IS609 Assignment 4 David Stern

David Stern

September 25, 2015

Page 191: #3

In this exercise, we will create a Monte Carlo function that approximates the area under the curve of a quarter-circle, which is centered at $(0, 0)$ and has a radius equal to 1. To do so, our function will take one input, n, and generate two vectors of length n that simulate (x, y) coordinates between $0 \leq x \leq 1$ and $0 \leq y \leq 1$. The function will then calculate each value of $f(x_i)$ where $f(x) = \sqrt{1 - x^2}$. If $y_1 \leq f(x)_i$, we will add the value to a counter and then divide that counter by the total number of random points, which should approximate the area of the quarter circle given a relatively large number n.

```
monte <- function(n){
  counter <- 0
  x <- runif(n,0,1)
  y <- runif(n,0,1)
  fx <- sqrt(1-x^2)
  for (i in 1:n){if (y[i] <= fx[i]){counter <- counter + 1}}
  area <- counter/n
  return(area)
                    }
monte(10)
```

```
## [1] 0.7
```

```
monte(100)
```

```
## [1] 0.83
```

```
monte(1000)
```

```
## [1] 0.805
```

```
monte(10000)
```

```
## [1] 0.787
```

We can compare this to the value of the area of a quarter of a circle with radius = 1, which would be A = $\frac{\pi}{4}$.

```
pi/4
```

```
## [1] 0.7853982
```

Page 194: #1

For this exercise, I am creating two separate middle-square algorithms to deal generate random numbers given a four or six digit seed. The algorithms are both designed with multiple conditionals to slice the middle 4 or 6 digits out of a square given the exact number of digits in a square. Since $9999^2 = 99980001$, we know that the highest number of digits in the square of a 4-digit seed is 8. We can then use the following algorithm. Likewise, the highest number of digits in the square of a 6-digit seed in 12.

```
seed <- 1009
midSquareFour <- function(seed,n){
  squares <- c(seed)
  for (i in 1:n){
    temp <- squares[i]^2
    if(nchar(temp)==8){
      squares[i+1] <- as.numeric(substr(temp,3,6))
      }
    else if (nchar(temp)==7){
      squares[i+1] <- as.numeric(substr(temp,2,5))
      }
    else if (nchar(temp)==6){
      squares[i+1] <- as.numeric(substr(temp,1,4))
      }
    else if (nchar(temp)==5){
      squares[i+1] <- as.numeric(substr(temp,1,3))
      }
    else if (nchar(temp)==4){
      squares[i+1] <- as.numeric(substr(temp,1,2))
      }
    else if (nchar(temp)==3){
      squares[i+1] <- as.numeric(substr(temp,1,1))
      }
    else
      squares[i+1] <- 0
    }
  return(squares)
   }

midSquareSix <- function(seed,n){
  squares <- c(seed)
  for (i in 1:n){
    temp <- squares[i]^2
    if(nchar(temp)==12){
      squares[i+1] <- as.numeric(substr(temp,4,9))
      }
    else if (nchar(temp)==11){
      squares[i+1] <- as.numeric(substr(temp,3,8))
      }
    else if (nchar(temp)==10){
      squares[i+1] <- as.numeric(substr(temp,2,7))
      }
    else if (nchar(temp)==9){
      squares[i+1] <- as.numeric(substr(temp,1,6))
      }
    else if (nchar(temp)==8){
```

```
      squares[i+1] <- as.numeric(substr(temp,1,5))
      }
    else if (nchar(temp)==7){
      squares[i+1] <- as.numeric(substr(temp,1,4))
      }
    else if (nchar(temp)==6){
      squares[i+1] <- as.numeric(substr(temp,1,3))
      }
    else if (nchar(temp)==5){
      squares[i+1] <- as.numeric(substr(temp,1,2))
      }
    else if (nchar(temp)==4){
      squares[i+1] <- as.numeric(substr(temp,1,1))
      }
    else
      squares[i+1] <- 0
    }
  return(squares)
   }
```

Use the middle-square method to generate:

a. 10 random numbers using $x_0 = 1009$.

```
midSquareFour(1009,10)
```

```
##  [1] 1009  180  324 1049 1004   80   64   40   16    2    0
```

b. 20 random numbers using $x_0 = 653217$.

```
midSquareSix(653217,20)
```

```
##  [1] 653217 692449 485617 823870 761776 302674 611550 993402 847533 312186
## [11] 460098 690169 333248  54229 940784  74534 555317 376970 106380 316704
## [21] 301423
```

c. 15 random numbers using $x_0 = 3043$.

```
midSquareFour(3043,15)
```

```
##  [1] 3043 2598 7496 1900 6100 2100 4100 8100 6100 2100 4100 8100 6100 2100
## [15] 4100 8100
```

d. Comment about the results of each sequence. Was there cycling? Did each sequence degenerate rapidly?

For $x_0 = 1009$, we see that the sequence degenerates the most rapidly of the three seed values. We also the sequence begin to cycle after the 10th random number.

```
midSquareFour(1009,20)
```

```
##  [1] 1009  180  324 1049 1004   80   64   40   16    2    0    0    0    0
## [15]    0    0    0    0    0    0    0
```

For $x_0 = 653217$ we do not see much degeneration or sequencing, even for a high value such as n = 500. The expression below will show that we still have 6-digit values at $n = 500$.

```
nchar(midSquareSix(653217,500))
```

```
##   [1] 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 4 5 5 6
##  [36] 6 6 6 6 6 6 6 6 5 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
##  [71] 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 6 5 6 6 5 6 6 6 6 6 6 6 4 5 6 6 6
## [106] 6 6 5 6 6 6 6 5 6 5 6 6 6 6 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6
## [141] 6 6 5 5 6 6 6 6 6 6 6 5 6 6 6 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
## [176] 6 6 6 5 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6
## [211] 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 6 6 6 6
## [246] 6 4 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 6 5 6 6 6 6 6 6 6
## [281] 6 6 5 6 6 6 6 5 6 6 6 6 6 6 5 6 6 5 6 5 6 6 6 6 6 6 5 6 6 6 6 6 6 5 6
## [316] 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 6 6 5 6 6 6 6 6 6 6 6
## [351] 6 6 6 6 6 6 6 6 5 6 6 6 6 6 6 6 6 6 6 6 6 5 6 6 6 6 5 6 6 6 6 6 6 6 6
## [386] 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
## [421] 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
## [456] 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
## [491] 6 6 6 6 6 6 6 6 6 6
```

For $x_0 = 3043$, the sequence of random numbers cycle very early, starting at $n = 6$.

Page 201: #4

Here we will use a Monte Carlo simulation of one thousand horse races given the following odds:

Entry's name Odds:

horse	odds	probability
Euler's Folly	7–1	0.1250000
Leapin' Leibniz	5–1	0.1666667
Newton Lobell	9-1	0.1000000
Count Cauchy	12-1	0.0769231
Pumped up Poisson	4-1	0.2000000
Loping L'Hopital	35-1	0.0277778
Steamin' Stokes	15-1	0.0625000
Dancin Dantzig	4-1	0.2000000

First we have to convert the odds to probabilities and then create a function that samples from the discrete probability distribution.

```
probability <- c(1/8,1/6,1/10,1/13,1/5,1/36,1/16,1/5)
races <- function(n) {
  sample(x = c("Euller's Folly","Leapin' Leibniz","Newton Lobell","Count Cauchy","Pumped up Poisson","L
```

```
  }
set.seed(45341)
table(races(1000))
```

```
##
##      Count Cauchy   Dancin' Dantzig    Euller's Folly   Leapin' Leibniz
##                94               208               127               168
##  Loping L'Hopital     Newton Lobell Pumped up Poisson   Steamin' Stokes
##                27               110               204                62
```

The results show us that Loping L'Hopital won the fewest races and Dancin' Dantzig won the most. We should expect both of these results from the odds. We can see this better if we divide the race results by 1000.

```
table(races(1000))/1000
```

```
##
##      Count Cauchy   Dancin' Dantzig    Euller's Folly   Leapin' Leibniz
##             0.077             0.201             0.142             0.192
##  Loping L'Hopital     Newton Lobell Pumped up Poisson   Steamin' Stokes
##             0.028             0.106             0.199             0.055
```

Some of the results are very close to the probabilities, but we see a lot of variation for a few horses. This may be because the probabilities do not add up to 1.

```
sum(probability)
```

```
## [1] 0.9588675
```

This is typical for racetracks, as they sum the probabilities of all of the horses to more than 1 in order for them to earn a take. We should expect there to be some inaccuracy in our results.

Page 211: #3

In many situations, the time T between deliveries and the order quantity Q is not fixed. Instead, an order is placed for a specific amount of gasoline. Depending on how many orders are placed in a given time interval, the time to fill an order varies. You have no reason to believe that the performance of the delivery operation will change. Therefore, you have examined records for the past 100 deliveries and found the following lag times, or extra days, required to fill your order:

```
lagTime <- 2:7
occurrences <- c(10,25,30,20,13,2)
relativeFrequency <- occurrences/sum(occurrences)
gasDelivery <- data.frame(lagTime,occurrences,relativeFrequency)
colnames(gasDelivery) <- c("Lag Time","Occurrences","Relative Frequency")
kable(gasDelivery)
```

Lag Time	Occurrences	Relative Frequency
2	10	0.10
3	25	0.25
4	30	0.30
5	20	0.20
6	13	0.13
7	2	0.02

Construct a Monte Carlo simulation for the lag time submodel. If you have a handheld calculator or computer available, test your submodel by running 1000 trials and comparing the number of occurrences of the various lag times with the historical data.

```
deliveries <- function(n){
  sample(x=lagTime,n,replace=T,prob=relativeFrequency)
}
results <- table(deliveries(1000))
results/1000
```

```
##
##     2     3     4     5     6     7
## 0.079 0.268 0.299 0.210 0.125 0.019
```

```
relativeFrequency
```

```
## [1] 0.10 0.25 0.30 0.20 0.13 0.02
```

Our results are very close to the probabilities derived from the historical data.