Random Number Generation:
A Practitioner’s Overview

Michael Mascagni

Department of Computer Science & School of Computational Science
Florida State University, Tallahassee, FL 32306 USA
E-mail: mascagni@cs.fsu.edu or mascagni@math.ethz.ch
URL: http://www.cs.fsu.edu/~mascagni
Research supported by ARO, DOE/ASCI, NATO, and NSF

Outline of the Talk

1. Types of random numbers and Monte Carlo Methods
2. Pseudorandom number generation
   - Types of pseudorandom numbers
   - Properties of these pseudorandom numbers
   - Parallelization of pseudorandom number generators
3. Quasirandom number generation
   - The Koksma-Hlawka inequality
   - Discrepancy
   - The van der Corput sequence
   - Methods of quasirandom number generation
### What are Random Numbers Used For?

- Random numbers are used extensively in simulation, statistics, and in *Monte Carlo* computations
  - Simulation: use random numbers to “randomly pick” event outcomes based on statistical or experiential data
  - Statistics: use random numbers to generate data with a particular distribution to calculate statistical properties (when analytic techniques fail)
- There are many Monte Carlo applications of great interest
  - Numerical quadrature “all Monte Carlo is integration”
  - Quantum mechanics: Solving Schrödinger’s equation with Green’s function Monte Carlo via random walks
  - Mathematics: Using the Feynman-Kac/path integral methods to solve partial differential equations with random walks
  - Defense: neutrons, nuclear weapons design
  - Finance: options, mortgage-backed securities

### Why Monte Carlo?

- Rules of thumb for Monte Carlo methods
  - Good for computing linear functionals of solution (linear algebra, PDEs, integral equations)
  - No discretization error but sampling error is $O(N^{-1/2})$
  - High dimensionality is favorable, breaks the “curse of dimensionality”
  - Appropriate where high accuracy is not necessary
  - Often algorithms are “naturally” parallel
- Exceptions
  - Complicated geometries often easy to deal with
  - Randomized geometries tractable
  - Some applications are insensitive to singularities in solution
  - Sometimes is the fastest high-accuracy algorithm (rare)

### Pseudorandom Numbers

- Pseudorandom numbers mimic the properties of ‘real’ random numbers
  - “Real” random numbers: uses a ‘physical source’ of randomness
  - Pseudorandom numbers: deterministic sequence that passes tests of randomness
  - Quasirandom numbers: well distributed (low discrepancy) points

### Types of random numbers and Monte Carlo Methods

<table>
<thead>
<tr>
<th>Pseudorandom number generation</th>
<th>Quasirandom number generation</th>
</tr>
</thead>
</table>

- Cryptographic numbers
- Uniformity
- Unpredictability

### What are Random Numbers Used For?

- There are many types of random numbers
  - “Real” random numbers: uses a ‘physical source’ of randomness
  - Pseudorandom numbers: deterministic sequence that passes tests of randomness
  - Quasirandom numbers: well distributed (low discrepancy) points

### Types of random numbers

- Pseudorandom numbers: deterministic sequence that passes tests of randomness
- Quasirandom numbers: well distributed (low discrepancy)

### Properties of these pseudorandom numbers

- Uniformity
- Unpredictability
- Independence

### Rules of thumb for Monte Carlo methods

- Good for computing linear functionals of solution (linear algebra, PDEs, integral equations)
- No discretization error but sampling error is $O(N^{-1/2})$
- High dimensionality is favorable, breaks the “curse of dimensionality”
- Appropriate where high accuracy is not necessary
- Often algorithms are “naturally” parallel

### Exceptions

- Complicated geometries often easy to deal with
- Randomized geometries tractable
- Some applications are insensitive to singularities in solution
- Sometimes is the fastest high-accuracy algorithm (rare)
## Pseudorandom Numbers

- Some properties of pseudorandom number generators, integers: \( \{x_n\} \) from modulo \( m \) recursion, and \( U[0,1], z_n = \frac{x_n}{m} \)

- **A.** Should be a purely periodic sequence (e.g.: DES and IDEA are not provably periodic)
- **B.** Period length: \( \text{Per}(x_n) \) should be large
- **C.** Cost per bit should be moderate (not cryptography)
- **D.** Should be based on theoretically solid and empirically tested recursions
- **E.** Should be a totally reproducible sequence

### Linear congruential: \( x_n = a x_{n-1} + c \) (mod \( m \)), \( \text{Per}(x_n) = m - 1, m \text{ prime, with } m \text{ a power-of-two,} \)

\( \text{Per}(x_n) = 2^k \), or \( \text{Per}(x_n) = 2^{k-2} \) if \( c = 0 \)

- Implicit inversive congruential: \( x_n = a x_{n-1} + c \) (mod \( p \)), \( \text{Per}(x_n) = p \)

- Explicit inversive congruential: \( x_n = a n + c \) (mod \( p \)), \( \text{Per}(x_n) = p \)

- Shift register: \( y_n = y_{n-s} + y_{n-r} \) (mod 2), \( r > s \), \( \text{Per}(y_n) = 2^r - 1 \)

- Additive lagged-Fibonacci: \( z_n = z_{n-s} + z_{n-r} \) (mod 2^k), \( r > s \), \( \text{Per}(z_n) = (2^r - 1)2^{k-1} \)

- Combined: \( w_n = y_n + z_n \) (mod \( p \)), \( \text{Per}(w_n) = \text{lcm}(\text{Per}(y_n), \text{Per}(z_n)) \)

- Multiplicative lagged-Fibonacci: \( x_n = x_{n-s} \times x_{n-r} \) (mod 2^k), \( r > s \), \( \text{Per}(x_n) = (2^r - 1)2^{k-3} \)

## Combining RNGs

- There are many methods to combine two streams of random numbers, \( \{x_n\} \) and \( \{y_n\} \), where the \( x_n \) are integers modulo \( m_x \), and \( y_n \)'s modulo \( m_y \):

- Addition modulo one: \( z_n = \frac{x_n}{m_x} + \frac{y_n}{m_y} \) (mod 1)

- Addition modulo either \( m_x \) or \( m_y \)

- Multiplication and reduction modulo either \( m_x \) or \( m_y \)

- Exclusive “or-ing”

- Rigorously provable that linear combinations produce combined streams that are “no worse” than the worst

- Tony Warnock: all the above methods seem to do about the same
Splittting RNGs for Use In Parallel

- We consider splitting a single PRNG:
  - Assume \( \{x_n\} \) has \( \text{Per}(x_n) \)
  - Has the fast-leap ahead property: leaping \( L \) ahead costs no more than generating \( O(\log_2(L)) \) numbers
- Then we associate a single block of length \( L \) to each parallel subsequence:
  - Blocking:
    - First block: \( \{x_0, x_1, \ldots, x_{L-1}\} \)
    - Second: \( \{x_L, x_{L+1}, \ldots, x_{2L-1}\} \)
    - \( i \)th block: \( \{x_{(i-1)L}, x_{(i-1)L+1}, \ldots, x_{iL-1}\} \)
- The Leap Frog Technique: define the leap ahead of \( \ell = \left\lfloor \frac{\text{Per}(x_0)}{L} \right\rfloor \):
  - First block: \( \{x_0, x_\ell, x_{2\ell}, \ldots, x_{(L-1)\ell}\} \)
  - Second block: \( \{x_1, x_{1+\ell}, x_{1+2\ell}, \ldots, x_{1+(L-1)\ell}\} \)
  - \( i \)th block: \( \{x_i, x_{i+\ell}, x_{i+2\ell}, \ldots, x_{i+(L-1)\ell}\} \)

The Lehmer Tree, designed for splitting LCGs:
- Define a right and left generator: \( R(x) \) and \( L(x) \)
- The right generator is used within a process
- The left generator is used to spawn a new PRNG stream
- Note: \( L(x) = R_W(x) \) for some \( W \) for all \( x \) for an LCG
- Thus, spawning is just jumping a fixed, \( W \), amount in the sequence

Recursive Halving Leap-Ahead, use fixed points or fixed leap aheads:
- First split leap ahead: \( \frac{\text{Per}(x_i)}{2} \)
- \( i \)th split leap ahead: \( \frac{\text{Per}(x_i)}{2^i} \)
- This permits effective use of all remaining numbers in \( \{x_n\} \) without the need for \textit{a priori} bounds on the stream length \( L \)

Splittting for parallelization is not scalable:
- It usually costs \( O(\log_2(\text{Per}(x_i))) \) bit operations to generate a random number
- For parallel use, a given computation that requires \( L \) random numbers per process with \( P \) processes must have \( \text{Per}(x_i) = O((LP)^e) \)
- Rule of thumb: never use more than \( \sqrt{\text{Per}(x_i)} \) of a sequence \( \rightarrow e = 2 \)
- Thus cost per random number is not constant with number of processors!!
New Results in Parallel RNGs: Using Distinct Parameterized Streams in Parallel

- Default generator: additive lagged-Fibonacci, 
  \[ x_n = x_{n-s} + x_{n-r} \mod 2^k, \quad r > s \]
  - Very efficient: 1 add & pointer update/number
  - Good empirical quality
  - Very easy to produce distinct parallel streams

- Alternative generator #1: prime modulus LCG, 
  \[ x_n = ax_{n-1} + c \mod m \]
  - Choice: Prime modulus (quality considerations)
  - Parameterize the multiplier
  - Less efficient than lagged-Fibonacci
  - Provably good quality
  - Multiprecise arithmetic in initialization

- Alternative generator #2: power-of-two modulus LCG, 
  \[ x_n = ax_{n-1} + c \mod 2^k \]
  - Choice: Power-of-two modulus (efficiency considerations)
  - Parameterize the prime additive constant
  - Less efficient than lagged-Fibonacci
  - Provably good quality
  - Must compute as many primes as streams

Parameterization Based on Seeding

- Consider the Lagged-Fibonacci generator: 
  \[ x_n = x_{n-5} + x_{n-17} \mod 2^{32} \]
  or in general: 
  \[ x_n = x_{n-s} + x_{n-r} \mod 2^k, \quad r > s \]

- The seed is 17 32-bit integers; 544 bits, longest possible period for this linear generator is \( 2^{17 \times 32} - 1 = 2^{544} - 1 \)

- Maximal period is \( \text{Per}(x_n) = (2^{17} - 1) \times 2^{31} \)

- Period is maximal \( \iff \) at least one of the 17 32-bit integers is odd

- This seeding failure results in only even “random numbers”

- Are \( (2^{17} - 1) \times 2^{31} \times 17 \) seeds with full period

- Thus there are the following number of full-period equivalence classes (ECs):

  \[ E = \frac{(2^{17} - 1) \times 2^{31} \times 17}{(2^{17} - 1) \times 2^{31}} = 2^{31} \times 16 = 2^{496} \]

The Equivalence Class Structure

With the “standard” l.s.b., \( b_0 \):

<table>
<thead>
<tr>
<th>m.s.b.</th>
<th>l.s.b.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_{k-1} )</td>
<td>( b_{k-2} )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td></td>
</tr>
</tbody>
</table>

or a special \( b_0 \) (adjoining 1’s):

<table>
<thead>
<tr>
<th>m.s.b.</th>
<th>l.s.b.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_{k-1} )</td>
<td>( b_{k-2} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>( b_{00} )</td>
</tr>
<tr>
<td>( x_0 )</td>
<td></td>
</tr>
<tr>
<td>( x_1 )</td>
<td></td>
</tr>
</tbody>
</table>
Consider only \( x_n = ax_{n-1} \pmod{m} \), with \( m \) prime has maximal period when \( a \) is a primitive root modulo \( m \).

If \( \alpha \) and \( a \) are primitive roots modulo \( m \) then 
\[
\exists \, l \text{ s.t. } \gcd(l, m-1) = 1 \text{ and } \alpha \equiv a^l \pmod{m}
\]

If \( m = 2^{2^k} + 1 \) (Fermat prime) then all odd powers of \( \alpha \) are primitive elements also.

If \( m = 2q + 1 \) with \( q \) also prime (Sophie-Germain prime) then all odd powers (save the \( q \)th) of \( \alpha \) are primitive elements.

The Riemann hypothesis over finite-fields implies 
\[
|C(j, l)| \leq (l - 1) \sqrt{m}
\]

With Mersenne prime modulus, \( m = 2^p - 1 \) must compute \( \phi_{m-1}(k) \), the \( k \)th number relatively prime to \( m - 1 \).

Mersenne modulus: relatively easy to do modular multiplication.

Can prove (Percus and Kalos) that streams have good spectral test properties among themselves.

Best to choose \( c_i \approx \sqrt{2^k} = 2^{k/2} \).

Must compute distinct primes on the fly either with table or something like Meissel-Lehmer algorithm.
Types of random numbers and Monte Carlo Methods

Pseudorandom number generation

Quasirandom number generation

Types of pseudorandom numbers

Properties of these pseudorandom numbers

Parallelization of pseudorandom number generators

---

**Types of random numbers and Monte Carlo Methods**

**Pseudorandom number generation**

**Quasirandom number generation**

**Types of pseudorandom numbers**

**Properties of these pseudorandom numbers**

**Parallelization of pseudorandom number generators**

---

### Quality Issues in Serial and Parallel PRNGs

- Empirical tests (more later)
- Provable measures of quality:
  - Full- and partial-period discrepancy (Niederreiter) test equidistribution of overlapping \( k \)-tuples
  - Also full- \(( k = \text{Per}(x_n))\) and partial-period exponential sums:
    \[
    C(j, k) = \sum_{n=0}^{k-1} e^{\frac{2\pi i}{m}(x_n - x_{n-j})}
    \]

- For LCGs and SRGs full-period and partial-period results are similar
  - \(|C(\cdot, \text{Per}(x_n))| < O(\sqrt{\text{Per}(x_n)})\)
  - \(|C(\cdot, j)| < O(\sqrt{\text{Per}(x_n)})\)

- Additive lagged-Fibonacci generators have poor provable results, yet empirical evidence suggests
  - \(|C(\cdot, \text{Per}(x_n))| < O(\sqrt{\text{Per}(x_n)})\)

---

### Parallel Neutronics: A Difficult Example

- The structure of parallel neutronics
  - Use a parallel queue to hold unfinished work
  - Each processor follows a distinct neutron
  - Fission event places a new neutron(s) in queue with initial conditions

- Problems and solutions
  - Reproducibility: each neutron is queued with a new generator (and with the next generator)
  - Using the binary tree mapping prevents generator reuse, even with extensive branching
  - A global seed reorders the generators to obtain a statistically significant new but reproducible result

---

### Many Parameterized Streams Facilitate Implementation/Use

- Advantages of using parameterized generators
  - Each unique parameter value gives an “independent” stream
  - Each stream is uniquely numbered
  - Numbering allows for absolute reproducibility, even with MIMD queuing
  - Effective serial implementation + enumeration yield a portable scalable implementation
  - Provides theoretical testing basis
Many Parameterized Streams Facilitate Implementation/Use

- Implementation details
  - Generators mapped canonically to a binary tree
  - Extended seed data structure contains current seed and next generator
  - Spawning uses new next generator as starting point: assures no reuse of generators
- All these ideas in the Scalable Parallel Random Number Generators (SPRNG) library: http://sprng.fsu.edu

Many Different Generators and A Unified Interface

- Advantages of having more than one generator
  - An application exists that stumbles on a given generator
  - Generators based on different recursions allow comparison to rule out spurious results
  - Makes the generators real experimental tools
- Two interfaces to the SPRNG library: simple and default
  - Initialization returns a pointer to the generator state: init_SPRNG()
  - Single call for new random number: SPRNG()
  - Generator type chosen with parameters in init_SPRNG()
  - Makes changing generator very easy
  - Can use more than one generator type in code
  - Parallel structure is extensible to new generators through dummy routines

Quasirandom Numbers

- Many problems require uniformity, not randomness: “quasirandom” numbers are highly uniform deterministic sequences with small star discrepancy
- Definition: The star discrepancy $D_N^*$ of $x_1, \ldots, x_N$:
  $$D_N^* = D_N^*(x_1, \ldots, x_N) = \sup_{0 \leq u \leq 1} \left| \frac{1}{N} \sum_{n=1}^{N} \chi_{[0,u)}(x_n) - u \right|,$$
  where $\chi$ is the characteristic function

Theorem (Koksma, 1942): if $f(x)$ has bounded variation $V(f)$ on $[0, 1]$ and $x_1, \ldots, x_N \in [0, 1]$ with star discrepancy $D_N^*$, then:

$$\left| \frac{1}{N} \sum_{n=1}^{N} f(x_n) - \int_{0}^{1} f(x) \, dx \right| \leq V(f)D_N^*,$$

this is the Koksma-Hlawka inequality
- Note: Many different types of discrepancies are definable
Discrepancy Facts

- Real random numbers have (the law of the iterated logarithm):
  \[ D_N^* = O(N^{-1/2}(\log \log N)^{-1/2}) \]
- Klaus F. Roth (Fields medalist in 1958) proved the following lower bound in 1954 for the star discrepancy of \( N \) points in \( s \) dimensions:
  \[ D_N^* \geq O(N^{-1}(\log N)^{s-1/2}) \]
- Sequences (indefinite length) and point sets have different "best discrepancies" at present
  - Sequence: \( D_N^* \leq O(N^{-1}(\log N)^{s-1}) \)
  - Point set: \( D_N^* \leq O(N^{-1}(\log N)^{s-2}) \)

Some Types of Quasirandom Numbers

- Must choose point sets (finite #) or sequences (infinite #) with small \( D_N^* \)
- Often used is the van der Corput sequence in base \( b \): \( x_n = \Phi_b(n-1), n = 1, 2, \ldots \) where for \( b \in \mathbb{Z}, b \geq 2 \):
  \[ \Phi_b \left( \sum_{j=0}^{\infty} a_j b^j \right) = \sum_{j=0}^{\infty} a_j b^{-j} \quad \text{with} \quad a_j \in \{0, 1, \ldots, b-1\} \]
- Other small \( D_N^* \) points sets and sequences:
  - Halton sequence: \( x_n = (\Phi_{b_1}(n-1), \ldots, \Phi_{b_s}(n-1)), \)
    \( n = 1, 2, \ldots \), \( D_N^* = O(N^{-1}(\log N)^s) \) if \( b_1, \ldots, b_s \) pairwise relatively prime
  - Hammersley point set:
    \( x_n = \left( \frac{n-1}{N}, \Phi_{b_1}(n-1), \ldots, \Phi_{b_{s-1}}(n-1) \right), n = 1, 2, \ldots, N, \)
    \( D_N^* = O(N^{-1}(\log N)^{s-1}) \) if \( b_1, \ldots, b_{s-1} \) are pairwise relatively prime

For the van der Corput sequence
\[
ND_N^* \leq \frac{\log N}{3\log 2} + O(1)
\]

With \( b = 2 \), we get \( \{1/2, 1/4, 3/8, 7/8, 5/8, 3/8, \ldots\} \)

With \( b = 3 \), we get \( \{1/3, 2/9, 1/9, 4/9, 7/9, 2/9, 5/9, 8/9, \ldots\} \)
Some Types of Quasirandom Numbers

- Ergodic dynamics: \( x_n = \{n\alpha\} \), where \( \alpha = (\alpha_1, \ldots, \alpha_s) \) is irrational and \( \alpha_1, \ldots, \alpha_s \) are linearly independent over the rationals then for almost all \( \alpha \in \mathbb{R}^s \),
  \[ D_N^* = O(N^{-1}(\log N)^{s+1+\epsilon}) \] for all \( \epsilon > 0 \)

- Other methods of generation
  - Method of good lattice points (Sloan and Joe)
  - Sobol sequences
  - Faure sequences
  - Niederreiter sequences

Some Types of Quasirandom Numbers

- Another interpretation of the v.d. Corput sequence:
  - Define the \( i \)-th \( \ell \)-bit “direction number” as: \( v_i = 2^\ell \) (think of this as a bit vector)
  - Represent \( n - 1 \) via its base-2 representation
    \[ n - 1 = b_{i-1}b_{i-2}\ldots b_1b_0 \]
  - Thus we have
    \[ \Phi_2(n - 1) = 2^{-\ell} \bigoplus_{i = 0}^{i = \ell - 1} v_i \]

- The Sobol sequence works the same!!
  - Use recursions with a primitive binary polynomial define the (dense) \( v_i \)
  - The Sobol sequence is defined as:
    \[ s_n = 2^{-\ell} \bigoplus_{i = 0}^{i = \ell - 1} v_i \]
  - For speed of implementation, we use Gray-code ordering

- \((t, m, s)\)-nets and \((t, s)\)-sequences and generalized Niederreiter sequences

- Let \( b \geq 2 \), \( s > 1 \) and \( 0 \leq t \leq m \in \mathbb{Z} \) then a \( b\)-ary box,
  \[ J \subset [0, 1)^s \], is given by
  \[ J = \prod_{i=1}^{s} \left[ \frac{a_i}{b^{d_i}}, \frac{a_i + 1}{b^{d_i}} \right) \]
  where \( d_i \geq 0 \) and the \( a_i \) are \( b\)-ary digits, note that
  \[ |J| = b^{-\sum_{i=1}^{s} d_i} \]

A set of \( b^m \) points is a \((t, m, s)\)-net if each \( b\)-ary box of volume \( b^{t-m} \) has exactly \( b^t \) points in it

Such \((t, m, s)\)-nets can be obtained via Generalized Niederreiter sequences, in dimension \( j \) of \( s \):
\[ y^{(j)}_i(n) = C^{(j)}a_i(n) \], where \( n \) has the \( b\)-ary representation
\[ n = \sum_{k=0}^{\infty} a_k(n) b^k \] and \[ x^{(j)}_i(n) = \sum_{k=1}^{m} y^{(j)}_k(n) q^{-k} \]
Types of random numbers and Monte Carlo Methods
Pseudorandom number generation
Quasirandom number generation

The Koksma-Hlawka inequality
Discrepancy
The van der Corput sequence
Methods of quasirandom number generation

Future Work on Random Numbers

**SPRNG** and pseudorandom number generation work

- New generators: Well, Mersenne Twister, different LCGs, etc.
- Spawn-intensive/small-memory footprint generators
- More comprehensive testing suite
- Improved theoretical tests
- C++ implementation
- Grid-based tools

**Quasirandom number work**

- Scrambling (parameterization) for parallelization
- Optimal scrambling
- Comparison to sparse grids
- “QPRNG"
- Grid-based tools

For Further Reading I

- [Y. Li and M. Mascagni (2005)]
  Grid-based Quasi-Monte Carlo Applications,

- [H. Chi, M. Mascagni and T. Warnock (2005)]
  On the Optimal Halton Sequence,

- [M. Mascagni and H. Chi (2004)]
  Parallel Linear Congruential Generators with Sophie-Germain Moduli,
[M. Mascagni and A. Srinivasan (2004)]
Parameterizing Parallel Multiplicative Lagged-Fibonacci
Generators,

[M. Mascagni and A. Srinivasan (2000)]
Algorithm 806: SPRNG: A Scalable Library for
Pseudorandom Number Generation,

© Michael Mascagni, 2005-2006