Simulation is the modeling of real world systems to test their behavior under various conditions. We are going to be concerned with computer simulations that involve some type of random behavior or inputs. Simulations that use randomness are called stochastic simulations.

Why would we use stochastic simulation?

- System is too dangerous to test in real world (explosions, nuclear reactions)
- Prediction of systems that may take a long time (climate effects, economics)
- Testing reliability of systems before they are deployed (bridges, airplanes, hardware, software)
- We want to make calculations that are too complex to do deterministically
- Graphics, physics-based animation, particle systems, ray-tracing (video games!)
  (see http://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels/)

Famous example: The ENIAC was the first general-purpose, electronic computer. It was used in the 1940’s to simulate hydrogen fusion (the physics behind the hydrogen bomb). This was done using a new type of stochastic simulation called the Monte Carlo Method, developed by Enrico Fermi and Stanislaw Ulam. The random number generation was developed by John von Neumann. Here is an interesting history of these first developments of stochastic computer simulations: [http://library.lanl.gov/la-pubs/00326866.pdf](http://library.lanl.gov/la-pubs/00326866.pdf)

**Pseudorandom Number Generation.** How would you generate a random number on a computer? Not so easy! Software is always deterministic! The best we can do in software is to generate a sequence of “pseudo” random numbers, that is, a sequence of numbers that “look” random even though they are deterministic. A pseudorandom number generator (PRNG) will generate a sequence of numbers that appear to be randomly and uniformly distributed integers between 0 and some maximum. Dividing by this maximum will give a floating point number that looks like a uniform random variable on [0, 1]. But what makes a “good” PRNG, that is, what types of sequences would fool us into thinking they are random? Here are some possible criteria:

1. The random sequence should not repeat itself (at least not for a long time).
2. The random sequence should be spread uniformly over the interval (no gaps, no piling up in a particular place).
3. Any number in the sequence should not appear to be dependent on previous numbers in the sequence (although since it is deterministic, it really does depend on previous numbers).
4. One property important in cryptography, and related to the last property, is that a random number should not be easy to guess from previous numbers in the sequence.
PRNG Methods:
All methods generate a sequence \(x_1, x_2, \ldots, x_N\). Initialized with a seed, \(x_0\), typically not included in the output sequence.

von Neumann’s Middle Square Method:
Square previous number, take the middle bits (\(n/2\) bits of an \(n\)-bit number).
\[
x_{k+1} = \left(\frac{x_k^2}{2^n}\right) \mod 2^n
\]

Linear Congruential Generator (LCG):
[https://en.wikipedia.org/wiki/Linear_congruential_generator](https://en.wikipedia.org/wiki/Linear_congruential_generator)

Choose constant numbers \(a, c, m\).
\[
x_{k+1} = (ax_k + c) \mod m
\]
This is one of the most commonly used PRNG’s. It is the `rand()` function in C/C++, Random class in Java, Matlab, etc. It has some bad properties that make it unacceptable for simulations where accuracy is critical or for cryptography/security applications. Namely, it shows structure in the sequence (lack of independence), and can have a relatively short period (repeats itself). Depending on the values of \(a, c, m\), this can be made better or worse. A very infamous LCG was RANDU, widely implemented and used in the 60’s and 70’s. It was so bad that many results that used it were later considered suspect.

Mersenne Twister and Blum Blum Shub:
[https://en.wikipedia.org/wiki/Mersenne_Twister](https://en.wikipedia.org/wiki/Mersenne_Twister)
[https://en.wikipedia.org/wiki/Blum_Blum_Shub](https://en.wikipedia.org/wiki/Blum_Blum_Shub)

Complicated algorithms (see Wikipedia pages for details), but much better PRNGs with nice properties. Mersenne twister is the default in R and maybe the most widely used PRNG. Good for accurate simulations, but still not considered safe for security applications. Blum Blum Shub is not appropriate for simulations (too slow), but it is considered safe for security.

Generating from Distributions Other Than Uniform:
Consider that we have a random variable \(X\) (remember this is a function), and we want to generate random numbers that follow the distribution of \(X\). These numbers will be called realizations of the random variable \(X\), and we will denote them with lower case letters, such as \(x\).

Discrete:
Given a discrete random variable \(X\) that takes values \(a_1, a_2, \ldots, a_n\), we can generate a realization of \(X\) from a realization of a uniform random variable \(U \sim \text{Unif}(0, 1)\) as follows. Let \(u\) be the realization of \(U\) and \(x\) the realization of \(X\).

\[
x = a_k, \quad \text{where } k \text{ is the smallest number such that } P(X \leq a_{k-1}) \leq u < P(X \leq a_k)
\]

Example: \(X \sim \text{Ber}(0.5)\)
1. Generate a \(\text{Unif}(0, 1)\) realization \(u\).
2. Generate \(X\) realization: \(x = 0\) if \(u < 0.5\), otherwise \(x = 1\).

Continuous:
Given a continuous random variable \(X\) with CDF \(F(a)\), we use the inverse of the CDF to generate a realization of \(X\). Again, starting with a \(\text{Unif}(0, 1)\) realization, \(u\):
\[ x = F^{-1}(u) \]

See derivation in Ch 6 for why this works.

Example: exponential distributed random variable (from book).

**Generating Gaussian Random Numbers: The Box-Muller Method**

https://en.wikipedia.org/wiki/Box-Muller_transform

Another way to generate random numbers from a distribution is called the *transformation method*. This means that you generate uniform pseudorandom numbers and then use some type of transformation to map them to another type of distribution. The Box-Muller Transform is one type of transformation method. It generates Gaussian random numbers from a \( \mathcal{N}(0,1) \) distribution. It works as follows

1. Generate \( u_1 \) and \( u_2 \) as independent uniform random values, that is, from \( \text{Unif}(0,1) \).

2. Transform them:

\[
\begin{align*}
    z_1 &= \sqrt{-2 \ln(u_1)} \cos(2\pi u_2) \\
    z_2 &= \sqrt{-2 \ln(u_1)} \sin(2\pi u_2).
\end{align*}
\]

In the end, \( z_1 \) and \( z_2 \) will be independent random values from a standard normal distribution, \( \mathcal{N}(0,1) \). If you want random variables from a general normal distribution with mean \( \mu \) and variance \( \sigma \), then you just adjust them as

\[
\begin{align*}
    y_1 &= \sigma z_1 + \mu, \\
    y_2 &= \sigma z_2 + \mu.
\end{align*}
\]

Now \( y_1, y_2 \sim \mathcal{N}(\mu, \sigma^2) \).