Homework 2.2

CUNY MSDS DATA 609

Duubar Villalobos Jimenez mydvtech@gmail.com

September 23, 2018

Problems

The below problems are taken from the text book:

A First Course in Mathematical Modeling, 5th Edition. Frank R. Giordano, William P. Fox, Steven B. Horton. ISBN-13: 9781285050904.

Exercise #3 Page 191.

Using Monte Carlo simulation, write an algorithm to calculate an approximation to π by considering the number of random points selected inside the quarter circle

$$Q : x^2 + y^2 = 1,\ x \geq 0,\ y \geq 0$$

where the quarter circle is taken to be inside the square

$$S : 0 \leq x \leq 1\, and\, 0 \leq y \leq 1$$

Use the equation $\pi/4 = area\, Q/area\, S$

Solution:

The below is the single simulation program for a given number of n values.

```
# FUNCTION THAT APPROXIMATES PI
approximate_pi <- function(n){

  # Defining original values
  M <- 1
  a <- 0
  b <- 1

  # Generating random values
  x <- runif(n,0,1)
  y <- runif(n,0,1)

  # Evaluating the function to compare
  y_eval <- (1 - x^2)^(1/2)

  xy.df <- data.frame(y_eval,y,x)

  # Assigning values if condition is met, 1 = Yes, 0 = No
  xy.df$`Under Curve` <- 0
```

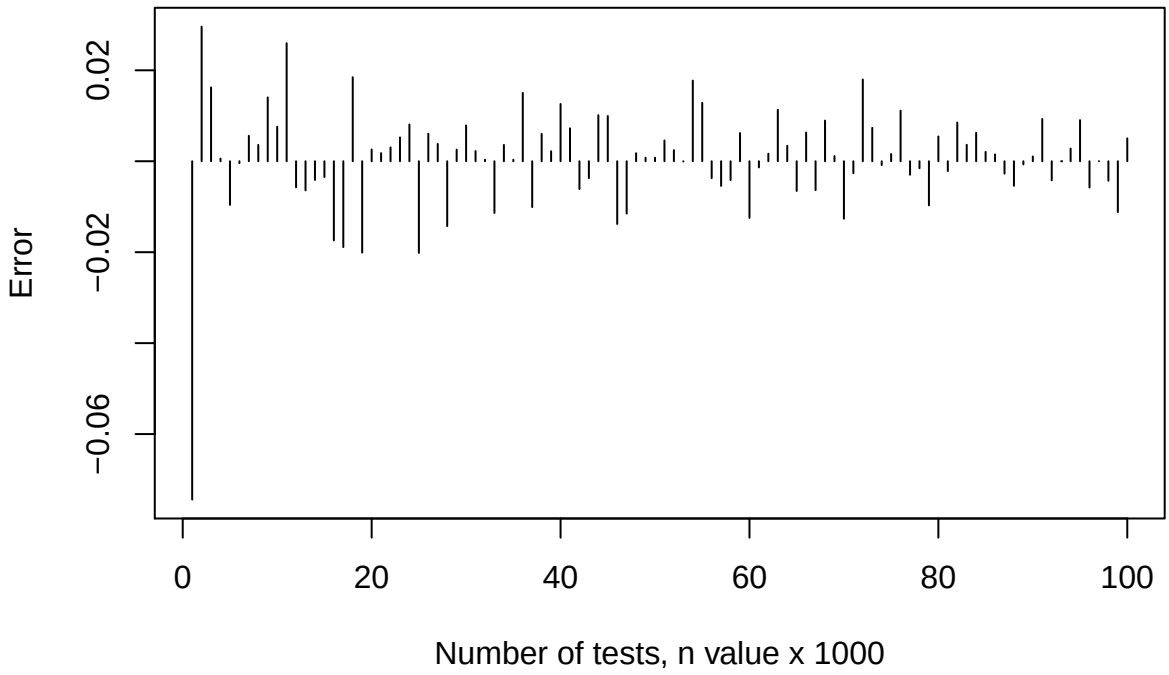
```
  xy.df$`Under Curve`[which(xy.df$y <= xy.df$y_eval)] <- 1

  COUNTER <- sum(xy.df$`Under Curve`)
  AreaS <- M * (b - a)
  AreaQ <- AreaS * COUNTER / n
  approxPi <- 4 * AreaQ

  return(approxPi)

}
```

In order to create a visualize a pattern, I will call the individual simulation 100 times for different incremental values.

```
# PROCESS TO RUN MULTIPLE SIMULATIONS
n <- c()
spi <- c()

simulation <- data.frame(n = n, 'Simul_pi' = spi)

# Let's run 100 simulations for different n values
for (i in 1:100){

  # The multiplier is to create increments of 1000 for each n
  nmultiplier <- 1000 * i
  temp <- data.frame(n = nmultiplier, 'Simul_pi' = approximate_pi(nmultiplier))

  simulation <- rbind(simulation, temp)

}

simulation$Real_pi <- pi
simulation$Error <- simulation$Real_pi - simulation$Simul_pi
```

Let's have a visual representation of the Error pattern for the comparison.

```
plot(x = simulation$n / 1000,
     y = simulation$Error,
     type = "h",
     xlab = 'Number of tests, n value x 1000',
     ylab = 'Error')
```



Something interesting to note is that it seems that the bigger the n, the lower the Error. In other words this method tends to approximate the real value of π.

Let's see a single simulation with n = 100 M.

```
approximate_pi(100000000)
```

```
## [1] 3.141696
```

Lets' see a table of values generated.

```
##          n Simul_pi  Real_pi         Error
## 1     1000 3.216000 3.141593 -7.440735e-02
## 2     2000 3.112000 3.141593  2.959265e-02
## 3     3000 3.125333 3.141593  1.625932e-02
## 4     4000 3.141000 3.141593  5.926536e-04
## 5     5000 3.151200 3.141593 -9.607346e-03
## 6     6000 3.142000 3.141593 -4.073464e-04
## 7     7000 3.136000 3.141593  5.592654e-03
## 8     8000 3.138000 3.141593  3.592654e-03
## 9     9000 3.127556 3.141593  1.403710e-02
## 10   10000 3.134000 3.141593  7.592654e-03
## 11   11000 3.115636 3.141593  2.595629e-02
## 12   12000 3.147333 3.141593 -5.740680e-03
## 13   13000 3.148000 3.141593 -6.407346e-03
## 14   14000 3.145714 3.141593 -4.121632e-03
## 15   15000 3.145067 3.141593 -3.474013e-03
## 16   16000 3.159000 3.141593 -1.740735e-02
## 17   17000 3.160471 3.141593 -1.887793e-02
## 18   18000 3.123111 3.141593  1.848154e-02
## 19   19000 3.161684 3.141593 -2.009156e-02
## 20   20000 3.139000 3.141593  2.592654e-03
## 21   21000 3.139810 3.141593  1.783130e-03
## 22   22000 3.138545 3.141593  3.047199e-03
```

```
## 23   23000 3.136348 3.141593  5.244828e-03
## 24   24000 3.133500 3.141593  8.092654e-03
## 25   25000 3.161760 3.141593 -2.016735e-02
## 26   26000 3.135538 3.141593  6.054192e-03
## 27   27000 3.137778 3.141593  3.814876e-03
## 28   28000 3.155857 3.141593 -1.426449e-02
## 29   29000 3.139034 3.141593  2.558171e-03
## 30   30000 3.133733 3.141593  7.859320e-03
## 31   31000 3.139355 3.141593  2.237815e-03
## 32   32000 3.141250 3.141593  3.426536e-04
## 33   33000 3.152970 3.141593 -1.137704e-02
## 34   34000 3.138000 3.141593  3.592654e-03
## 35   35000 3.141257 3.141593  3.355107e-04
## 36   36000 3.126556 3.141593  1.503710e-02
## 37   37000 3.151676 3.141593 -1.008302e-02
## 38   38000 3.135579 3.141593  6.013706e-03
## 39   39000 3.139385 3.141593  2.208038e-03
## 40   40000 3.129000 3.141593  1.259265e-02
## 41   41000 3.134341 3.141593  7.251190e-03
## 42   42000 3.147714 3.141593 -6.121632e-03
## 43   43000 3.145302 3.141593 -3.709672e-03
## 44   44000 3.131455 3.141593  1.013811e-02
## 45   45000 3.131644 3.141593  9.948209e-03
## 46   46000 3.155391 3.141593 -1.379865e-02
## 47   47000 3.153106 3.141593 -1.151373e-02
## 48   48000 3.139833 3.141593  1.759320e-03
## 49   49000 3.140816 3.141593  7.763271e-04
## 50   50000 3.140800 3.141593  7.926536e-04
## 51   51000 3.137020 3.141593  4.573046e-03
## 52   52000 3.139154 3.141593  2.438807e-03
## 53   53000 3.141660 3.141593 -6.772377e-05
## 54   54000 3.123852 3.141593  1.774080e-02
## 55   55000 3.128727 3.141593  1.286538e-02
## 56   56000 3.145286 3.141593 -3.693061e-03
## 57   57000 3.147018 3.141593 -5.424890e-03
## 58   58000 3.145724 3.141593 -4.131484e-03
## 59   59000 3.135390 3.141593  6.202823e-03
## 60   60000 3.154067 3.141593 -1.247401e-02
## 61   61000 3.142951 3.141593 -1.358166e-03
## 62   62000 3.139935 3.141593  1.657170e-03
## 63   63000 3.130286 3.141593  1.130694e-02
## 64   64000 3.138187 3.141593  3.405154e-03
## 65   65000 3.148123 3.141593 -6.530423e-03
## 66   66000 3.135273 3.141593  6.319926e-03
## 67   67000 3.147940 3.141593 -6.347645e-03
## 68   68000 3.132647 3.141593  8.945595e-03
## 69   69000 3.140464 3.141593  1.128885e-03
## 70   70000 3.154229 3.141593 -1.263592e-02
## 71   71000 3.144225 3.141593 -2.632699e-03
## 72   72000 3.123611 3.141593  1.798154e-02
## 73   73000 3.134247 3.141593  7.346078e-03
## 74   74000 3.142432 3.141593 -8.397788e-04
## 75   75000 3.140000 3.141593  1.592654e-03
## 76   76000 3.130474 3.141593  1.111897e-02
```

```
## 77   77000 3.144571 3.141593 -2.978775e-03
## 78   78000 3.143128 3.141593 -1.535552e-03
## 79   79000 3.151291 3.141593 -9.698486e-03
## 80   80000 3.136150 3.141593  5.442654e-03
## 81   81000 3.143753 3.141593 -2.160433e-03
## 82   82000 3.133073 3.141593  8.519483e-03
## 83   83000 3.137976 3.141593  3.616750e-03
## 84   84000 3.135333 3.141593  6.259320e-03
## 85   85000 3.139529 3.141593  2.063242e-03
## 86   86000 3.140093 3.141593  1.499630e-03
## 87   87000 3.144322 3.141593 -2.729185e-03
## 88   88000 3.147000 3.141593 -5.407346e-03
## 89   89000 3.142292 3.141593 -6.994812e-04
## 90   90000 3.140533 3.141593  1.059320e-03
## 91   91000 3.132308 3.141593  9.284961e-03
## 92   92000 3.145783 3.141593 -4.189955e-03
## 93   93000 3.141591 3.141593  1.255740e-06
## 94   94000 3.138809 3.141593  2.784143e-03
## 95   95000 3.132547 3.141593  9.045285e-03
## 96   96000 3.147375 3.141593 -5.782346e-03
## 97   97000 3.141567 3.141593  2.564328e-05
## 98   98000 3.145878 3.141593 -4.284897e-03
## 99   99000 3.152768 3.141593 -1.117502e-02
## 100 100000 3.136560 3.141593  5.032654e-03
```

Exercise #1 Page 194.

```
# FUNCTION THAT EXTRACT THE 'RANDOM' VALUE FROM A SEED
generate_random_middle_square <- function(x0){

  # Need to figure the lenght for leading zero if needed.
  nlength <- 8
  nstart <- 3
  nend <- 6

  if (x0 > 9999){
    nlength <- 12
    nstart <- 4
    nend <- 9
  }

  # Find square value
  x2 <- x0^2

  # Need to add leading zero if needed.
  x2_lenght <- nchar(x2)
  while (x2_lenght < nlength){

    x2 <- toString(x2)
    x2 <- paste(0, x2, sep='')
    x2_lenght <- nchar(x2)

  }
```

```
  # Extract middle number
  x0 <- as.numeric(substr(x2, nstart, nend))

  return(x0)

}
```

```
# FUNCTION THAT CREATE A TABLE LIST OF "RANDOM" VALUES FROM A SEED
random_middle_square_list <- function(x0, n){

  xn.table <- data.frame('n' = 0, 'xn' = x0)

  for (i in 1:n){

    x0 <- generate_random_middle_square(x0)
    temp.df <- data.frame('n' = i, 'xn' = x0)

    xn.table <- rbind(xn.table, temp.df)

  }

  xn.table

}
```

Use the middle-square method to generate

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

```
random_middle_square_list(x0 = 1009, n = 10)
```

```
##     n   xn
## 1   0 1009
## 2   1  180
## 3   2  324
## 4   3 1049
## 5   4 1004
## 6   5   80
## 7   6   64
## 8   7   40
## 9   8   16
## 10  9    2
## 11 10    0
```

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

```
random_middle_square_list(x0 = 653217, n = 20)
```

```
##     n     xn
## 1   0 653217
## 2   1 692449
## 3   2 485617
```

```
## 4   3 823870
## 5   4 761776
## 6   5 302674
## 7   6 611550
## 8   7 993402
## 9   8 847533
## 10  9 312186
## 11 10 460098
## 12 11 690169
## 13 12 333248
## 14 13  54229
## 15 14 940784
## 16 15  74534
## 17 16 555317
## 18 17 376970
## 19 18 106380
## 20 19 316704
## 21 20 301423
```

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

```
random_middle_square_list(x0 = 3043, n = 15)
```

```
##     n   xn
## 1   0 3043
## 2   1 2598
## 3   2 7496
## 4   3 1900
## 5   4 6100
## 6   5 2100
## 7   6 4100
## 8   7 8100
## 9   8 6100
## 10  9 2100
## 11 10 4100
## 12 11 8100
## 13 12 6100
## 14 13 2100
## 15 14 4100
## 16 15 8100
```

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

In the first sequence with $x_0 = 1009$ the sequence did not present cycling, however the sequence degenerate quickly.

In the second sequence, there seems to be some "randomness" since it doesn't seems to degenerate nor does it seems to cycle.

In the third sequence, the sequence did not degenerate quickly, however there's cycling involved since we see a repetitive pattern after a few iterations.

END.