Stats 102C - Lecture 3-1: Random Number Generation

Miles Chen

Week 3 Monday

Section 1

Random Number Generation

Motivation

With Monte Carlo estimation we approximated expectations of functions of random variables.

$$\mathbb{E}_f[h(X)] \approx \bar{h}_n$$

The estimates depend on being able to draw random values from the distribution f.

So far, we have just been using R's ability to generate values from known distributions.

We now turn our discussion to learn how we are able to generate random values from distributions to begin with.

Random Numbers

Nearly all future algorithms for random variable generation depend on the ability to generate random values from a uniform distribution.

Generating random numbers was once a challenging task.

Researchers who needed random numbers would consult random digit tables published by corporations who had access to computer systems that were able to generate random numbers.

RAND corporation recently republished a book of a million random digits:

https://www.amazon.com/Million-Random-Digits-Normal-Deviates/dp/0833030477/

Unimportant musings: Is anything *truly* random?

Is anything **truly** random? Or is everything deterministic?

Is a coin flip or rolling a die truly random? If you know the exact force applied to a coin or die and other physical properties, can you predict how it will land?

Brain activity is based on electrical impulses between neurons, which are simply responding to stimuli. Are our thoughts or decisions our own or are they the natural result of neurons firing impulses according to the rules that have been set out? Do people have free-will? Is anything random or is everything pre-determined?

Quantum mechanics as a theory states that there is true randomness at the quantum level. Einstein famously argued that we simply don't understand what's happening "inside" and so the observations only appear random.

This slide has nothing to do with the rest of the course, but I think it's worth thinking about.

Pseudo Random Number Generators

A psuedo random number generator (PRNG) is a deterministic way to generate a sequence of numbers that appear random.

PRNGs are not random as they will always produce the same sequence of numbers if the algorithm is given the same starting seed.

The topic of pseudo random number generators was an important area of research for several decades. Different techniques and algorithms were developed for generating random numbers.

PRNG: Linear Congruential Generator

The Linear Congruential Generator (LCG) is one of the oldest and well-known methods for generating pseudo random numbers.

The generator works by generating the next random number based on the current random number.

$$X_{n+1} = (aX_n + c) \mod m$$

X_n is the current random number. When $n = 0$, X_n is the starting seed.

X_{n+1} is the next random number.

a, c, m are constants

LCG Example

$$X_{n+1} = (aX_n + c) \mod m$$

Example when

$seed = 1, a = 2, c = 0, m = 9$

- $(2 \cdot 1 + 0) \mod 9 = 2 \mod 9 = 2$
- $(2 \cdot 2 + 0) \mod 9 = 4 \mod 9 = 4$
- $(2 \cdot 4 + 0) \mod 9 = 8 \mod 9 = 8$
- $(2 \cdot 8 + 0) \mod 9 = 16 \mod 9 = 7$
- $(2 \cdot 7 + 0) \mod 9 = 14 \mod 9 = 5$
- $(2 \cdot 5 + 0) \mod 9 = 10 \mod 9 = 1$
- $(2 \cdot 1 + 0) \mod 9 = 2 \mod 9 = 2$
- cycle repeats

Clearly, this is a **VERY BAD** set of values to use for a PRNG.

LCG Code

```r
rand <- function(n, seed, a, c, m) {
  output <- rep(NA, n)
  x <- seed
  for(i in 1:n) {
    x <- (a * x + c) %% m
    output[i] <- x
  }
  output
}
rand(12, seed = 1, a = 2, c = 0, m = 9)
```

```
##  [1] 2 4 8 7 5 1 2 4 8 7 5 1
```

ANSI C

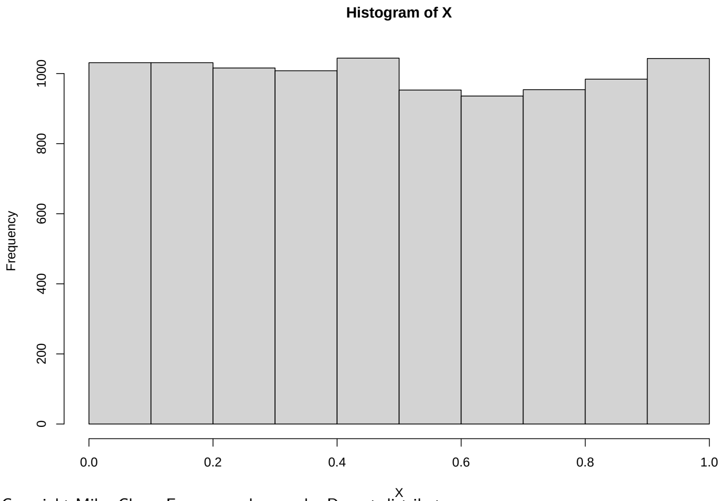
ANSI C is a standard of the C language. It used a Linear Congruential Generator for random number generation with the following values:

- $a = 1103515245$
- $c = 12345$
- $m = 2^{31}$

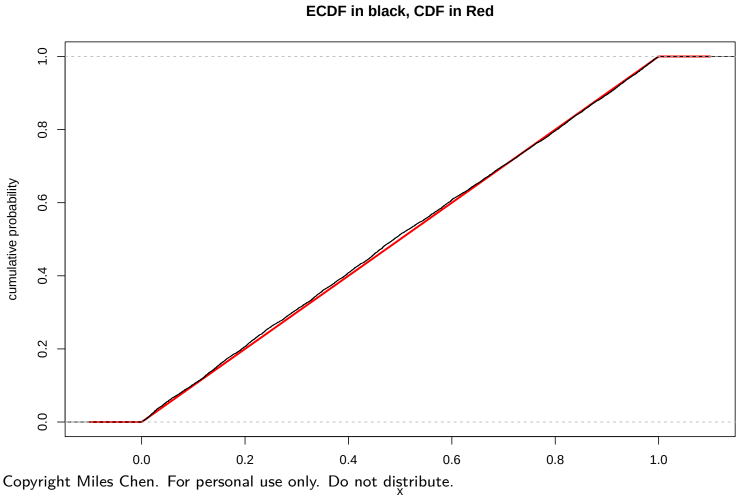
```r
rand(100, seed = 1, a = 1103515245, c = 12345, m = 2^31)
```

```
##   [1] 1103527590  377401600  333417792  314102912  611429056 1995203584
##   [7]   18793472 1909564472  295447552  484895808  600721280 1704829312
##  [13]  877851648 1168774144 1937945600  964613120  395867136  927044672
##  [19] 1805111040  716526336  545163008 1291954944  231735040 2067907392
##  [25] 1207816704 1762564608 1243857408   39176704 1578316344 1992925696
##  [31] 1328949760  920392192 1429199360 1976304128 1551436288 2090168832
##  [37] 1079884288 1385456128  722070016 1417990656 1619630592  595771904
##  [43]   29936128  738865720 1794417152 1602562560  981406208  194435584
##  [49]   27722304  340501880  594555968 1882320640 1444213504 2062919424
##  [55] 1568295680  300687104  217801536  583463552 2073587840  355407360
##  [61] 1640009280  265568512  791534912 1869669376  814302208  355083264
##  [67]  837387328 1882783488 1610007296 1568356096 1647876864 1934896896
##  [73]  455750400 1585571648 1833248256 2080726528  619525632 1131687424
##  [79]  846972416 1493663232  613733888 1888701952  612132352 1410047488
##  [85]  118912512  545549888  930783488   38059264  383428920 1945718016
##  [91] 1687943424 1917269248 1484243200 1842044160  602720512 1729086720
##  [97]  230494464  447638848 1091449984  571181568
```

```r
X <- rand(10000, seed = 1, a = 1103515245, c = 12345, m = 2^31)
X <- X/(2^31) # to scale it between 0 and 1
hist(X, breaks = 10)
```



```r
x <- seq(-.1, 1.1, by = .001)
plot(x, punif(x), type = "l", lwd = 2.5, ylab = "cumulative probability",
     col = "red", main = "ECDF in black, CDF in Red")
plot(ecdf(X), add = TRUE)
```



Mersenne Twister

While the Linear Congruential Generator is simple and easy to understand, its usage has fallen out of favor for more complicated and robust PRNGs.

Today, one of the most popular choices for pseudo random number generation is the Mersenne-Twister (MT). It is the default algorithm choice in R, Python, MATLAB, some versions of C++, and many other languages.

It was developed in 1997 by Makoto Matsumoto and Takuji Nishimura. The name was chosen because the algorithm uses a Mersenne Prime: $2^{19937} - 1$. (A Mersenne Prime is a prime number that is 1 less than a power of 2.)

The algorithm itself is complicated and beyond the scope of this course.

The Mersenne Twister has passed many statistical tests for randomness.

The MT algorithm is not cryptographically secure: based on a series of generated numbers, it is possible to figure out what starting seed was used, and thus what numbers it will generate next. However, for most random number purposes, the MT algorithm is fast, reliable, and more than adequate.

Section 2

Generating Random Variables

Random Uniform Assumption

The rest of the week will focus on generating values from different random distributions.

We will rely on the assumption that we are able to generate values from a random uniform distribution on the interval (0,1): Unif$(0, 1)$.

The PDF is:

$$f(x) = \begin{cases} 1 & \text{for } x \in (0,1) \\ 0 & \text{otherwise} \end{cases}$$

In R, we can use `runif()` to generate random uniform values.

Inverse CDF Method

The Inverse CDF method or inverse transform method uses the inverse of the Cumulative Distribution function to generate values from a probability distribution.

The definition of the CDF of a continuous random variable X is:

$$F(x) := \Pr(X \leq x) = \int_{-\infty}^{x} f(t)dt$$

The CDF of X maps the support of x to the interval [0,1].

More reading: https://en.wikipedia.org/wiki/Inverse_transform_sampling

The Probability Integral Transform

The Probability Integral Transform states that

If continuous random variable X has CDF $F(x)$, then a random variable U defined as

$$U = F(X)$$

has a standard uniform distribution.

The function $F(x)$ transforms X into Unif(0,1). If we start with a value from Unif(0,1), we can use the inverse of $F(x)$ to transform back to X.

More reading: https://en.wikipedia.org/wiki/Probability_integral_transform

Inverse Transform Method (Inverse CDF Method)

X is a continuous random variable with CDF $F(x)$.

The inverse CDF transform is defined as:

$$F^{-1}(u) := \min\left\{t : F(t) \geq u\right\}, \text{for } 0 \leq u \leq 1$$

If $U \sim$ Unif$(0, 1)$, then $F^{-1}(U) \sim F(X)$.

$\min\left\{t : F(t) \geq u\right\}$ means the smallest value of t that satisfies the criteria $F(t) \geq u$.

Proof

To prove that the method works, we'll look at the distribution of the result of putting a random uniform number into the inverse CDF function $F^{-1}(u) = \min\{t : F(t) \geq u\}$

If $U \sim$ Unif$(0, 1)$, then for all $x \in \mathbb{R}$,

$$\begin{aligned}
\Pr[F^{-1}(U) \leq x] &= \Pr[F^{-1}(U) \leq x] \\
&= \Pr[F(F^{-1}(U)) \leq F(x)] \qquad \text{apply F to both sides} \\
&= \Pr[U \leq F(x)] \\
&= F(x) \qquad \text{because } \Pr(U \leq p) = p \text{ when U is uniform}
\end{aligned}$$

Therefore the $F^{-1}(U)$ has the same distribution as X.

Inverse CDF Method Summary

Goal: generate samples $X \sim F(x)$

1. Derive the inverse CDF $F^{-1}(u)$
2. Generate $U \sim \text{Unif}(0,1)$
3. Let $X = F^{-1}(U)$

The inverse CDF transform is defined as:

$$F^{-1}(u) := \min\left\{t : F(t) \geq u\right\}, \text{for } 0 \leq u \leq 1$$

The inverse CDF method works for any distribution as long as F^{-1} can be found.

It can work for both continuous and discrete distributions.

Section 3

Examples of Inverse CDF

Example: Uniform on (a,b)

The PDF for a random uniform variable on the interval (a, b) is:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } x \in (a,b) \\ 0 & \text{otherwise} \end{cases}$$

The CDF is:

$$\begin{aligned} F(x) &= \int_a^x \frac{1}{b-a} dt \\ &= \frac{x}{b-a} - \frac{a}{b-a} \\ &= \frac{x-a}{b-a} \qquad \text{for } a \leq x \leq b \end{aligned}$$

Inverse CDF of Uniform(a, b)

Find the inverse of F

Set $F(x) = u$ and solve for x

$$\begin{aligned} F(x) &= u \\ \frac{x-a}{b-a} &= u \\ x - a &= (b-a)u \\ x &= (b-a)u + a \end{aligned}$$

So $F^{-1}(u) = (b-a)u + a$

Inverse CDF of Uniform(a, b)

Let's say we want to generate random uniform values from Unif(10, 20)

$F^{-1}(u) = (20 - 10)u + 10 = 10u + 10$

So if `runif()` produces:

- `u = 0`, then $x = 10$
- `u = 1`, then $x = 20$
- `u = 0.22`, then $x = 12.2$
- `u = 0.5`, then $x = 15$
- etc.

Example: Exponential distribution

The PDF for a random exponential variable with rate parameter lambda λ is:

$$f(x) = \lambda e^{-\lambda x}, \qquad \text{for } x \geq 0$$

The CDF of Exp(λ) is:

$$\begin{aligned} F(x) &= \int_0^x \lambda e^{-\lambda t} dt \\ &= 1 - e^{-\lambda x}, \qquad \text{for } x \geq 0 \end{aligned}$$

Inverse CDF of Exponential Distribution

Find the inverse of F. Set $F(x) = u$ and solve for x

$$\begin{aligned}
F(x) &= u \\
1 - e^{-\lambda x} &= u \\
-e^{-\lambda x} &= u - 1 \\
e^{-\lambda x} &= 1 - u \\
-\lambda x &= \log(1-u) \\
x &= \frac{-1}{\lambda}\log(1-u)
\end{aligned}$$

Note that $(1-U) \sim Unif(0,1)$ has the same distribution as U.

So $F^{-1}(u) = \frac{-1}{\lambda}\log(u)$

Inverse CDF of Exponential Distribution

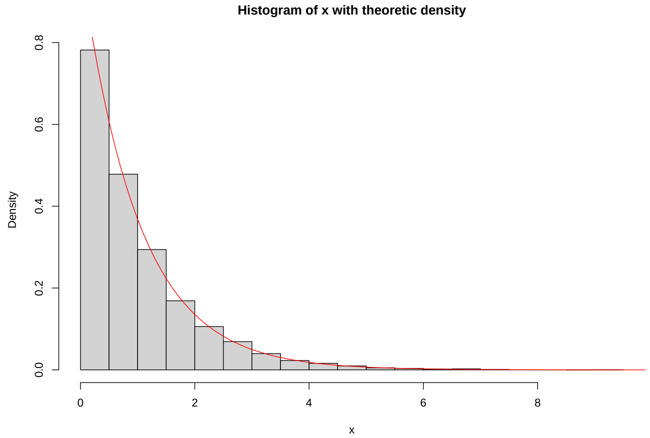
Let's say we want to generate random uniform values from Exp(1)

$F^{-1}(u) = -\log(u)$

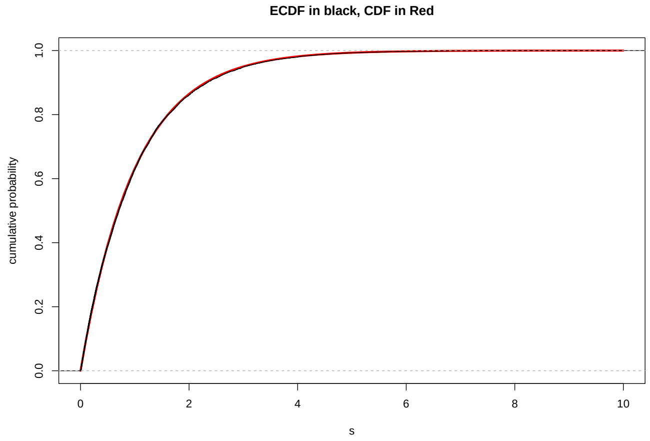
So if `runif()` produces:

- `u = 0`, then $x = \infty$
- `u = 1`, then $x = 0$
- `u = 0.1`, then $x \approx 2.303$
- `u = 0.5`, then $x \approx 0.693$
- `u = 0.9`, then $x \approx 0.105$
- etc.

```r
set.seed(1)
u <- runif(10^4) # random uniform
x <- -log(u) # inverse CDF of exp(rate = 1)
hist(x, freq = FALSE, main = "Histogram of x with theoretic density")
s <- seq(0, 10, by = 0.1); lines(s, dexp(s), col = "red") # add theoretic density
```



```r
plot(s, pexp(s), type = "l", lwd = 3, ylab = "cumulative probability",
     col = "red", main = "ECDF in black, CDF in Red")
plot(ecdf(x), add = TRUE)
```



Kolmogorov-Smirnov test

The Kolmogorov-Smirnov test is a statistical test to see if values could have been generated from a given distribution. It can also be used to test if two samples could have come from the same distribution.

```
# form: ks.test(sample of values, cdf of distribution, additional parameters)
ks.test(x, pexp, rate = 1)
```

```
##
##  Asymptotic one-sample Kolmogorov-Smirnov test
##
## data:  x
## D = 0.00903, p-value = 0.3886
## alternative hypothesis: two-sided
```

The results of this KS test produce a p-value that is greater than 0.05. This means we have no reason to doubt that these values came from an exponential distribution with $\lambda = 1$. (The test does not prove that the values came from this distribution, but the test provides no evidence to the contrary.)

More reading: https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Smirnov_test

The normal distribution

R uses inverse CDF method to generate values from the normal distribution.

While the true CDF and inverse CDF of the normal density have no closed form solutions, the functions `pnorm()` and `qnorm()` provide approximations that are accurate to 16 digits.

When you use `rnorm()` to generate random normal values, R first generates uniform values with `runif()` and converts them into normal values by plugging them into the inverse CDF function `qnorm()`.

```r
set.seed(1); print(runif(5), digits = 12)
```

```
## [1] 0.265508663142 0.372123899637 0.572853363352 0.908207789995 0.201681931037
```

```r
qnorm(0.265508663142) # I plug in the values generated by runif() into qnorm()
```

```
## [1] -0.6264538
```

```r
qnorm(0.572853363352) # qnorm is the inverse CDF function
```

```
## [1] 0.1836433
```

```r
qnorm(0.201681931037) # for some reason, we only use the odd positioned values
```

```
## [1] -0.8356286
```

```r
set.seed(1) # same seed
rnorm(3) # values produced by rnorm match the values from inverse CDF
```

```
## [1] -0.6264538  0.1836433 -0.8356286
```