EAAD - Lab 1

Greg Ceccarelli

June 2, 2015

Question 1

Write a function in R that takes a positive integer and returns all the prime numbers up to it. Use the Sieve of Eratosthenes (Links to an external site.) for doing this. Following are a few helpful notes.

- The base of the algorithm is that we start from 2 and for every prime number, we cross out all of its multiples up n. In the end, numbers that are not crossed out, are prime.
- We only need to cross out multiples of numbers which are not yet crossed out.
- We only need to cross out multiples of prime numbers that are smaller than the square root of n.
- The basic structure of your function is like this:

```
prime.sieve = function(n) {
...
return(primes)
}
```

```r
prime.sieve <- function(number, first_prime=2) {
##hold a vector of numbers to check in a variable
##start at 2 given the above note using the function's default argument
  to_check <- c(first_prime:number)

##initatilize primes variable vector to store results into
  primes <- NULL
  count <- 0

##initatilize loop variable
  loop <- first_prime
  while (loop*loop < number) {
    primes <- c(primes, to_check[1])
##remove multiples from to_check vector
    to_check <- subset(to_check, !to_check %% loop == 0)

## update loop variable with next prime after multiples are removed
    loop <- to_check[1]
    count <- count+1
}
##given the loop stops "prematurely", concatenate remaining primes from to_check vector with primes sto
primes <- c(primes,to_check)

##can comment out these two print lines, for info only
print(paste("Loop executed ", count, " times"))
print(paste("Your number,", number, ", contains ", length(primes), " primes up to it and they are:"))
return(primes)
}

prime.sieve(1000)
```

```
## [1] "Loop executed  11  times"
## [1] "Your number, 1000 , contains  168  primes up to it and they are:"

##   [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59
##  [18]  61  67  71  73  79  83  89  97 101 103 107 109 113 127 131 137 139
##  [35] 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233
##  [52] 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337
##  [69] 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439
##  [86] 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557
## [103] 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653
## [120] 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769
## [137] 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883
## [154] 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997
```

Question 2

We want to use Monte Carlo simulation to calculate pi. Here is how:

We generate a large number (call this number n) of randomly distributed x and y coordinates in the range of [-1, +1], and use the formula for a circle with radius=1 and centered on the origin, i.e., x^2 + y^2 = 1, to figure out which of these points are inside and which are outside the circle. Then, we just calculate what fraction of the points are inside and multiply this by the are of the square containing all the coordinates (2*2= 4). The result should be an estimation of pi.

- use runif(n, min = -1, max = 1) to generate uniformly distributed random numbers that range from -1 to +1.
- Bonus question: What is the variance of your estimation of pi (calculate this theoretically)? To confirm this, do the above procedure N times, and calculate the variance numerically.

```
pi.compute <- function(random_numbers) {
##initialize x and y vectors based on random_numbers input
  n <-random_numbers
##randomly select n number of uniformly distributed points for x and y based on function input
  x <- runif(n, min = -1, max = 1)
  y <- runif(n, min = -1, max = 1)
##create a vector to test based on unit circle equation
  eq <- x^2 + y^2

#count the points that are less than or equal to 1
  m <- length(subset(eq, eq<=1))

##ratio of the area of the circle to the area of the square
  p <- m/n
  pi.estimate <- p*4
  return(pi.estimate)
}

##define a function to compute estimation variance
pi.runner <- function(runs=100,random=1000){
  pi.hold = NULL
  for (i in 1:runs) {

##append estimated value of pi in a vector based on number of runs
```

```r
    pi.hold <- append(pi.hold, pi.compute(random))
  }
  return(pi.hold)
}

#compute various estimation variances based on changing number of runs and random numbers selected
#default
var(pi.runner())
```

```
## [1] 0.002608115
```

```r
#misc changes to input parameters
var(pi.runner(500))
```

```
## [1] 0.002639066
```

```r
var(pi.runner(1000))
```

```
## [1] 0.002787338
```

```r
var(pi.runner(500,10000))
```

```
## [1] 0.0002752184
```