**MAT 3103: Computational Statistics and Probability**

**Chapter 12: Monte Carlo Methods**


**Simulation:**

Another way of thinking about sample spaces, and randomness in general, is through the notion of simulation. Simulation is what if we asked a computer tossed a fair coin 30 times and again asked the computer to repeat the process; we would get different 30 coin-tosses. This procedure is the exact same one we imagine nature (or casino equipment) follows whenever a non-deterministic situation is involved. The difference is, of course, that if we use the random walk to model out winnings in a fair gamble, it is much cheaper and faster to use the computer than to go out and stake (and possibly loose) large amounts of money. Another obvious advantage of the simulation approach is that it can be repeated; a simulation can be run many times and various statistics (mean, variance, etc.) can be computed.

More technically, every simulation involves two separate inputs. The first one if the actual sequence of outcomes of coin-tosses. The other one is the structure of the model - I have to teach the computer to "go up" if heads shows and to "go down" if tails show, and to repeat the same procedure several times. In more complicated situations this structure will be more complicated. What is remarkable is that the first ingredient, the coin-tosses, will stay almost as simple as in the random walk case, even in the most complicated models. In fact, all we need is a sequence of so-called random numbers. If we can get my computer to produce an independent sequence of uniformly distributed numbers between 0 and 1(these are the random numbers) we can simulate trajectories of all important stochastic processes. Just to start you thinking, here is how to produce a coin-toss from a random number: declare heads if the random number drawn is between 0 and 0.5, and declare tails otherwise.

## 1. Random number generation:

Before we get into intricacies of simulation of complicated processes, let us spend some time on the (seemingly) simple procedure of the generation of a single random number between 0 and 1. In practice, we can get quite close.

Suppose we have written a computer program, a random number generator (RNG); call it **rand**- which produces a random number between 0 and 1every time we call it. So far, there is nothing that prevents rand from always returning the same number 0.4, or from alternating between0.3 and 0.83. Such an implementation of rand will, however, hardly qualify for an RNG since the values it spits out come in a predictable order. We should, therefore, require any candidate for a random number generator to produce a sequence of numbers which is as unpredictable as possible. This is, admittedly, a hard task for a computer having only deterministic functions in its arsenal, and that is why the random generator design is such a difficult field. The state of the affairs is that we speak of good or less good random number generators, based on some statistical properties of the produced sequences of numbers.

One of the most important requirements is that our RNG produce uniformly distributed numbers in [0, 1], namely; the sequence of numbers produced by rand will have to cover the interval [0, 1] evenly, and, in the long run, the number of random numbers in each sub interval [a, b] of [0, 1] should be proportional to the length of the interval $b - a$.

Another problem with RNGs is that the numbers produced will start to repeat after a while (this is a fact of life and finiteness of your computer's memory). The number of calls it takes for a RNG to start repeating its output is called the period of a RNG. You might have wondered how is it that an RNG produces a different number each time it is called, since, after all, it is only a function written in some programming language. Most often, RNGs use a hidden variable called the random seed which stores the last output of rand and is used as an (invisible) input to the function rand the next time it is called. If we use the same seed twice, the RNG will produce the same number, and so the period of the RNG is limited by the number of possible seeds. It is worth remarking that the actual random number generators usually produce a "random" integer between0and some large number RAND_MAX, and report the result normalized (divided) by RAND_MAX to get a number in [0, 1).

## 2. Simulation of Random Variables:

Having found a random number generator good enough for our purposes, we might want to use it to simulate random variables with distributions different from the uniform on [0, 1] (coin-tosses, normal, exponential, . . . ). This is almost always achieved through transformations of the output of a RNG, and we will present several methods for dealing with this problem.

Following is a list of procedures commonly used to simulate popular random variables:

### a) Discrete Random Variables:

Let X have a discrete distribution given by

$$X \sim \begin{pmatrix} x_1 & x_2 & \dots & x_n \\ p_1 & p_2 & \dots & p_n \end{pmatrix}$$

For discrete distributions taking an infinite number of values we can always truncate at a very large $n$ and approximate it with a distribution similar to the one of $X$. We know that the probabilities $p_1, p_2, \dots, p_n$ add-up to 1, so we define the numbers $0 = q_0 < q_1 < \cdots < q_n = 1$by

$$q_0 = 0,\ q_1 = p_1,\ q_2 = p_1 + p_2,\ \dots\ q_n = p_1 + p_2 + \cdots + p_n = 1.$$

To simulate our discrete random variable $X$, we call **rand** and then return $x_1$ if $0 \le \text{rand} < q_1$, return $x_2$ if $q_1 \le$ **rand** $< q_2$, and so on. It is quite obvious that this procedure indeed simulates a random variable $X$. The transformation function $f$ is in this case given by

$$f(x) = \begin{cases} x_1, & 0 \le x < q_1 \\ x_2, & q_1 \le x < q_2 \\ & \dots \\ x_n, & q_{n-1} \le x \le 1 \end{cases}.$$

### b) The Method of Inverse Functions:

The basic observation in this method is that , for any continuous random variable $X$ with the distribution function $F_X$, the random variable $Y = F_X(X)$ is uniformly distributed on [0, 1]. By inverting the distribution function $F_X$ and applying it to $Y$, we recover $X$. Therefore, if we wish to simulate a random variable with an invertible distribution function $F$, we first simulate a uniform random variable on [0, 1] (using **rand**) and then apply the function $F^{-1}$ to the result. In other words, use $f = F^{-1}$ as the transformation function. Of course, this method fails if we cannot write $F^{-1}$in closed form.

**Example 12.1:** (Exponential Distribution) Let us apply the method of inverse functions to the simulation of an exponentially distributed random variable $X$ with parameter $\lambda$. Remember that the density $f_X$ of $X$ is given by

$$f_X(x) = \lambda e^{-x\lambda},\ x > 0, \text{ and so } F_X(x) = 1-e^{-x\lambda},\ x > 0,$$

and so $F_X^{-1}(y) = -\frac{1}{\lambda}\log(1-y)$. Since, 1− **rand** has the same U[0, 1] distribution as rand, we conclude that $f(x) = -\frac{1}{\lambda}\log(x)$ works as a transformation function in this case, i.e., that

$$-\frac{log(\mathbf{rand})}{\lambda}$$

has the required Exp (λ) distribution.

**c) The Box-Muller method:**

This method is useful for simulating normal random variables; since for them the method of inverse function fails (there is no closed-form expression for the distribution function of a standard normal). Note that this method does not fall under that category of transformation function methods as described above.

**Proposition 12.1** Let $\gamma_1$ and $\gamma_2$ be independent U[0, 1] distributed random variables. Then the random variables

$$X_1 = \sqrt{-2\log(\gamma_1)}\cos(2\pi\gamma_2),\ X_2 = \sqrt{-2\log(\gamma_1)}\sin(2\pi\gamma_2)$$

are independent and standard normal (N(0,1)).

Therefore, in order to simulate a normal random variable with mean $\mu = 0$ and variance $\sigma^2 = 1$, we produce call the function **rand** twice to produce two random numbers **rand1** and **rand2**. The numbers

$$X_1 = \sqrt{-2\log(rand1)}\cos(2\pi\, rand2),\ X_2 = \sqrt{-2\log(rand1)}\sin(2\pi\, rand2)$$

will be two independent normals. Note that it is necessary to call the function rand twice, but we also get two normal random numbers out of it. It is not hard to write a procedure which will produce 2 normal random numbers in this way on every second call, return one of them and store the other for the next call. In the spirit of the discussion above, the function $f = (f_1, f_2)$: $(0,1]\times[0,1]\rightarrow R^2$ given by

$$f_1(x,y) = \sqrt{-2\log(x)}\cos(2\pi y),\ f_2(x,y) = \sqrt{-2\log(x)}\sin(2\pi y).$$

can be considered a transformation function in this case.

**d) Method of the Central Limit Theorem:**

The following algorithm is often used to simulate a normal random variable:

(a) Simulate 12 independent uniform random variables (rands) $\gamma_1, \gamma_2, ..., \gamma_{12}$.

(b) Set $X = \gamma_1 + \gamma_2 + \cdots + \gamma_{12} - 6$.

The distribution of $X$ is very close to the distribution of a unit normal, although not exactly equal (e.g. $P[X > 6] = 0$, and $P[Z > 6] \neq 0$, for a true normal Z). The reason why $X$ approximates the normal distribution well comes from the following theorem.

**Theorem 12.1:** Let $X_1, X_2,...$ be a sequence of independent random variables, all having the same (square-integrable) distribution. Set $\mu = E[X_1]$ (= $E[X_2]$ = ...) and $\sigma^2 = Var[X_1]$ (= $Var[X_2]$ = ...). The sequence of normalized random variables

$$\frac{(X_1 + X_2 + \cdots + X_n) - n\mu}{\sigma\sqrt{n}},$$

converges to the normal random variable (in a mathematically precise sense). The choice of exactly 12 **rands** (as opposed to 11 or 35) comes from practice: it seems to achieve satisfactory performance with relatively low computational cost. Also, the standard deviation of a U[0, 1] random variable is $\frac{1}{\sqrt{12}}$, so the denominator $\frac{\sigma}{\sqrt{n}}$ conveniently becomes 1 for $n = 12$. It might seem a bit wasteful to use 12 calls of **rand** in order to produce one draw from the unit normal. If you try it out, you will see, however, that it is of comparable speed to the Box-Muller method described above; while Box-Muller uses computationally expensive cos, sin, √ and log, this method uses only addition and subtraction. The final verdict of the comparison of the two methods will depend on the architecture you are running the code on, and the quality of the implementation of the functions cos, sin, ....

## 3. Monte Carlo Simulation:

Monte Carlo simulation is a computerized mathematical technique that allows people to account for risk in quantitative analysis and decision making. The technique is used by professionals in such widely disparate fields as finance, project management, energy, manufacturing, engineering, research and development, insurance, oil & gas, transportation, and the environment.

Monte Carlo simulation furnishes the decision-maker with a range of possible outcomes and the probabilities they will occur for any choice of action. It shows the extreme possibilities the outcomes of going for broke and for the most conservative decision along with all possible consequences for middle-of-the-road decisions.

Monte Carlo simulation performs risk analysis by building models of possible results by substituting a range of values a probability distribution for any factor that has inherent uncertainty. It then calculates results over and over, each time using a different set of random values from the probability functions. Depending upon the number of uncertainties and the ranges specified for them, a Monte Carlo simulation could involve thousands or tens of thousands of recalculations before it is complete. Monte Carlo simulation produces distributions of possible outcome values.

During a Monte Carlo simulation, values are sampled at random from the input probability distributions. Each set of samples is called an iteration, and the resulting outcome from that sample is recorded. Monte Carlo simulation does these hundreds or thousands of times, and the result is a probability distribution of possible outcomes. In this way, Monte Carlo simulation provides a much more comprehensive view of what may happen. It tells you not only what could happen, but how likely it is to happen. Monte Carlo simulation provides a number of advantages over deterministic, or "single-point estimate" analysis:

- Probabilistic Results- Results show not only what could happen, but how likely each outcome is.
- Graphical Results- Because of the data a Monte Carlo simulation generates, it's easy to create graphs of different outcomes and their chances of occurrence. This is important for communicating findings to other stakeholders.
- Sensitivity Analysis- With just a few cases, deterministic analysis makes it difficult to see which variables impact the outcome the most. In Monte Carlo simulation, it's easy to see which inputs had the biggest effect on bottom-line results.
- Scenario Analysis- In deterministic models, it's very difficult to model different combinations of values for different inputs to see the effects of truly different scenarios. Using Monte Carlo simulation, analysts can see exactly which inputs had which values together when certain outcomes occurred. This is invaluable for pursuing further analysis.

- Correlation of Inputs- In Monte Carlo simulation, it's possible to model interdependent relationships between input variables. It's important for accuracy to represent how, in reality, when some factors go up, others go up or down accordingly.

**Monte Carlo Integration:**

Having described some of the procedures and methods used for simulation of various random objects (variables, vectors, processes), we turn to an application in probability and numerical mathematics. We start off by the following version of the Law of Large Numbers which constitutes the theory behind most of the Monte Carlo applications

**Theorem 12.2** (Law of Large Numbers)

Let $X_1$, $X_2$, ... be a sequence of identically distributed random variables, and let g:R→R be function such that $\mu = E[g(X_1)]$ (= $E[g(X_2)]$ = ...) exists. Then

$$\frac{g(X_1) + g(X_2) + \cdots + g(X_n)}{n} \to \mu = \int_{-\infty}^{\infty} g(x) f_{X_1}(x) dx, as\ n \to \infty$$

The key idea of Monte Carlo integration is the following

Suppose that the quantity $y$ we are interested in can be written as $y = \int_{-\infty}^{\infty} g(x) f_X(x) dx$ for some random variable $X$ with density $f_X$ and come function $g$, and that $x_1$, $x_2$, ... are random numbers distributed according to the distribution with density $f_X$. Then the average

$$\frac{g(X_1)+g(X_2)+\cdots+g(X_n)}{n},$$

will approximate $y$. It can be shown that the accuracy of the approximation behaves like $\frac{1}{\sqrt{n}}$, so that you have to quadruple the number of simulations if you want to double the precision of you approximation.

**Example 12.2**

1. (numerical integration)

Let $g$ be a function on [0, 1]. To approximate the integral $\int_0^1 g(x)dx$ we can take a sequence of $n$(U[0,1]) random numbers $x_1$, $x_2$, ...,

$$\int_0^1 g(x)dx \approx \frac{g(X_1)+g(X_2)+\cdots+g(X_n)}{n},$$

because the density of $X$∼U[0,1] is given by

$$f_X(x) = \begin{cases} 1, & 0 \le x \le 1, \\ 0, & \text{otherwise} \end{cases}$$

2. (estimating probabilities)

Let $Y$ be a random variable with the density function $f_Y$. If we are interested in the probability P[Y∈[a, b]] for some a < b, we simulate $n$ draws $y_1, y_2, ..., y_n$ from the distribution $F_Y$ and the required approximation is

$$P[Y \in [a, b]] \approx \frac{number\ of\ y_n's\ falling\ in\ the\ interval[a,b]}{n}.$$

One of the nicest things about the Monte Carlo method is that even if the density of the random variable is not available, but you can simulate draws from it, you can still preform the calculation above and get the desired approximation. Of course, everything works in the same way for probabilities involving random vectors in any number of dimensions.

3. (approximating π)

We can devise a simple procedure for approximating π ≈ 3.141592 by using the Monte-Carlo method. All we have to do is remember that π is the area of the unit disk. Therefore, $\frac{\pi}{4}$ equals to the portion of the area of the unit disk lying in the positive quadrant, and we can write

$$\frac{\pi}{4} = \int_o^1 \int_0^1 g(x, y) dx dy,$$

where

$$g(x, y) = \begin{cases} 1, & x^2 + y^2 \le 1 \\ 0, & \text{otherwise} \end{cases}$$

So, simulate $n$ pairs ($x_i$, $y_i$), $i$ = 1... $n$ of uniformly distributed random numbers and count how many of them fall in the upper quarter of the unit circle, i.e. how many satisfy $x_i^2 + y_i^2 \le 1$, and divide by $n$. Multiply the result by 4, and it should be close to π.

**12.3 Worked Example of Monte Carlo Sampling:**

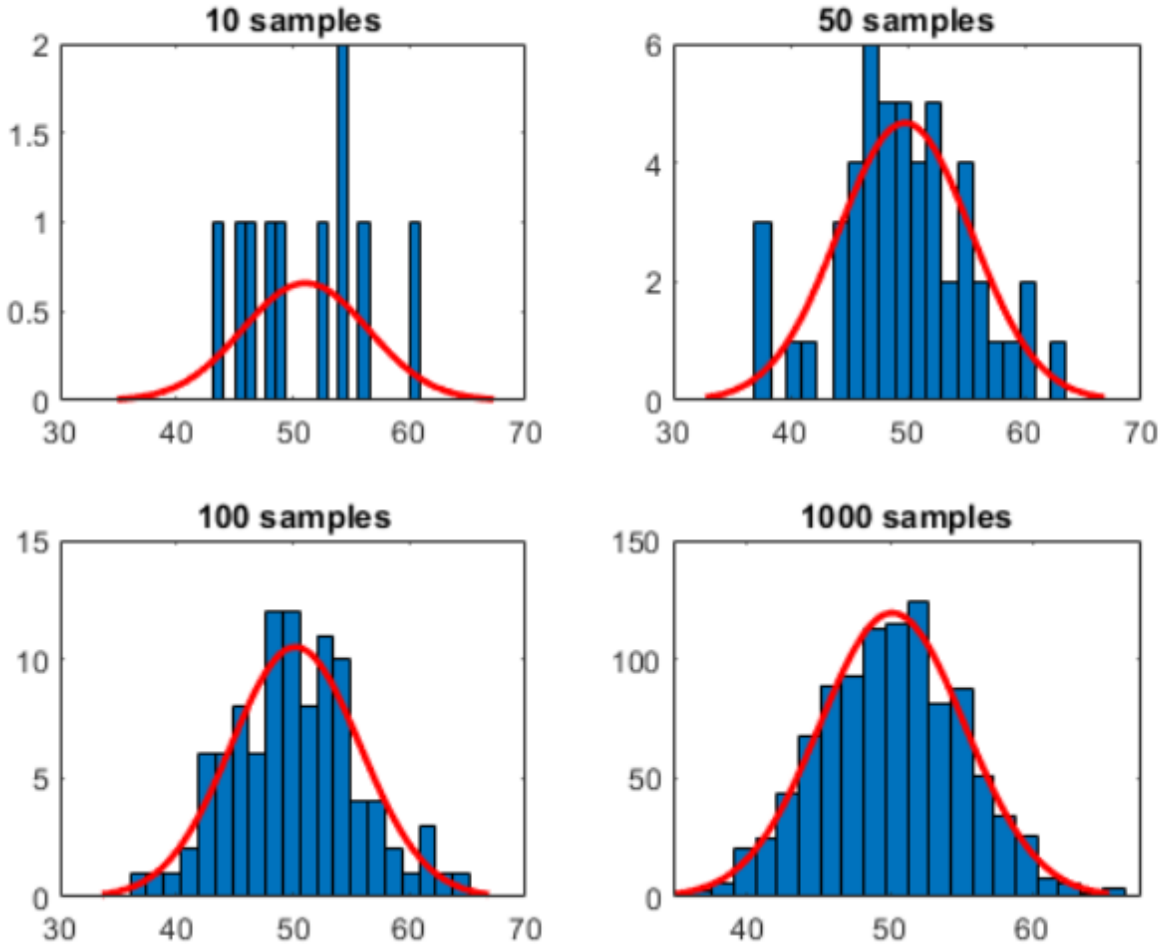
In this case, we will have a function that defines the probability distribution of a random variable. We will use a Gaussian distribution with a mean of 50 and a standard deviation of 5 and draw random samples from this distribution. Let's pretend we don't know the form of the probability distribution for this random variable and we want to sample the function to get an idea of the probability density. We can draw a sample of a given size and plot a histogram to estimate the density. The normal() NumPy function can be used to randomly draw samples from

a Gaussian distribution with the specified mean (*mu*), standard deviation (*sigma*), and sample size.

To make the example more interesting, we will repeat this experiment four times with different sized samples. We would expect that as the size of the sample is increased, the probability density will better approximate the true density of the target function, given the law of large numbers.

```
size = [10 50 100 1000];
mu = 50;
sigma = 5;
bin = 20;
for i = 1:length(size)
    r = normrnd(mu,sigma,[1,size(i)]);
    subplot(2,2,i)
    histfit(r,bin,'normal')
    title([num2str(size(i)) ' samples'])
end
```

Running the example creates four differently sized samples and plots a histogram for each. We can see that the small sample sizes of 10 and 50 do not effectively capture the density of the target function. We can see that 100 samples is better, but it is not until 1,000 samples that we clearly see the familiar bell-shape of the Gaussian probability distribution. This highlights the need to draw many samples, even for a simple random variable, and the benefit of increased accuracy of the approximation with the number of samples drawn.




**Applications of Monte Carlo Simulation:**

It is used by professionals in such widely disparate fields as finance, project management, energy, manufacturing, engineering, research and development, insurance, oil & gas, transportation, and the environment. It was first used by scientists working on the atom bomb; it was named for Monte Carlo, the Monaco resort town renowned for its casinos. Since its introduction in World War II, it has been used to model a variety of physical and conceptual systems. It is used in many areas of industry and science, including:

- Analyzing radiative heat transfer problems
- Estimating the transmission of particles through matter
- Calculating the probability of cost overruns in large projects

- Foreseeing where prices of securities are likely to move
- Analyzing how a network or electric grid will perform in different scenarios
- Assessing risk for credit or insurance
- Simulating proteins in biology

Note: Probability distributions with shape, variables can have different probabilities of different outcomes occurring.