Random Number Generation

Stephen Booth
David Henty
Random numbers are frequently used in many types of computer simulation

Frequently as part of a sampling process:

- Generate a representative sample of a large population by choosing members at random.
- Monte-carlo integration is approximating an integral by sampling the function at random points.
- Even when simulating a stochastic process (random walk/random events etc.) we are sampling the possible evolutions of the system.
What is Random anyway

• “Random” is actually a very difficult philosophical concept.
• However in most cases the real requirement is “unbiased sampling” which is more straightforward.
Random numbers are chosen from a probability distribution.

For random integers each possible result $X$ occurs with a probability $P(X)$.

For random real numbers $R$ this becomes a probability density $P(R)$.

- Chance of the results occurring within a region is the integral of the probability density over that region.
- Most generators are designed to generate a “uniform” distribution.
  - $P(R) = 1 \quad 0 < R < 1$
  - $P(R) = 0 \quad$ elsewhere
- Other distributions then generated from this
Resolution

• However computers use floating point numbers not true real numbers.
  – Only a finite set of possible values can be represented.
  – Any random “real” number must come from this set.

• Most techniques generate an even smaller sub-set of values e.g.
  – $R = X/N$  where $X$ is a random integer between 0 and $N$
  – $1/N$ is the resolution of that generator.

• Generated distribution is only an approximation to uniform.
  – May bias the results if you are not careful
  – Always worth understanding the resolution of the generator you use.
Correlations

• True Random numbers are also un-correlated with each other.

• The probability of getting a particular set of random results should be the product of the probabilities of each result in isolation.
You can build hardware random number generators.

- These work by taking measurements of some random physical process
  - Thermal noise
  - Quantum processes.
- Problems
  - Debugging is very hard as can never reproduce the same program run twice.
  - May still suffer from limited resolution
  - Often quite slow.
  - May not result in any visible improvement in quality of results.
- More commonly used in cryptographic applications.
Pseudo Random Numbers

- Pseudo Random Numbers are a deterministic sequence of numbers generated by some algorithm that are used in-place of true random numbers.
- Aim is for the sequence to share enough of the statistical properties of true random numbers to not bias the results.
- PRNs are NOT random. It is always possible to come up with some test that demonstrates this.
• Quality of a PRNG sequence depends on the intended use.
  – Each use case only depends on some of the statistical properties of true random numbers.
  – Some generators may introduce problems for some calculations but not others.
• In practice, algorithms exist that can stand in for true random numbers for most common types of simulation.
• Unfortunately language default generators are often fairly poor.
Structure of a PRNG

- Logically PRNGs consist of:
  - An internal state $S_i$
  - An update transform $T S_i \rightarrow S_{i+1}$ that maps one state onto the next
  - An output transform $F S_i \rightarrow X_i$ that generates the next number in the PRNG sequence from the current state.

- Algorithms are rated on the statistical properties of the output sequence
  - Speed of execution and memory consumption are also important.

- Different algorithms may generate the same PRNG sequence via different state representations and transforms.
Seeding the Generator

- Also need some mechanism of initialising the starting state.
- Traditional algorithms only used a single word of state so many programs assume the generator is initialised using a single integer.
- If you don’t set a starting seed you either:
  1. Get the same sequence every time you run the program.
  2. The generator seeds from the current time (makes debugging hard).
- If your program checkpoints remember to save the RNG state so you can restart exactly where you left off.
  - Write tests to check this!
Period of a generator

- There are only a finite number of possible states.
- Eventually generator will return to its starting state.
- The update transform should generate a cyclic group
  \[ T^{\text{period}} = I \]
- The size of this group is the \textit{period} of the generator.
- It is also the number of valid states.
  - If the update transform does not form a cyclic group then state information can be lost initially but the generator will eventually settle down into a cycle of recurring states.
How state is stored

• In principle you could store the position in the sequence.
  – Update transform is just an increment $i \rightarrow i + 1$
  – All the randomness is in the output-transform.
  – Need very expensive output-transform to have good randomness properties.

• In practice use state representations that approximate random values and keep the output transform simple.
  – Even fairly simple (inexpensive) update transforms can have good randomness properties
PRNG Algorithms

• PRNG Algorithms are deceptively simple.
  – Usually made up from a few simple operations.
  – Typically bitwise operations or modular arithmetic.

• Very tempting to try and “Improve” on published algorithms

• **DON’T DO THIS** unless you really know what you are doing.

• Each new algorithm requires theoretical (Number theory) analysis to determine the period of the generator.
  – Many other statistical properties can also be derived theoretically.
Selecting Generators

- Most generators are selected based on the properties of small sets of consecutive numbers from the sequence.
  - \{ X_i, X_{i+1} \} approximate a pair of random number.
- Non consecutive sets may appear less random.
  - E.g \{ X_i, X_{i+1024} \}
- Consecutive sets important for most applications (especially when used to generate non-uniform distributions) so this is a good heuristic for general purpose generators.
- For a specific application may be other correlations that are equally important.
• Selection uses a combination of theory and statistical tests.
• Statistical tests augment theory, not good enough by themselves.
Linear Congruential Generators

- \[ S_{i+1} = (a S_i + c) \mod M \]
- If \( a, c \) and \( M \) chosen correctly, has \( M \) possible states.
- If \( c = 0 \) then \( (M-1) \) possible states (\( S=0 \) always maps to itself).
Java.util.Random

- Optimised for speed not quality
  - \( a = 0x5DEECE66DL \)
  - \( C = 0xBL \)
  - \( M = 2^{48} \)

- Mod \( 2^{48} \) is a bit-mask so very fast.

- 47 bits of state in total.
  - However bit-\( n \) of the state has repeats with at most period \( 2^{n+1} \)
    - bit-0 period 2
    - bit-1 period 4
  - Only the high order bits repeat with any degree of randomness.
  - Class only exposes the top 32-bits to the user making it ok for quick and dirty use.
MRGs

- LCG are a special case of Multiply Recursive Generators
  \[ S_n = a_1 S_{n-1} + \ldots + a_k S_{n-k} \mod M \]
  - Needs array of state variables.

- Some number theory ...
  \[ M = P^q \text{ with } P \text{ prime then maximum period is } P^{q-1}(P^k - 1). \]
  - Special values of \{a_k\} generate full period if M is prime.

- Many other common generators are special cases of these.
Other Common generators

• LFSR
  – Linear Feedback Shift Register
  – M=2  Note XOR is the same as multiplication mod 2

• Lagged Fibonacci Generators
  – Only 2 Values of \{ a_k \} non-zero so faster then the general case.

• Mersenne-Twister appears quite different but is equivalent to a MRG with M=2 and \((2^k - 1)\) a Mersenne prime.
Non uniform distributions

• Non-uniform distributions are constructed out of (multiple) normally distributed values.
• For any probability distribution \( p(x) \)
  - \( \int_{\min}^{\max} p(x) = 1 \)
  - Selecting small areas under the curve uniformly is the same as selecting \( x \) with probability \( p(x) \)
  - Inverse transform sampling.
    - Divide area into thin strips of equal area and select strip at random.
  - Rejection sampling
    - Choose \( x,y \) points at random but reject points above the curve i.e. \( y > p(x) \)
Simple example

- \( p(x) = 3 \ x^2 \)
Inverse transform sampling

• Generally quite hard to do:
  – Generate uniform deviate $U$.
  – Return $x : \int_{\min}^{x} p(y) dy = U$

• Only analytically solvable for certain distributions.
  – e.g for $p(x) = 3x^2$
  – $x = \sqrt[3]{U}$ (cube root of $U$)
  – 100,000 samples & 100 bins

```plaintext
call random_number(myrng)
myrng = myrng**(1.0/3.0)
```
Rejection sampling

- Only need to be able to evaluate \( p(x) \).
  - Needs special handling for unbounded distributions.

- e.g for \( p(x) = 3 \times x^2 \)

```python
    call random_number(myrng1)
call random_number(myrng2)
myrng2 = 3.0*myrng2
if (myrng2 < 3.0*myrng1**2)
    myrng = mynrng1
```
Generating Gaussians

- Most commonly required non-uniform distribution is the normal / gaussian distribution

\[ P(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{\frac{-x^2}{2\sigma^2}} \]

- e.g for \( \sigma = 0.5 \)
Box Muller

• Generates pairs of gaussians from pairs of uniform.
  – Generate 2 Uniform random numbers U,V from (0:1]
    – \( X = \sqrt{-2 \ln U} \cdot \cos(2\pi V) \)
    – \( Y = \sqrt{-2 \ln U} \cdot \sin(2\pi V) \)

• Generally quite slow due to math library functions.

• With care an be vectorised so may be better algorithm for GPGPU.
Polar method

• Variation of box-muller that uses accept-reject step instead of trig functions.

1. \( a = 2U - 1 \)
2. \( b = 2V - 1 \)
3. \( s = a^2 + b^2 \)
4. If \( s > 1 \) goto (1)
5. \( X = a \sqrt{\frac{-2 \ln(s)}{s}} \)
6. \( Y = b \sqrt{\frac{-2 \ln(s)}{s}} \)

• Usually faster overall but accept/reject inhibits vectorisation
Summary

• (Pseudo) random numbers are key to many algorithms
  – a number of high quality algorithms exist

• Typically generate number in the range \([0.0, 1.0)\)

• Are often transformed to other distributions
  – analytically
  – using accept-reject stage

• Repeatability is a key requirement
  – necessary to test correctness of any computation