(random/Implementation.html)

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

Last modified: 1998-01-10

Literature references for implemented objects

doc.generators.table

Summary descriptive data for generators

doc.generators.maxvalue

Values returned by [myGen getUnsignedSample]

doc.quality.generators

Results of statistical and timing tests

doc.timings

Timing results for generators and distributions

methods.SimpleGenerators.h

Methods implemented by a typical Simple generator

methods.SplitGenerators.h

Methods implemented by a typical Split generator

Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin. --Jon von Neuman

---

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

Last modified: 1998-01-08

Random Library: Interface Reference

The *random* library uses the module definition conventions
(../defobj/module.html)
established by the *defobj* library
(../defobj.html)
. It also depends on standard supertypes and classes defined by this library.

Dependencies

Following are the other header files imported by <random.h>:

```
#import <defobj.h>
#import <random/generators.h>
#import <random/distributions.h>
#import <random/RandomDefs.h>
#import <random/RandomVars.h>
```

The *defobj* library interface is included to provide the basic object support. `RandomDefs.h` contains some C preprocessor macros and typedefs used in the library.

Object Types (with links to reference pages)

Contents:

The following shows the protocol inheritance trees and object definitions for generators and distributions. The complete set of methods implemented by these objects have been subdivided into sub-protocols of methods that belong together logically. Just click on a (sub-)protocol name to see what methods it implements. (You may want to review the section on Protocols in the Objective-C book here!)

In the list below, any line that contains the word CREATABLE defines an object that you can use in your program. (This is part of the Swarm DefObj machinery.) In other words, while 'InternalState' is a normal protocol (a list of method definitions), the name 'ACGgen' refers to both a protocol and a class that implements that protocol. (You create an ACG generator by saying myGen = [ACGgen create: ....]). Similarly, 'GammaDist' defines both a protocol and a class that implements that protocol.

All generators and distributions ultimately inherit from SwarmObject.

(Prot.html#SingleLongSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

- CombinedShortSplitGenerator
(Prot.html#CombinedShortSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

- CombinedLongSplitGenerator
(Prot.html#CombinedLongSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

- ACGgen
(ACGgen.html)
<SingleLongGenerator, CREATABLE>

- C2LCGXgen
(C2LCGXgen.html)
<CombinedShortSplitGenerator, CREATABLE>

- C2MRG3gen
(C2MRG3gen.html)
<CombinedLongGenerator, CREATABLE>

- C2TAUSgen
(C2TAUS1-3gen.html)
<CombinedShortGenerator, CREATABLE>

  - C2TAUS1gen
  (C2TAUS1-3gen.html#C2TAUS1gen)
  : C2TAUSgen <C2TAUSgen, CREATABLE>

  - C2TAUS2gen
  (C2TAUS1-3gen.html#C2TAUS2gen)
  : C2TAUSgen <C2TAUSgen, CREATABLE>

  - C2TAUS3gen
  (C2TAUS1-3gen.html#C2TAUS3gen)
  : C2TAUSgen <C2TAUSgen, CREATABLE>

- C3MWCgen
(C3MWCgen.html)
<CombinedLongGenerator, CREATABLE>

- C4LCGXgen
(C4LCGXgen.html)
<CombinedShortSplitGenerator, CREATABLE>

- LCGgen
(LCG1-3gen.html)
<SingleShortGenerator>

  - LCG1gen
  (LCG1-3gen.html#LCG1gen)
  : LCGgen <LCGgen, CREATABLE>

(PMMLCG1-9gen.html#PMMLCG4gen)
: PMMLCGgen <PMMLCGgen, CREATABLE>

  - PMMLCG5gen
    (PMMLCG1-9gen.html#PMMLCG5gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG6gen
    (PMMLCG1-9gen.html#PMMLCG6gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG7gen
    (PMMLCG1-9gen.html#PMMLCG7gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG8gen
    (PMMLCG1-9gen.html#PMMLCG8gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG9gen
    (PMMLCG1-9gen.html#PMMLCG9gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>

- PSWBgen
  (PSWBgen.html)
  <SingleLongGenerator, CREATABLE>
- RWC2gen
  (RWC28gen.html)
  <SingleLongGenerator, CREATABLE>
- RWC8gen
  (RWC28gen.html#RWC8gen)
  <CombinedLongGenerator, CREATABLE>
- SCGgen
  (SCGgen.html)
  <SingleLongGenerator, CREATABLE>
- SWBgen
  (SWB1-3gen.html)
  <SingleLongGenerator>
  - SWB1gen
    (SWB1-3gen.html#SWB1gen)
    : SWBgen <SWBgen, CREATABLE>
  - SWB2gen
    (SWB1-3gen.html#SWB2gen)
    : SWBgen <SWBgen, CREATABLE>
  - SWB3gen
    (SWB1-3gen.html#SWB3gen)
    : SWBgen <SWBgen, CREATABLE>

- UniformIntegerDist (UniformIntegerDist.html) <IntegerDistribution, CREATABLE>
- UniformUnsignedDist (UniformUnsignedDist.html) <UnsignedDistribution, CREATABLE>
- UniformDoubleDist (UniformDoubleDist.html) <DoubleDistribution, CREATABLE>
- ExponentialDist (ExponentialDist.html) <DoubleDistribution, CREATABLE>
- GammaDist (GammaDist.html) <DoubleDistribution, CREATABLE>
- NormalDist (NormalDist.html) <Normal, CREATABLE>
- LogNormalDist (NormalDist.html#LogNormalDist) <Normal, CREATABLE>

(../random.html)

(../random.html)

(UsageGuide.html)

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

Last modified: 1998-01-10

(void) -getState: (void *) state writes a copy of the state of the object into the memory address pointed to by the "state" pointer passed to it. The memory should be allocated before the message is sent.

(void) -setState: (void *) state copies the values contained in the memory address pointed to by "state" into the state variables used by the object. The block of memory containing the state may be sent back to the heap after the message has been sent.

(./Interface.html)

(./Interface.html)

These descriptions are not finished. Please see the documentation in $(SWARMHOME)/src/random/

Generator Protocols

SingleSeed

```
@deftype SingleSeed
CREATING
+  create: aZone setStateFromSeed: (unsigned) seed;
USING
- setStateFromSeed:  (unsigned)   seed;
- (unsigned) getMaxSeedValue;
- (unsigned)  getInitialSeed;
@end
```

MultiSeed

```
@deftype MultiSeed
CREATING
+ create: aZone setStateFromSeeds: (unsigned *) seeds;
```

```
   setw: (unsigned) w // log2(segment length)
setStateFromSeeds: (unsigned *) seeds;
USING
- setStateFromSeeds: (unsigned *) seeds;
- (unsigned)  lengthOfSeedVector;
- (unsigned *) getMaxSeedValues; // min is 1
- (unsigned *) getInitialSeeds; // = getInitialSeeds: 0
@end
```

Split

```
@deftype Split
CREATING
+ createWithDefaults: aZone;
USING
- setAntithetic: (BOOL) antiT;
- (unsigned)    getNumGenerators; // returns A
- (unsigned)    getNumSegments; // returns v
- (unsigned)    getSegmentLength; // returns w
- (BOOL) getAntithetic;
- initGenerator:    (unsigned) vGen; // to segment #0
- restartGenerator: (unsigned) vGen; // start of current segment
- advanceGenerator: (unsigned) vGen; // to next segment
- jumpGenerator:    (unsigned) vGen  toSegment: (unsigned long long int) seg;
- initAll; // to segment #0
- restartAll; // start of current segment
- advanceAll; // to next segment
- jumpAllToSegment: (unsigned long long int) seg;
- (unsigned long long int) getCurrentCount:   (unsigned) vGen;
- (unsigned long long int) getCurrentSegment: (unsigned) vGen;
@end
```

SplitOut

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

SingleShortGenerator
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed>

No extra messages added.

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

```
- setGenerator: (id) generator;
- setGenerator: (id) generator
setVirtualGenerator: (unsigned) vGen;
- createEnd;
- reset;
-               getGenerator;
- (unsigned) getVirtualGenerator;
- (BOOL) getOptionsInitialized;
- (unsigned long long int) getCurrentCount;
@end
```

BooleanDistribution
<ProbabilityDistribution>

```
//
// BooleanDistribution --
//   A probability distribution that returns YES/NO sample values.
//
@deftype BooleanDistribution <ProbabilityDistribution>
- (BOOL) getBooleanSample;
- (int) getIntegerSample; // for convenience
@end
```

IntegerDistribution
<ProbabilityDistribution>

```
//
// IntegerDistribution --
//   A probability distribution that returns integer sample values.
//
@deftype IntegerDistribution <ProbabilityDistribution>
- (int) getIntegerSample;
@end
```

UnsignedDistribution
<ProbabilityDistribution>

```
//
// UnsignedDistribution --
//   A probability distribution that returns non-
negative integer sample values.
//
@deftype UnsignedDistribution <ProbabilityDistribution>
- (unsigned) getUnsignedSample;
@end
```

DoubleDistribution
<ProbabilityDistribution>

```
//
// DoubleDistribution --
//   A probability distribution that returns an approximation of continuous
```

ACGgen Class

```
//
// ACGgen--
//   predefined random number generators, not recommended for serious use;
//   included for historical reasons (compatibility with earlier releases).
//
@protocol ACGgen      <SingleLongGenerator, CREATABLE> @end
```

Use of this generator is NOT Recommended

Documentation

```
Generator name:  | ACG

Description:  | Additive Congruential Generator

Reference:  | ACG is an often used algorithm, mentioned in Knuth,
| Numerical Recipes, etc.

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

| ACG is in the Lagged Fibonacci class of generators.
| 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.

Comments:  | 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.)
```

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

(./Interface.html)

(./Interface.html)

(./C2LCGXgen.html)

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

Last modified: 1998-01-26

```
Comments:  | For a discussion of PMMLCG generators, see the source file
| PMMLCGgen.h.

Configuration | unsigned int numGenerators; // # virtual generators
parameters:  | unsigned int numSegments; // log2(#segments/vGen)
| unsigned int segmentLength; // log2(segment length)

State:  | unsigned int state [numGenerators] [2];
| (each virtual generator has a 4-integer state vector)

Output types | unsigned int in [0,unsignedMax].
(for each | float in [0.0,1.0) using 1 iteration of the generator.
virtual | double in [0.0,1.0) using 1 or 2 iterations of the generator.
generator):  | long double in [0.0,1.0) using 2 iterations of the generator.

Output range:  | The combined generator yields output in the range [0,m0-1]
| (i.e. [0,2147483562].)

Limits on | Number of generators A <= 2^25  (33,554,432)
parameters:  | (You'll likely run out of memory before then ...)
| Number of segments / generator <= 2^60 (i.e. v <= 60)
| Length of each segment <= 2^60 (i.e. w <= 60)

Defaults:  | A=32, v=20, w=30

Valid seeds:  | Using create:setStateFromSeed: in [1, 2^32-1]
| Using create:setStateFromSeeds:
| The generator must be seeded with 2 starting seeds:
|  Seed[0]: [1, 2147483562] = [1,m1-1]
|  Seed[1]: [1, 2147483398] = [1,m2-1]

Cycles:  | With properly chosen parameters, the component generators
| have a single full cycle of length (m-2).
| The combined generator has a cycle length of
| (m1-1)*(m2-1)/2 = 2147483562*2147483398/2 ~= 2^61.

Output Quality:  | All bits should be safe.
| The combined generator has much better statistical
| properties than either of its component PMMLCG's.

Reference for | L'Ecuyer and Cote, 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
```

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

Last modified: 1998-01-26

```
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,m1-2].

Valid seeds:  | This generator requires 3 seeds for each component:
| within [1,m1-1] for component 1;
| within [1,m2-1] for component 2.
| Using create:setStateFromSeed: in [1,2^32-1]

Cycles:  | With properly chosen parameters, this generator
| has a single cycle of length (m1^3-1)(m2^3-1)/2
| or 2^184 < cycle < 2^185 (4.9e55).

Output Quality:  | Good.

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

C2MRG3gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       13.459 uS
|   -getFloatSample, getDoubleSample:         15.203 uS
|   -getFatDoubleSample, getLongDoubleSample: 29.699 uS

Relative speed:
| Speed 0.275 (time 3.640) relative to MT19937 getUnsignedSample
--------------- . --------------------------------------------------------------
--

Parameters:  | unsigned int k = 3 // lag order

| unsigned int m1 = 2^31 - 1 // modulus
| unsigned int a12 = 63308 // multiplier
| unsigned int a13 = -183326 // multiplier

| unsigned int m2 = 2145483479 // modulus
| unsigned int a21 = 86098 // multiplier
| unsigned int a23 = -539608 // 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)^3 - 1) * (2145483479^3 - 1) / 2 ~= 4.89e55,
| i.e. 2^184 < cycle < 2^185.
| (The paper, in apparent error, claims
| that the cycle is close to 2^205.)
```

C2TAUSgen Class

```
//
// C2TAUS1, C2TAUS2, C2TAUS3 --
//   short component based generator recommended for use.
//
@deftype C2TAUSgen    <CombinedShortGenerator> @end
@deftype C2TAUS1gen   <C2TAUSgen, CREATABLE> @end
@deftype C2TAUS2gen   <C2TAUSgen, CREATABLE> @end
@deftype C2TAUS3gen   <C2TAUSgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | C2TAUS

Description:  | Combined Tausworthe Generator
| based on 2 component generators of periods 2^31-1 and 2^29-1

Reference:  | Shu Tezuka and Pierre L'Ecuyer,
| "Efficient and Portable Combined Tausworthe
| Random Number Generators."
| ACM Transactions on Modeling and Computer Simulation,
| vol. 1 no. 2, April 1991, pp. 99-112.

| Original reference for the Tausworthe algorithm:
| R. C. Tausworthe, "Random Numbers Generated by Linear
| Recurrence modulo 2", Math. Comput. 19 (1965), 201-209.

Algorithm:  | state = seed, then:

(Each | b = ((old << Q) ^ old) & mask
component:)  | new = ((old << S) ^ (b >> (P-S)) & mask

(Combination:)  | output = ((new1 ^ (new2 << (P1-P2))

Implementation:  |

Parameters:  | int   P [2] // word length
| int   Q [2] //
| int   S [2] //
| int PmS [2] // P - S
| int P1mP2 // P[0] - P[1]
```

```
| Q =  2

Reference for | These values recommended by Tezuka and L'Ecuyer,
parameters:  | op. cit., table I 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:  |
```

C2TAUS2gen Documentation

```
Parameters:  | P = 31 so Mask = 2^31-1
(component 1) | S = 21 so PmS = 10
| Q =  3

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

Reference for | These values recommended by Tezuka and L'Ecuyer,
parameters:  | op. cit., table II 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:  |
```

C2TAUS3gen Documentation

```
Parameters:  | P = 31 so Mask = 2^31-1
(component 1) | S = 13 so PmS = 18
| Q = 13

| P = 29 so Mask = 2^29-1
(component 2) | S = 20 so PmS =  9
| Q =  2

Reference for | These values recommended by Tezuka and L'Ecuyer,
```

C3MWCgen Class

```
//
// C3MWC --
//   long component based generator recommended for use.
//
@deftype C3MWCgen   <CombinedLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C3MWC

Description:  | Combined 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);
| I = c * (I & 65535) + (I>>16);
| L = d * (L & 65535) + (L>>16);
| M = e * (M & 65535) + (M>>16);
| N = f * (N & 65535) + (N>>16);
| return ((K+I+M)>>16) + (J+L+N);

Comments:  | This generator is a combination of 3 MWC generators,
| each of which is a combination of 2 16-bit
| Multiply-With-Carry generators.

Parameters:  | unsigned int a = 30903 // multiplier
| unsigned int b = 18000
| unsigned int c = 29013
| unsigned int d = 30345
| unsigned int e = 30903
| unsigned int f = 31083

State:  | unsigned int K,J,I,L,M,N // state vector

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

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

Last modified: 1998-01-26

```
Comments:  | For a discussion of PMMLCG generators, see the source file
| PMMLCGgen.h.

Configuration | unsigned int numGenerators; // # virtual generators
parameters:  | unsigned int numSegments; // log2(#segments/vGen)
| unsigned int segmentLength; // log2(segment length)

State:  | unsigned int state [numGenerators] [4];
| (each virtual generator has a 4-integer state vector)

Output types | unsigned int in [0,unsignedMax].
(for each | float in [0.0,1.0) using 1 iteration of the generator.
virtual | double in [0.0,1.0) using 1 or 2 iterations of the generator.
generator):  | long double in [0.0,1.0) using 2 iterations of the generator.

Output range:  | The combined generator yields output in the range [0,m0-2]
| (i.e. [0,2147483645].)

Limits on | Number of generators A <= 2^25  (33,554,432)
parameters:  | (You'll likely run out of memory before then ...)
| Number of segments / generator <= 2^63 (i.e. v<=63)
| Length of each segment <= 2^63 (i.e. w<=63)

Defaults:  | A=128, v=31, w=41

Valid seeds:  | Using create:setStateFromSeed: in [1, 2^32-1]
| Using create:setStateFromSeeds:
| The generator must be seeded with 4 starting seeds:
|  Seed[0]: [1, 2147483646] = [1,m1-1]
|  Seed[1]: [1, 2147483542] = [1,m2-1]
|  Seed[2]: [1, 2147483422] = [1,m3-1]
|  Seed[3]: [1, 2147483322] = [1,m4-1]

Cycles:  | With properly chosen parameters, the component generators
| have a single full cycle of length (m-1).
| The combined generator has a cycle length of
| 2147483646*2147483542*2147483422*2147483322/8 > 2^120.

Output Quality:  | All bits should be safe.
| The combined generator has much better statistical
| properties than either of its component PMMLCG's.

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

Implementation  | The code for -getUnsignedMax uses a trick devised by
```

(./LCG1-3gen.html)

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

Last modified: 1998-01-26

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

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: [1,2^32-1]

Cycles:  | With properly chosen parameters, these generators
| have a single full cycle of length (2^32).

Output Quality:  | Low order bits are poor.

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

LCGgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       2.564 uS
|   -getFloatSample, getDoubleSample:         4.295 uS
|   -getFatDoubleSample, getLongDoubleSample: 7.734 uS

Relative speed:
| Speed 1.442 (time 0.693) relative to MT19937 getUnsignedSample
```

LCG1gen Documentation

```
Parameters:  | a =     1,664,525
| c = 1,013,904,223

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:  |
```

(./MRG5-7gen.html)

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

Last modified: 1998-01-26

```
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,unsignedMax] = [0,m-2].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds: [1, unsignedMax]
| (unsignedMax differs between MRG5, MRG6, MRG7)

Cycles:  | With properly chosen parameters, these generators
| have a single cycle of length m^k - 1.
| Thus, for m ~= 2^31 we have:
| k=5 => cycle ~= 2^155,
| k=6 => cycle ~= 2^186,
| k=7 => cycle ~= 2^217.

Output Quality:  | Good.

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

MRG5gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        9.674 uS
|   -getFloatSample, getDoubleSample:         11.524 uS
|   -getFatDoubleSample, getLongDoubleSample: 22.135 uS

Relative speed:
| Speed 0.382 (time 2.616) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--

MRG6gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       10.449 uS
|   -getFloatSample, getDoubleSample:         12.337 uS
|   -getFatDoubleSample, getLongDoubleSample: 27.735 uS

Relative speed:
| Speed 0.354 (time 2.826) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--

MRG7gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       10.913 uS
```

MRG7gen Documentation

```
Parameters:  | unsigned int k = 7 // lag order
| unsigned int m = 2^31 - 19 // modulus
| unsigned int a1 = 1071064 // multiplier
| unsigned int a7 = 2113664 // 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)^7 - 1,
| i.e. 2^216 < cycle < 2^217.

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

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

(./LCG1-3gen.html)

(./Interface.html)

(./MT19937gen.html)

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

Last modified: 1998-01-26

```
Output range:  | Any number in the range [0,2^32] may be returned

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

Cycles:  | This generator has a single cycle of length 2^19937-1.

Output Quality:  | Excellent equidistribution properties in all dimensions
| up to 623 are claimed.

Reference for | Matsumoto and Nishimura, op. cit.
output quality:  | See also pLab Report #07 on the pLab web site.

MT19937gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        3.698 uS
|   -getFloatSample, getDoubleSample:          5.502 uS
|   -getFatDoubleSample, getLongDoubleSample: 10.159 uS

Relative speed:
| Speed 1.0 (time 1.0) relative to itself (it's the benchmark)

--------------- . -----------------------------------------------------------
--

Parameters:  | unsigned int w=32 // word length
| unsigned int N=624 // size of state array
| unsigned int M=397 // lag
| unsigned int s=7 // 'magic vector'
| unsigned int t=15 //    "     "
| unsigned int u=11 //    "     "
| unsigned int l=18 //    "     "
| unsigned int b=0x102d1200 //    "     "
| unsigned int c=0x66e50000 //    "     "
| unsigned int a=0x6b5eccf6 //    "     "

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

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^19937 -1,
| or 10^6001 !!! (The longest period yet.)

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

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

MWCA-Bgen Classes

```
//
// MWC --
//   two long generators recommended for use.
//
@deftype MWCAgen    <SingleLongGenerator, CREATABLE> @end
@deftype MWCBgen    <SingleLongGenerator, CREATABLE> @end
```

MWCAgen Documentation

```
Generator name:  | MWCA

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;

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

Implementation:  |

Parameters:  | unsigned int a = 30903 // multiplier
| unsigned int b = 18000 // multiplier

State:  | unsigned int K,J // 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,unsignedMax] = [0,2^32-1].
```

```
| It also passes all tests and is fast.

Parameters:  | unsigned int a = 30903 // multiplier
| unsigned int b = 18000 // multiplier

State:  | unsigned int K,J // 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,unsignedMax] = [0,2^32-1].

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
```

(./MT19937gen.html)

PMMLCG 1-9 Classes

```
//
// PMMLCG1, PMMLCG2, PMMLCG3 --
//   single short random number generators recommended for use.
//
// PMMLCG4, PMMLCG5, PMMLCG6, PMMLCG7 --
//   component generators of C4LCGX
//
// PMMLCG8, PMMLCG9 --
//   component generators of C2LCGX
//
@deftype PMMLCGgen   <SingleShortGenerator> @end
@deftype PMMLCG1gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG2gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG3gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG4gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG5gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG6gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG7gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG8gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG9gen  <PMMLCGgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | PMMLCG

Description:  | Prime Modulus Multiplicative Linear Congruential Generator

Reference:  | Stephen K. Park and Keith W. Miller
| (CACM 31:10, October 1988, 1192-1201),
| "Random Number Generators: Good Ones Are Hard To Find."
| The authors refer to this generator as a "minimally
| acceptable" or "minimally sufficient" generator.

| state = seed
Algorithm:  | newstate = (a * oldstate) mod m
| output = state-1

Comments:  | The modulus m must be a prime for this class of generator
| (as opposed to the more common power of two modulus).
| For these generators we use m=2^31-1 (2,147,483,647).
```

```
--------------- | -----------------------------------------------------------
---
```

PMMLCG1gen Documentation

```
Parameters:  | a =        16,807
| m = 2,147,483,647 (2^31-1)

Reference for | Park and Miller, op. cit.
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 have been extensively tested by the authors.

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

PMMLCG2gen Documentation

```
Parameters:  | a =        48,271
| m = 2,147,483,647 (2^31-1)

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

PMMLCG3gen Documentation

```
Parameters:  | a =        69,621
| m = 2,147,483,647 (2^31-1)

Reference for | Park and Miller, op. cit.
```

```
Output Quality:  | All bits should be safe.

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

PMMLCG6gen Documentation

```
Parameters:  | a =       138,556
| m = 2,147,483,423 (2^31-225)

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:  |
```

PMMLCG7gen Documentation

```
Parameters:  | a =        49,689
| m = 2,147,483,323 (2^31-325)

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:  |
```

(./Interface.html)

(./PSWBgen.html)

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

Last modified: 1998-01-26

```
| unsigned int m // modulus

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

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,m-1] may be returned.

Valid seeds:  | Using create:setStateFromSeed:  [1,2^32-1]
| Using create:setStateFromSeeds: r seeds in [1,2^32-7]
| and one seed in [0,1] (the carry)

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

| Unfortunately, there are no 'jump-ahead' or 'splitting'
| facilities that would allow us to jump ahead in the cycle,
| other than calling the generator a number of times.

Output Quality:
| Much better than other Lagged Fibonacci generators, such as
| ACG and SCG.

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

PSWBgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        3.452 uS
|   -getFloatSample, getDoubleSample:         5.166 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.544 uS

Relative speed:
| Speed 1.070 (time 0.933) relative to MT19937 getUnsignedSample
--------------- | -----------------------------------------------------------
---

Parameters:  | r = 43 // long lag
| s = 22 // short lag
| m = 2^32 - 5 // modulus (prime)

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

RWC2,8gen Classes

```
//
// RWC2 --
// RWC8 ("Mother") --
//   single long generator recommended for use.
//
@deftype RWC2gen   <SingleLongGenerator, CREATABLE> @end
@deftype RWC8gen   <CombinedLongGenerator, CREATABLE> @end
```

RWC2gen Documentation

```
Generator name:  | RWC2

Description:  | 2-lag Recursion-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/rwc1.ps on his Diehard CD-ROM,
| found at <http://www.hku.hk/internet/randomCD.html>.

Algorithm:  | x[n]  = a*(x[n-1]+x[n-2]) + carry % 2^32
| carry = a*(x[n-1]+x[n-2]) + carry / 2^32

Comments:  | This generator is a 2-lag MWC generator

Implementation:  | This generator is implemented using 64-bit math

Parameters:  | unsigned int a = 1,111,111,464 // multiplier

State:  | unsigned int carry,x1,x2 // 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,unsignedMax] = [0,2^32-1].

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

```
State:  | unsigned short m1[10], m2[10] // 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,unsignedMax] = [0,2^32-1].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds:
|   a total of 18 16-bit seeds are needed;
|   each xi may be in [1, 2^31-1].
|   each carry may be in [1, sum(multipliers)-1].

Cycles:  | This generator is claimed to have a period > 2^250.
| It may have rho sequences up to length r = 8.
| It may have multiple cycles of the stated period length.

Output Quality:  | Claimed to be good.

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

RWC8gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       14.649 uS
|   -getFloatSample, getDoubleSample:         16.400 uS
|   -getFatDoubleSample, getLongDoubleSample: 31.915 uS

Relative speed:
| Speed 0.252 (time 3.961) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

(./PSWBgen.html)

(./Interface.html)

SCGgen Class

```
//
// SCG --
//   single long random number generator,
//   NOT recommended for serious use; included for
//   historical reasons (compatibility with earlier releases).
//
@deftype SCGgen     <SingleLongGenerator, CREATABLE> @end
```

Use of this generator is NOT Recommended

Documentation

```
Generator name:  | SCG

Description:  | Subtractive Congruential Generator

Reference:  | SCG is an often used algorithm, mentioned in Knuth,
| Numerical Recipes, etc.

| state vector = f(seed)
Algorithm:  | state_n = (state_(n-r) - state_(n-s)) mod 10^9; r > s
| output = state_n

| SCG is in the Lagged Fibonacci class of generators.
| 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
```

Output Quality:	Low-order bits are poor.
Reference for output quality:	Knuth, op.cit.

(./RWC28gen.html)

(./Interface.html)

(./SWB1-3gen.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
| int carry // carry bit

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: r seeds in [1,2^32-2]
| and one seed in [0,1] (the carry)

Cycles:  | With properly chosen parameters, these generators
| have a number of long cycles, up to 10^354 in length.
| UNFORTUNATELY there is no way to pick different
| cycles, nor any easy way to 'jump ahead' in the
| chosen cycle (other than calling -getUnsignedSample
| a number of times).

Output Quality:
| Much better than other Lagged Fibonacci generators, such as
| ACG and SCG.

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

SWBgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       3.285 uS
|   -getFloatSample, getDoubleSample:         5.057 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.209 uS

Relative speed:
| Speed 1.126 (time 0.888) relative to MT19937 getUnsignedSample
--------------- | -----------------------------------------------------------
---
```

SWB1gen Documentation

```
Parameters:  | r = 37
```

(./SCGgen.html)

(./Interface.html)

(./TGFSRgen.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 [0,2^w-1] may be returned

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

Cycles:  | With properly chosen parameters, these generators
| have a single cycle of length 2^(w*N) -1.

Output Quality:  | Good.

Reference for | Matsumoto and Kurita, op. cit.
output quality:  | Also recommended by P. L'Ecuyer (private communication).

TT403gen, | On a 486/66, the following approximate measures were obtained:
TT775gen, |   -getUnsignedSample:                        4.654 uS
TT800gen |   -getFloatSample, getDoubleSample:          6.430 uS
speed:  |   -getFatDoubleSample, getLongDoubleSample: 11.953 uS

Relative speed:
| Speed 0.795 (time 1.259) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

TT403gen Documentation

```
Parameters:  | unsigned int w=31 // word length
| unsigned int N=13 // size of state array
| unsigned int M=2 // lag
| unsigned int s=8 // 'magic vector'
| unsigned int t=14 //    "     "
| unsigned int b=0x102d1200 //    "     "
| unsigned int c=0x66e50000 //    "     "
| unsigned int a=0x6b5eccf6 //    "     "

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

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^403 -1.

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

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

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

(./SWB1-3gen.html)

(./Interface.html)

(./Prot.html#ProbabilityDist)

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

Last modified: 1998-01-26

(Interface.html)

(BernoulliDist.html)

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

Last modified: 1998-01-10

--------------- . ---
--

(./RandomBitDist.html)

(./Interface.html)

(./UniformIntegerDist.html)

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

Last modified: 1998-01-26

```
|  -getIntegerSample:                         7.534 uS
|  -average # of generator calls per variate: 1.000 (U)
|  -distribution net time:                    3.836 uS

Relative speed:
| Speed 0.491 (time 2.037) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

(./BernoulliDist.html)

(./Interface.html)

(./UniformUnsignedDist.html)

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

Last modified: 1998-01-26

```
comment:        | Then minValue is returned "with probability 1".

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

|  -getUnsignedSample:                        7.632 uS
|  -average # of generator calls per variate: 1.000 (U)
|  -distribution net time:                    3.934 uS

Relative speed:
| Speed 0.485 (time 2.064) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

(./UniformIntegerDist.html)

(./Interface.html)

(./UniformDoubleDist.html)

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

Last modified: 1998-01-26

```
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          13.777 uS
|  -average # of generator calls per variate:  1.000 (D)
|  -distribution net time:                     3.618 uS

Relative speed:
| Speed 0.268 (time 3.726) relative to MT19937 getUnsignedSample
--------------- . -------------------------------------------------------------
--
```

(./UniformUnsignedDist.html)

(./Interface.html)

(./ExponentialDist.html)

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

Last modified: 1998-01-26

--------------- . ---
--

(./UniformDoubleDist.html)

(./Interface.html)

(./GammaDist.html)

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

Last modified: 1998-01-26

```
Relative speed:
| 0.00310 (time 32.294) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

(./ExponentialDist.html)

(./Interface.html)

(./NormalDist.html)

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

Last modified: 1998-01-10

```
Name:  | LogNormalDistribution

Description:  | Log-Normal distribution returning double values

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

Algorithm:  | The Box-Muller method

Parameters:  | double Mean
| double Variance

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

Output types:  | double

Output Quality:  | All bits should be safe.

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

|  -getDoubleSample:                          33.430 uS
|  -average # of generator calls per variate:  1.273 (D)
|  -distribution net time:                    20.498 uS

Relative speed:
| Speed 0.111 (time 9.040) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

(GammaDist.html)

(Interface.html)

Random Library: Usage Guide

Table Of Contents:

Overview

The random library basically consists of two parts, the actual pseudorandom number algorithms and the transformations that change the output of the pseudorandom number algorithms into the simulated distributions. The pseudorandom number algorithms are called generators throughout these documents and many times the "pseudo" is dropped from pseudorandom, even though there is no true random number generator implemented in Swarm.

Generator Usage Notes

Primary Author: Sven Thommesen
(mailto:sthomme@humsci.auburn.edu)
Swarm rel. 1.0.3 (random rel. 0.8)
Notes on the use of the random number generators
Sven Thommesen <sthomme@humsci.auburn.edu>
1997-09-01

```
IMPROVEMENTS over v. 0.6:

A host of new generators, located on the web or in the literature, have
been added since the last version of Random. There is now a total of 36
different generators defined! Some of these have immense periods, some
are very fast, and some have much better statistical properties than
the old generators.

A new *type* of generator, the 'split' generator, has been introduced
in the form of L'Ecuyer's C2LCGXgen and C4LCGXgen generators.

A 'split' generator is a long-period generator for which we are able to
split the period into arbitrary sub-periods, which we can access quickly.
```

A bug in SWBgen was corrected. Code for ACG and SCG was also changed.

The -verifySelf method is gone. Instead see the test program located in /random/testR0.

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.

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

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

USAGE notes:

I shall first discuss 'simple' generators, and then discuss how 'split' generators differ from 'simple' ones.

In the following text, wherever I use PSWBgen as an example you may substitute any other generator.

Note: any name that starts with 'my' is meant to designate a variable which you have defined in your own program.

SIMPLE generators:

You create a generator in one of 3 ways:

(a) the lazy way:

```
   myGenerator = [ PSWBgen createWithDefaults: [self getZone] ];
```

This allocates the object and initializes it with STARTSEED.

(b) using a single seed value:

```
   myGenerator = [ PSWBgen create: [self getZone]
setStateFromSeed: mySeed ];
```

```
myVector = [ myGenerator getInitialSeeds ];
```

And you can find out the largest seed values that are allowed for the particular generator:

```
myVector = [ myGenerator getMaxSeedValues ];
```

(These values vary from generator to generator, and they may not be the same for all elements of the vector for a given generator. Seeds may not be 0.)

NOTE: in the above two calls, the variable myVector is set to point to an array internal to the generator. If you want to preserve the values, you need to allocate space in your program either statically or dynamically, and use a for-loop to copy data from myVector[i] to myAllocatedVector[i].

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

```
[ myGenerator setStateFromSeeds: (unsigned *) mySeedVector ];
```

This will also reset to 0 the currentCount variable.

NOTE: if you set a generator's state from a vector of seeds, the call

```
myUnsignedValue = [ myGenerator getInitialSeed ];
```

will return a value of 0 (an invalid seed). On the other hand, if you initialize the generator with a single seed value, the call

```
mySeedVector = [ myGenerator getInitialSeeds ];
```

will return the seed vector that would produce identical output to what you obtained using the single seed.

(d) antithetic values:

You can make the generator serve up antithetic values by setting:

```
[ myGenerator setAntithetic: myBooleanValue ];
```

If set, this makes -getUnsignedSample return (unsignedMax-x) instead of x, and the floating point methods return (1.0 - y) instead of y.

You can ascertain if this flag is set by calling

```
myBooleanValue = [ myGenerator getAntithetic ];
```

```
   myGenerator = [ C4LCGXgen create: [self getZone]
setA: 32 setv: 25 setw: 60
setStateFromSeeds: (unsigned *) mySeedVector ];
```

(In both cases, the only limitation is that A * 2^v * 2^w must be less than the generator's period, 2^60 for C2LCGX and 2^120 for C4LCGX.)

For obtaining output, we need to specify which of the A 'virtual' generators we want to draw from:

```
   myUnsignedValue = [ myGenerator getUnsignedSample: 12 ];
   myFloatValue    = [ myGenerator getFloatSample: myVirtualGenerator ];
   myDoubleValue   = [ myGenerator getThinDoubleSample: someUnsignedValue ];
   myDoubleValue   = [ myGenerator getDoubleSample: 32 ];
   myLongDoubleValue = [ myGenerator getLongDoubleSample: 0 ];
```

Note: virtual generators are numbered from 0 to A-1.

Obtaining the current count of variates generated likewise:

```
   myLongLongInt = [ myGenerator getCurrentCount: myVirtualGenerator ];
   myLongLongInt = [ myGenerator getCurrentSegment: myVirtualGenerator ];
```

The latter call indicates what segment number the virtual generator is currently drawing numbers from.

Other than these methods, the other methods discussed above under 'simple' generators are the same for 'split' generators.

In *addition* to this, 'split' generators have the following methods to manage the virtual generators:

```
   // Place all virtual generators at the start of the first segment:
   [ myGenerator initAll ]; // done automatically at creation

   // Place all virtual generators back to the start of the current segment:
   [ myGenerator restartAll ];

   // Place all virtual generators at the start of the next segment:
   [ myGenerator advanceAll ];

   // Place all virtual generators at the start of the indicated segment:
   [ myGenerator jumpAllToSegment: myLongLongIntValue ];
```

You may also address individual virtual generators:

```
   [ myGenerator initGenerator:    myVgen ];
```

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:
stateSizeG = [ myGenerator getStateSize ];

// Allocate memory for the buffer:
stateBufG  = [[self getZone] alloc: stateSizeG];

// Ask the generator to put state data into the buffer:
[ myGenerator putStateInto: (void *) stateBufG ];

// Open a disk file for output:
myFile = fopen(myFileName, "w");
if (myFile == NULL) { }; // error on open: disk full, or no permissions

// Write the state buffer to disk in binary form:
status = fwrite(stateBufG, stateSizeG, 1, myFile);
if (status < 1) { }; // error on write: disk full?

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

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

// Or, for test purposes, just zero the buffer data instead:
// memset(stateBufG, 0, stateSizeG);
```

This code shows how to set an object’s state from a disk file:

```
// Ask how big a buffer we need:
stateSizeG = [ myGenerator getStateSize ];

// Allocate memory for the buffer:
stateBufG  = [[self getZone] alloc: stateSizeG];

// Open a disk file for input:
myFile = fopen(myFileName, "r");
if (myFile == NULL) { }; // error on open: file not found

// Read state data into the memory buffer:
status = fread(stateBufG, stateSizeG, 1, myFile);
if (status < 1) { }; // error on read
```

```
For what it's worth, Professor L'Ecuyer recommends his own C4LCGX and C2MRG3
generators as well as Matsumoto's TT800 (the monster MT19937 hadn't been
released yet), and Prof. Marsaglia recommends his own Multiply-With-Carry
generators (MWCA, MWCB, C3MWC, RWC2, RWC8="Mother").


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.
```

Distribution Usage Notes

Primary Author: Sven Thommesen
(mailto:sthomme@humsci.auburn.edu)
Swarm rel. 1.0 (random rel. 0.6)
Notes on the use of the distribution objects
Sven Thommesen <sthomme@humsci.auburn.edu>
1997-09-01

```
IMPROVEMENTS over v. 0.6:

One new distribution class, BernoulliDist, has been added. It returns
binary values (yes/true/1) with a given probability (while the old
RandomBitDist has a fixed 50% probability, a fair coin toss.)

Distributions now have a new create method, '+createWithDefaults: aZone'.
This method creates the distribution object, and also a new generator
object for its exclusive use. Each distribution class has a different
default generator class assigned. These generators are initialized with
STARTSEED, which by default equals the fixed value DEFAULTSEED, but will
be equal to the varying RANDOMSEED if you start your program with the
command line parameter '-varySeed'.

All distributions have code to interact with the new 'split' generators.
```

Each distribution object must be assigned a random generator on creation. You may not re-assign generators to a distribution after it has been created. You may, however, connect more than one distribution object to one generator, so that they end up drawing output from the generator in an interleaved fashion. Or you may create a new generator for each distribution object.

Each distribution has its own set of (1-2) parameters. You may deal with these parameters in two different ways: (1) you assign a set of default parameter values to the object on creation, and draw from the distribution using those parameters. Or (2), you may refrain from assigning default parameters, in which case you must specify the (possibly different) desired parameters on each call. You can (re-)set the default parameters any time, and you may call for a variate with specified parameter values even if different default parameters have been set.

Just like the generator objects, the distribution objects allow you to save and later restore their internal state.

Where I use NormalDistribution in examples below, substitute any other distribution and its parameters as needed.

NOTE: any name that starts with 'my' is meant to designate a variable which you have defined in your own program.

You create a distribution in one of several ways:

(a) the lazy way:

```
   myNormalDist = [ NormalDist createWithDefaults: [self getZone]];
```

This method will create a distribution object with no default parameters set, as well as a fresh generator object connected to it. The generator object is initialized with STARTSEED (see the discussion above). Different distribution classes use different generators for this purpose.

(b) Without default parameters, using a simple generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySimpleGenerator ];
```

'myGenerator' must of course first have been set to point to a random generator of the 'simple' type. Note that you cannot assign a different generator to a distribution after it has been created.

```
myDouble = [ myNormalDist getDoubleSample ];
```

However, you *are* allowed to specify parameters even if default parameters have been set.

(Of course, different distributions have different parameters: RandomBitDist has none, the Uniform objects have minimum and maximum limit val-
ues, NormalDist
and LogNormalDist use Mean and Variance, ExponentialDist only Mean, and GammaDist used alpha and beta. See the file random/distributions.h for the specific methods available.)

(f) You may reset the default parameters this way, as often as you like:

```
[ myNormalDist setMean: 3.3 setVariance: 2.2 ];
```

(g) You can obtain the current values of parameters:

```
// Default parameters:
myDouble1 = [ myNormalDist getMean ];
myDouble2 = [ myNormalDist getVariance ];
myDouble3 = [ myNormalDist getStdDev ];

// Get a pointer to the generator object:
myOtherGenerator = [ myNormalDist getGenerator ];

// Get the number of the virtual generator (if a split generator is used):
myUnsignedValue  = [ myNormalDist getVirtualGenerator];

// Find out if default parameters have been set:
myBoolean        = [ myNormalDist getOptionsInitialized ];

// Find out how many variates the object has delivered so far:
// (The counter is an unsigned long long int, which goes up to 2^64.)
myLongLongInt    = [ myNormalDist getCurrentCount ];
```

(h) You can reset the variate counter and other state variables this way:

```
[ myNormalDist reset ];
```

This is most likely done in conjunction with resetting the connected generator, using [myGenerator setStateFromSeed: mySeedValue];

(i) Finally, we have the InternalState protocol methods:

```
[ myNormalDist putStateInto: (void *) stateBufD ];

// Open a disk file for output:
myFile = fopen(myFileName, "w");
if (myFile == NULL) { }; // error on open: disk full, or no permissions

// Write the state buffer to disk in binary form:
status = fwrite(stateBufD, stateSizeD, 1, myFile);
if (status < 1) { }; // error on write: disk full?

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

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

// Or, for test purposes, just zero the buffer data instead:
// memset(stateBufD, 0, stateSizeD);
```

This code shows how to set an object's state from a disk file:

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

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

// Open a disk file for input:
myFile = fopen(myFileName, "r");
if (myFile == NULL) { }; // error on open: file not found

// Read state data into the memory buffer:
status = fread(stateBufD, stateSizeD, 1, myFile);
if (status < 1) { }; // error on read

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

// Ask the distribution object to set its state from the buffer data:
[ myNormalDist setStateFrom: (void *) stateBufD ];

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

Random Library: Advanced Usage Guide

There should also be a general expository on the use of pseudo random number generation in simulation, especially as applied to Swarm.

(./UsageGuide.html)

(../random.html)

(./InterfaceDesign.html)

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

Last modified: 1998-01-10

(AdvancedUsage.html)

(../random.html)

(Implementation.html)

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

Last modified: 1998-01-10