5206_hw5

Shuyuan Wang, sw3449

11/22/2019

```r
set.seed(0)
```

Part 1: Inverse Transform Method

Problem 1

1. Let U be a uniform random variable over [0,1]. Find a transformation of U that allows you to simulate X from U.

$\int_{\infty}^{x} \frac{1}{\pi}\frac{1}{(1+t^2)}dt = \frac{1}{\pi}(arctan(x) + \frac{\pi}{2}) = \frac{1}{\pi}arctan(x) + \frac{1}{2}$

$\frac{1}{\pi}arctan(x) + \frac{1}{2} = U \Rightarrow X = tan((U - \frac{1}{2})\pi)$

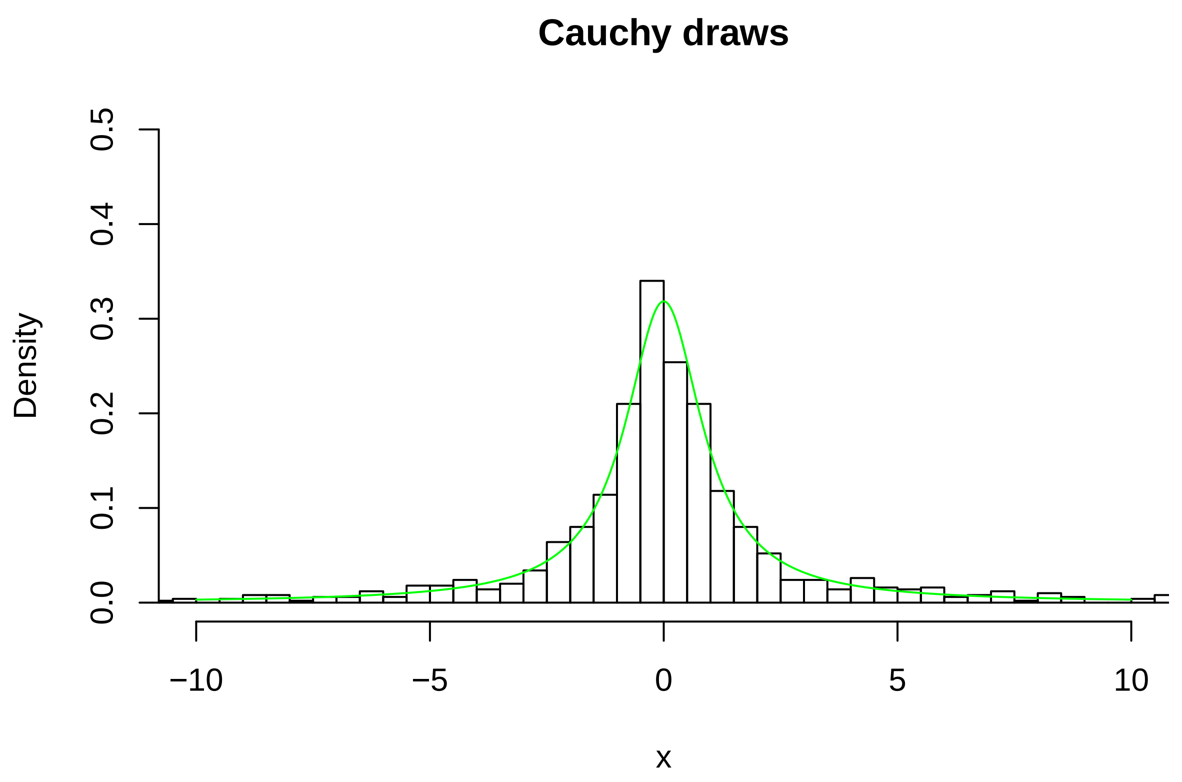
2. Write a `R` function called cauchy.sim that generates `n` simulated Cauchy random variables. The function should have the single input `n` and should use the inverse-transformation from Part 1. Test your function using 10 draws.

```r
cauchy.sim <- function(n){
  U <- runif(n)
  return(tan((U-0.5)*pi))
}
cauchy.sim(10)
```

```
##  [1]  2.9723830 -0.9070131 -0.4248394  0.2329576  3.3710605 -1.3611982
##  [7]  3.0255173  5.6954298  0.5530230  0.4294381
```

3. Using your function `cauchy.sim`, simulate 1000 random draws from a Cauchy distribution. Store the 1000 draws in the vector `cauchy.draws`. Construct a histogram of the simulated Cauchy random variable with $f_X(x)$ overlaid on the graph. **Note**: when plotting the density curve over the histogram, include the argument `prob = T`. **Also note**: the Cauchy distribution produces extreme outliers. I recommend plotting the histogram over the interval (-10,10).

```r
cauchy.draws <- cauchy.sim(1000)
hist(cauchy.draws,prob=T,xlim=c(-10,10),ylim=c(0,0.5),breaks=10000,main='Cauchy draws',xlab='x')
x <- seq(-10,10,0.01)
lines(x,1/(pi*(1+x^2)),col='green')
```

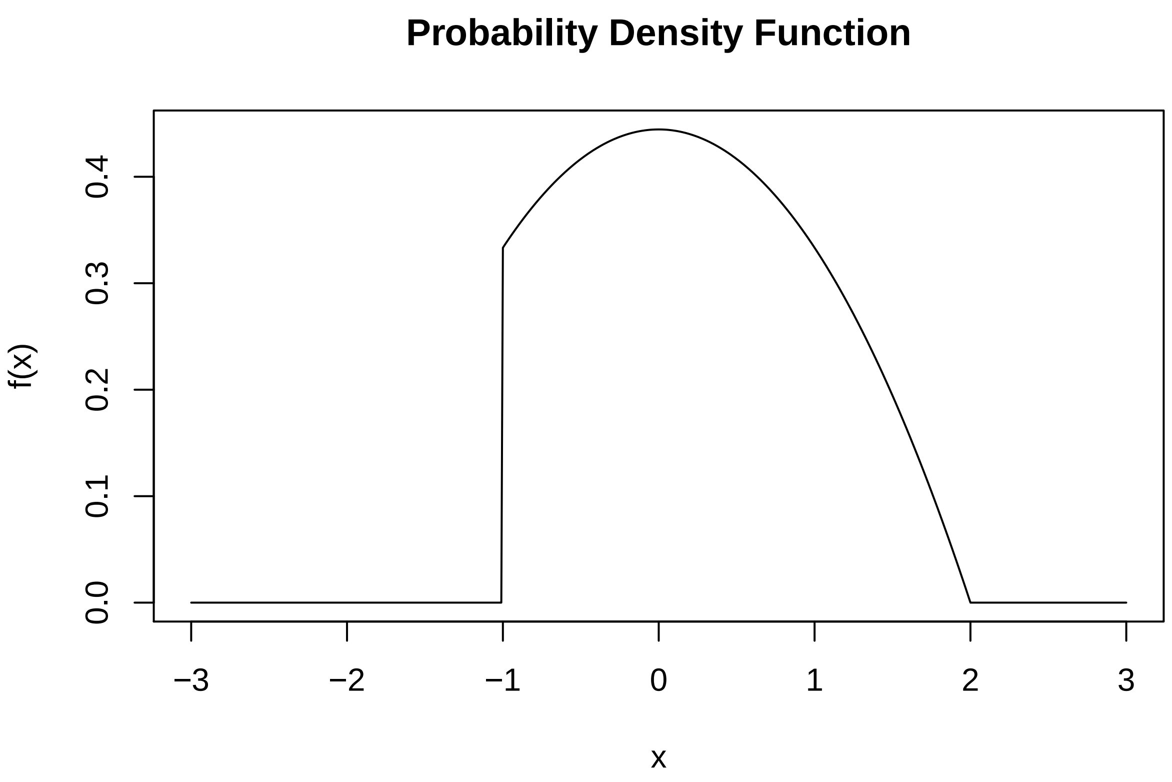


Part 2: Reject-Accept Method

Problem 2

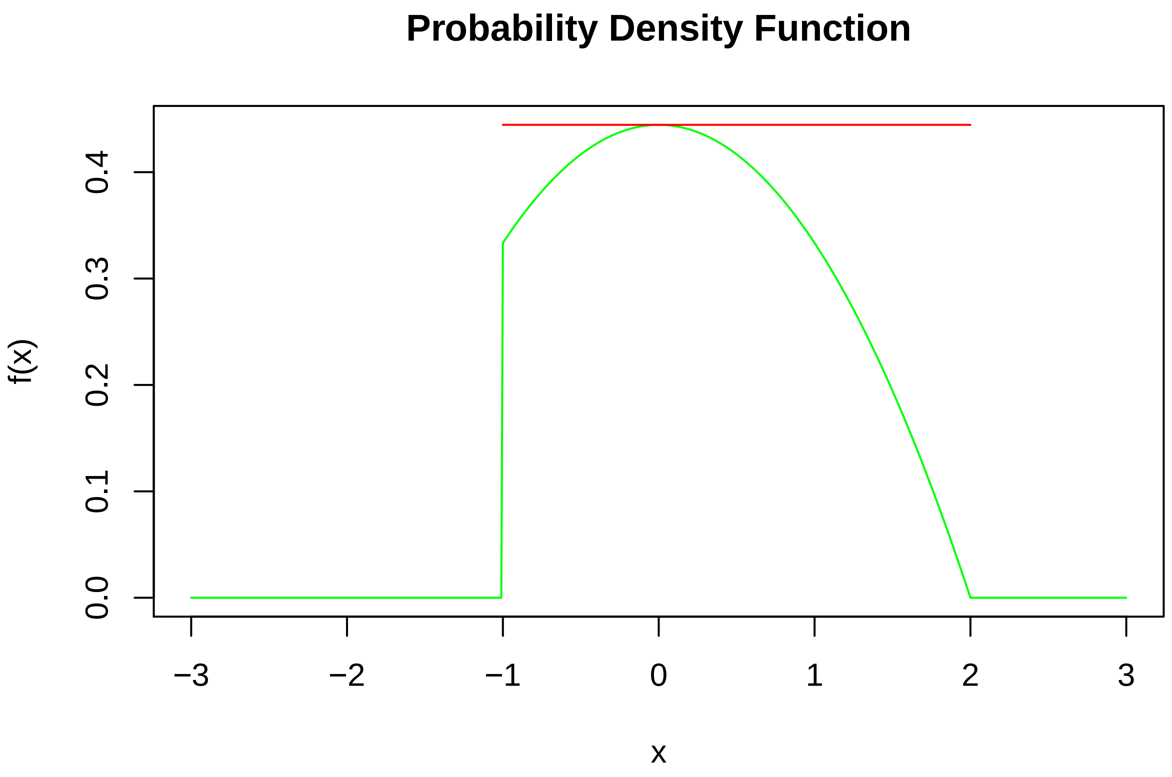
4. Write a function `f` that takes as input a vector `x` and returns a vector of `f(x)` values. Plot the function between -3 and 3.

```
f <- function(x){
  return(ifelse((x<(-1)|x>2),0,(4-x^2)/9))
}
x <- seq(-3,3,0.01)
plot(x,f(x),type='l',main='Probability Density Function')
```



5. Determine the maximum of $f(x)$ and find an envelope function $e(x)$ by using a uniform density for $g(x)$. Write a function `e` which takes as input a vector `x` and returns a vector of `e(x)` values.

```r
# f(x) gets its maximum at x=0
f.max <- f(0)
e <- function(x) {
  ifelse((x<(-1) | x>2), Inf, f.max)
}

plot(x,f(x),main="Probability Density Function",type="l",col="green")
lines(x,e(x),col="red")
```

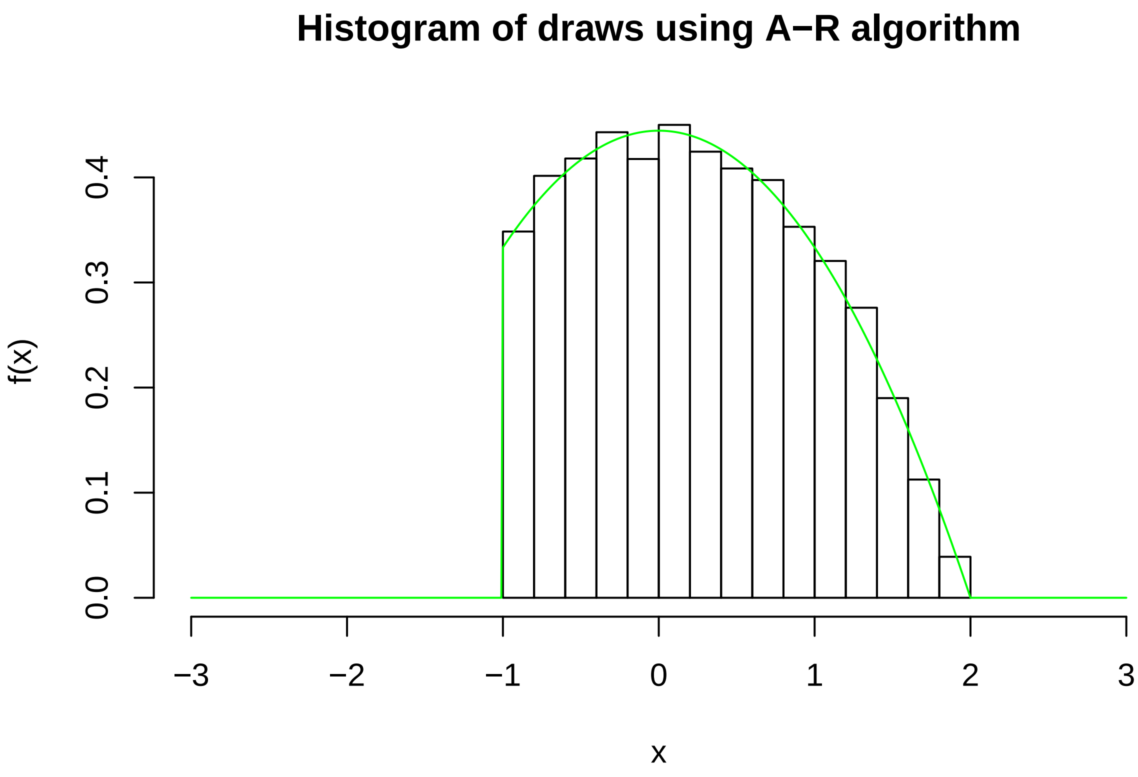


6. Using the **Accept-Reject Algorithm**, write a program that simulates 10,000 draws from the probability density function $f(x)$ from Equation 1. Store your draws in the vector `f.draws`.

```
n.samps <- 10000 # number of samples desired
n <- 0 #counter
f.draws <- numeric(n.samps)
while (n<n.samps){
  y <- runif(1,-1,2) # random draw from g
  u <- runif(1)
  if (u<f(y)/e(y)){
    n <- n+1
    f.draws[n] <- y
  }
}
```

7. Plot a histogram of your simulated data with the density function `f` overlaid in the graph. Label your plot appropriately.

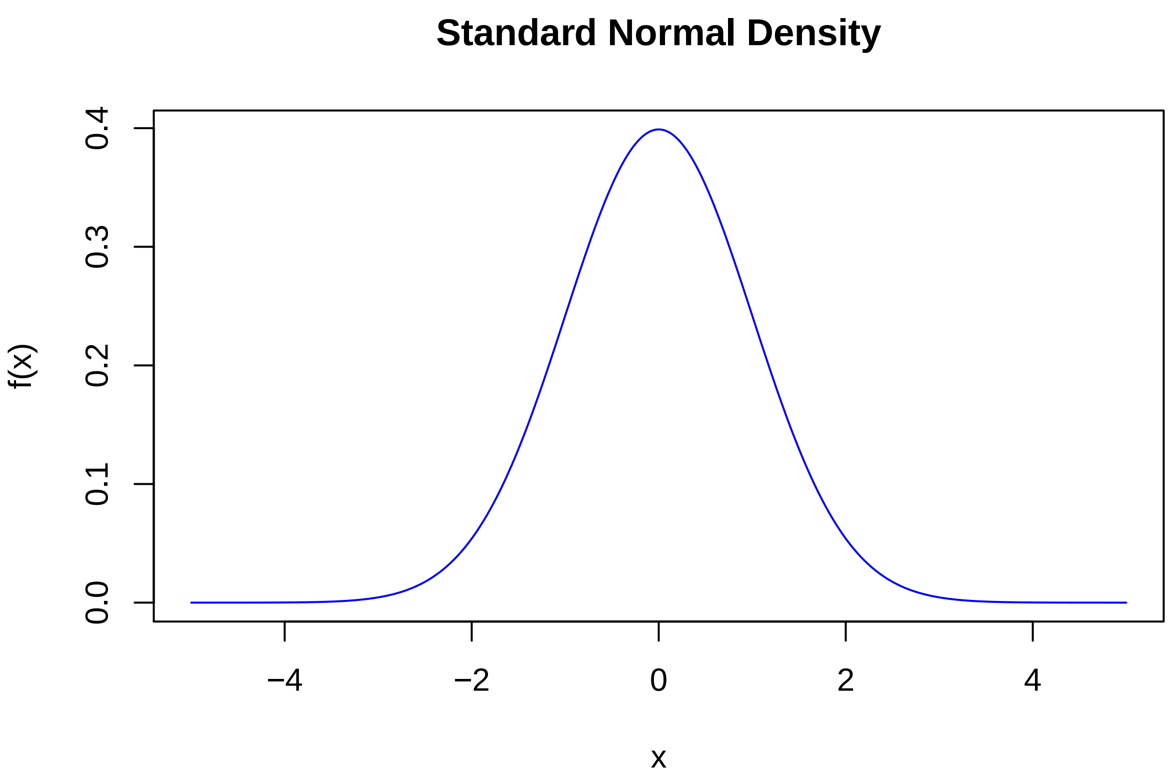
```
hist(f.draws,prob=T,xlim=c(-3,3),ylab='f(x)',main='Histogram of draws using A-R algorithm',xlab='x')
lines(x,f(x),type='l',col='green')
```



Problem 3: Reject-Accept Method Continued

8. Write a function `f` that takes as input a vector `x` and returns a vector of `f(x)` values. Plot the function between -5 and 5.

```r
f <- function(x){
  return(1/sqrt(2*pi)*exp(-x^2/2))
}
x <- seq(-5,5,0.01)
plot(x,f(x),type='l',col='blue',main='Standard Normal Density')
```

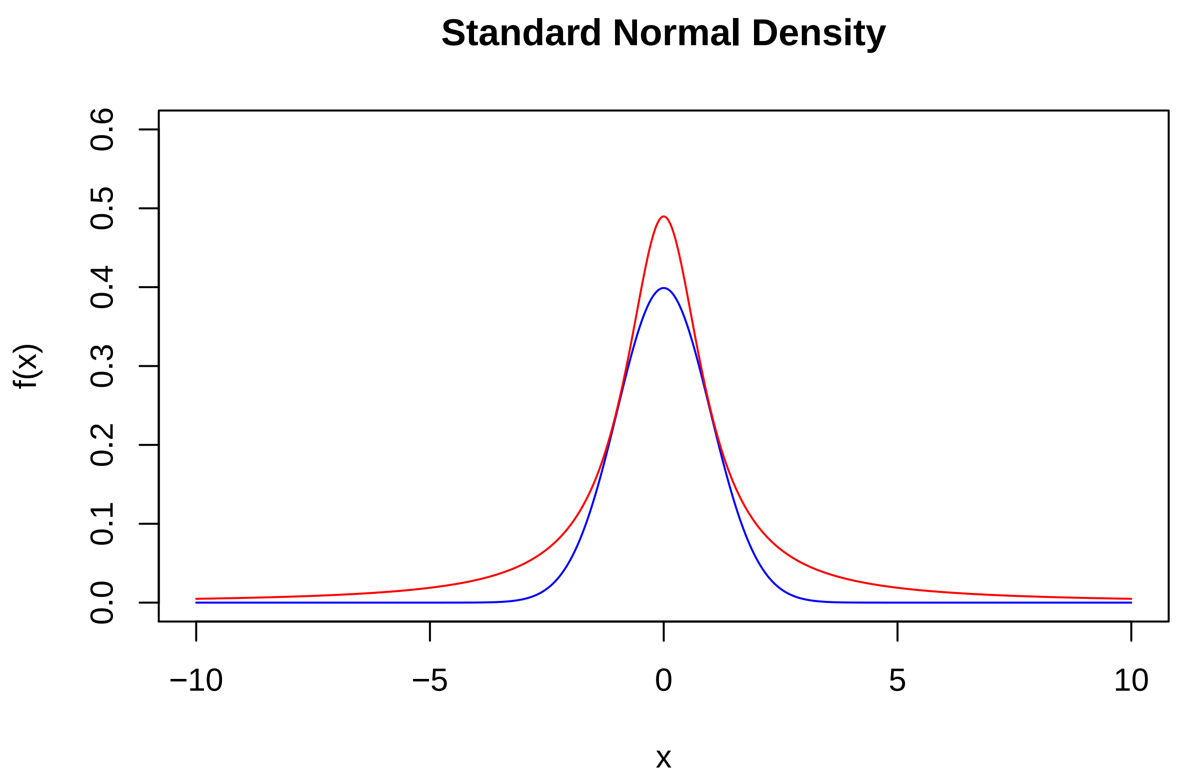


9. Let the known density `g` be the Cauchy density defined by pdf. Write a function `e` that takes as input a vector `x` and constant α $(0 < \alpha < 1)$ and returns a vector of `e(x)` values. The envelope function should be defined as $e(x) = g(x)/\alpha$.

```
e <- function(x,alpha){
  return(1/(alpha*pi*(1+x^2)))
}
```

10. Determine a 'good' value of α. You can solve this problem graphically. To show your solution, plot both $f(x)$ and $e(x)$ on the interval [-10,10].

```
alpha=0.65
x <- seq(-10,10,0.01)
plot(x,f(x),type='l',col='blue',ylim=c(0,.6),main='Standard Normal Density')
lines(x,e(x,alpha),col='red')
```



```
all(e(x,alpha)>f(x))
```

```
## [1] TRUE
```

A good α would be 0.65.

11. Write a function named `normal.sim` that simulates `n` standard normal random variables using the **Accept-Reject Algorithm**. The function should also use the **Inverse Transformation** from Part 1. Test your function using `n=10` draws.

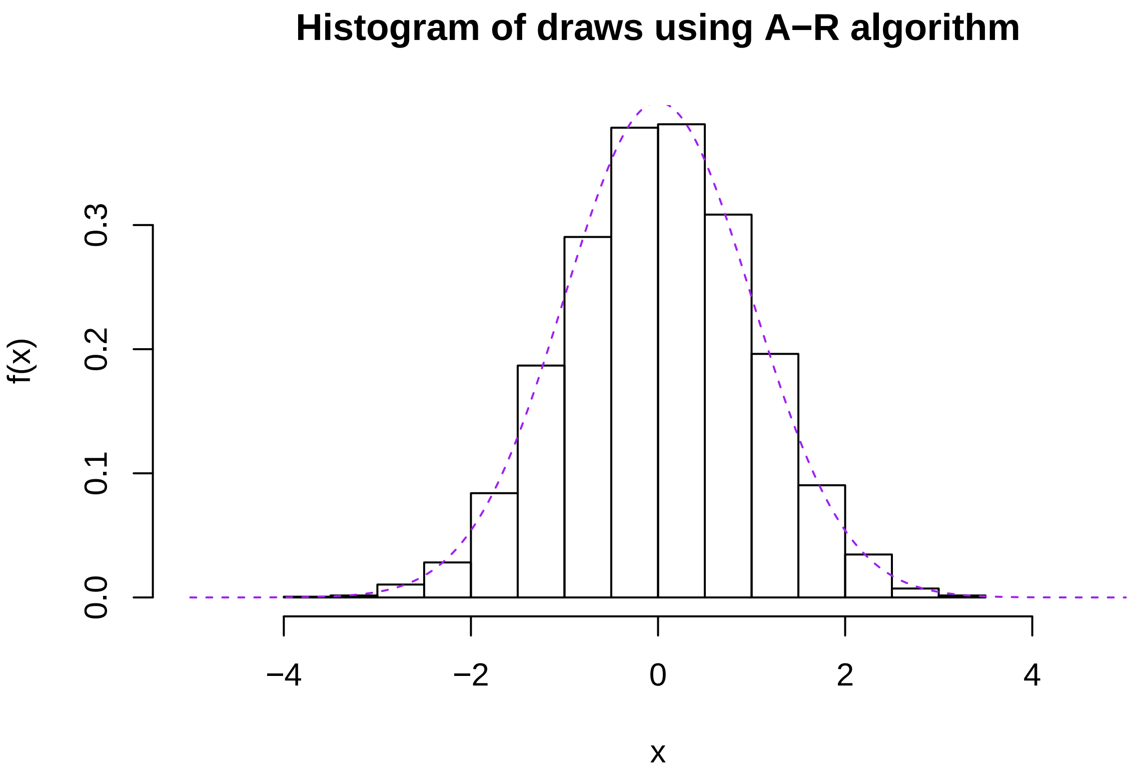
```
normal.sim <- function(n){
  n.samps <- n # number of samples desired
  k <- 0 # counter
  samps <- numeric(n.samps)
  while (k<n.samps){
    y <- cauchy.sim(1) #random draw from g
    u <- runif(1)
    if (u<f(y)/e(y,alpha=0.65)){
      k <- k+1
      samps[k] <- y
    }
  }
  return(samps)
}

normal.sim(10)
```

```
##  [1] -1.2733231 -0.3452198  0.1276412  0.9416057 -0.6668286  0.2782496
##  [7] -0.8005083  0.0507360 -0.8621806 -0.4424169
```

12. Using your function `normal.sim`, simulate 10,000 random draws from a standard normal distribution. Store the 10,000 draws in the vector `normal.draws`. Construct a histogram of the simulated standard

normal random variable with $f(x)$ overlaid on the graph. **Note**: when plotting the density curve over the histogram, include the argument `prob = T`.

```
normal.draws <- normal.sim(10000)
hist(normal.draws,prob=T,xlim=c(-5,5),xlab='x',ylab='f(x)',
     main='Histogram of draws using A-R algorithm')
x <- seq(-5,5,0.01)
lines(x,f(x),lty=2,col='purple')
```



Part 3: Simulation with Built-in R Functions

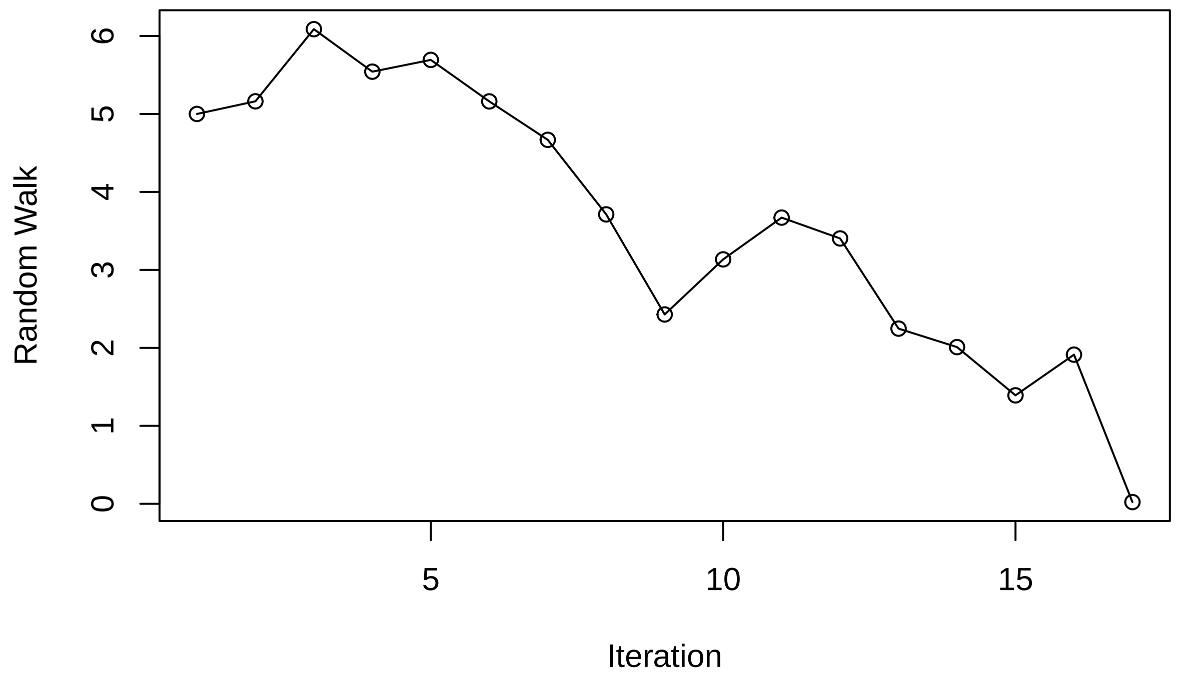
13. Write a `while()` loop to implement this procedure. Importantly, save all the positive values of x that were visited in this procedure in a vector called `x.vals`, and display its entries.

```
x <- 5  # initial value
k <- 0  # counter
x.vals <- NULL
x.vals[1] <- 5
while (x>0){
  k <- k+1
  r <- runif(1,-2,1)
  x <- x+r
  if (x>0)
    x.vals[k+1] <- x
}

x.vals
```

```
##  [1] 5.00000000 5.16201043 6.08741197 5.54216925 5.69312742 5.16089301
##  [7] 4.66768435 3.71182767 2.42847525 3.13491093 3.67054017 3.40338955
## [13] 2.24689415 2.00977852 1.39095443 1.91293909 0.02246046
```

14. Produce a plot of the random walk values `x.vals` from above versus the iteration number. Make sure the plot has an appropriately labeled x-axis and and y-axis. Also use `type="o"` so that we can see both points and lines.

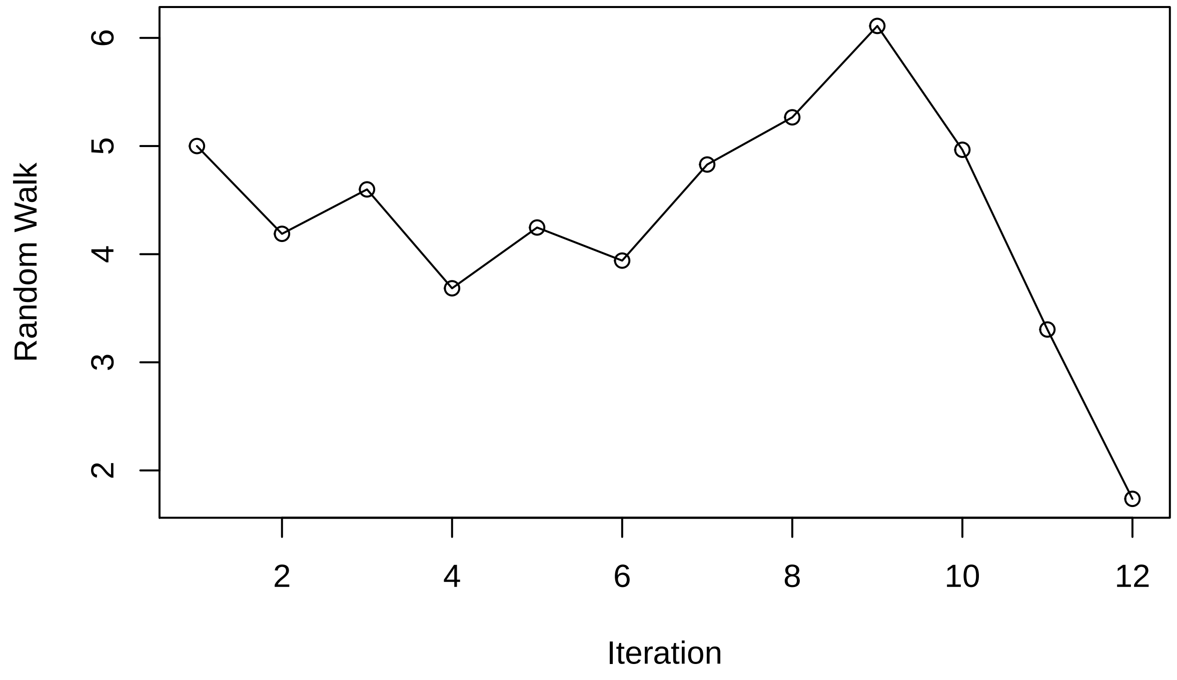
```
plot(x.vals, type = "o", xlab = 'Iteration', ylab = 'Random Walk')
```



15. Write a function `random.walk()` to perform the random walk procedure that you implemented in question (13). Its inputs should be: `x.start`, a numeric value at which we will start the random walk, which takes a default value of 5; and `plot.walk`, a boolean value, indicating whether or not we want to produce a plot of the random walk values `x.vals` versus the iteration number as a side effect, which takes a default value of `TRUE`. The output of your function should be a list with elements: `x.vals`, a vector of the random walk values as computed above; and `num.steps`, the number of steps taken by the random walk before terminating. Run your function twice with the default inputs, and then twice times with `x.start` equal to 10 and `plot.walk` = `FALSE`.

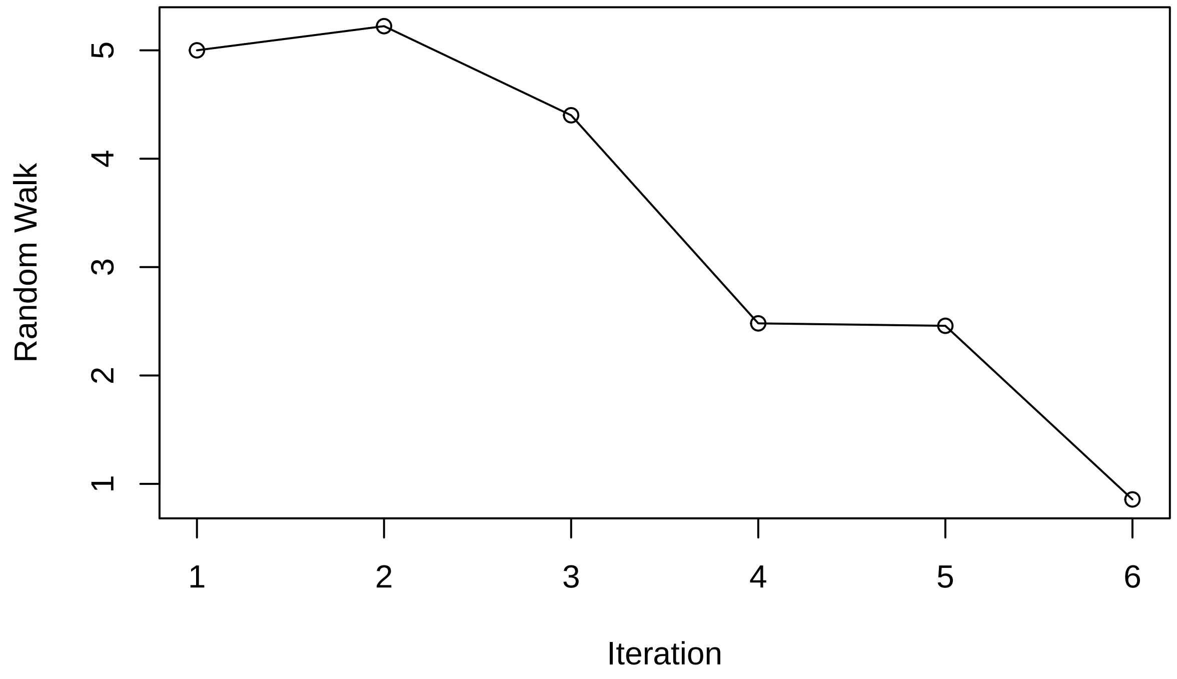
```
random.walk <- function(x.start=5,plot.walk=TRUE){
  x <- x.start # initiate
  k <- 0  # counter
  x.vals <- NULL
  x.vals[1] <- x
  while (x>0){
    k <- k+1
    r <- runif(1,-2,1)
    x <- x+r
    if (x>0)
      x.vals[k+1] <- x
  }
  if (plot.walk==TRUE){
    plot(x.vals, type = "o", xlab = 'Iteration', ylab = 'Random Walk')
  }
  return(list(x.vals=x.vals,num.steps=k))
}
```

```r
random.walk()
```



```
## $x.vals
##  [1] 5.000000 4.188118 4.598897 3.684343 4.246438 3.940477 4.829110
##  [8] 5.265688 6.110725 4.965344 3.303056 1.737144
##
## $num.steps
## [1] 12
```

```r
random.walk()
```



```
## $x.vals
## [1] 5.0000000 5.2227890 4.4008262 2.4809738 2.4585063 0.8567027
##
## $num.steps
## [1] 6
```

```r
random.walk(10,F)
```

```
## $x.vals
##  [1] 10.0000000 10.6666538 10.2424595  9.8672053  8.7790010  8.2524294
##  [7]  8.4197907  8.3303814  6.4427543  4.8668799  4.9602163  5.6038014
## [13]  5.3389273  3.7449302  2.2659345  2.7822164  3.3687783  1.4977571
## [19]  0.7585860  0.4998803  1.4773735  1.2628796  1.5448577
##
## $num.steps
## [1] 23
```

```r
random.walk(10,F)
```

```
## $x.vals
##  [1] 10.000000 10.948403 11.538170 11.695935 12.395112 10.921384 10.879116
##  [8] 10.376427  9.599251  9.524665  9.433029  9.635237  9.491667  9.793536
## [15]  8.807033  8.226065  6.427942  5.970875  6.116639  4.670549  4.692955
## [22]  2.713045  3.268956  3.537524  4.490203  5.363265  4.498462  2.528369
## [29]  3.261957  1.366087  2.152878  3.149659  3.194824  3.197720  1.965129
## [36]  1.682443  1.864844  1.269908  2.205776  2.391307  2.559505  2.851576
## [43]  1.066293  1.506177  1.908744
##
## $num.steps
## [1] 45
```

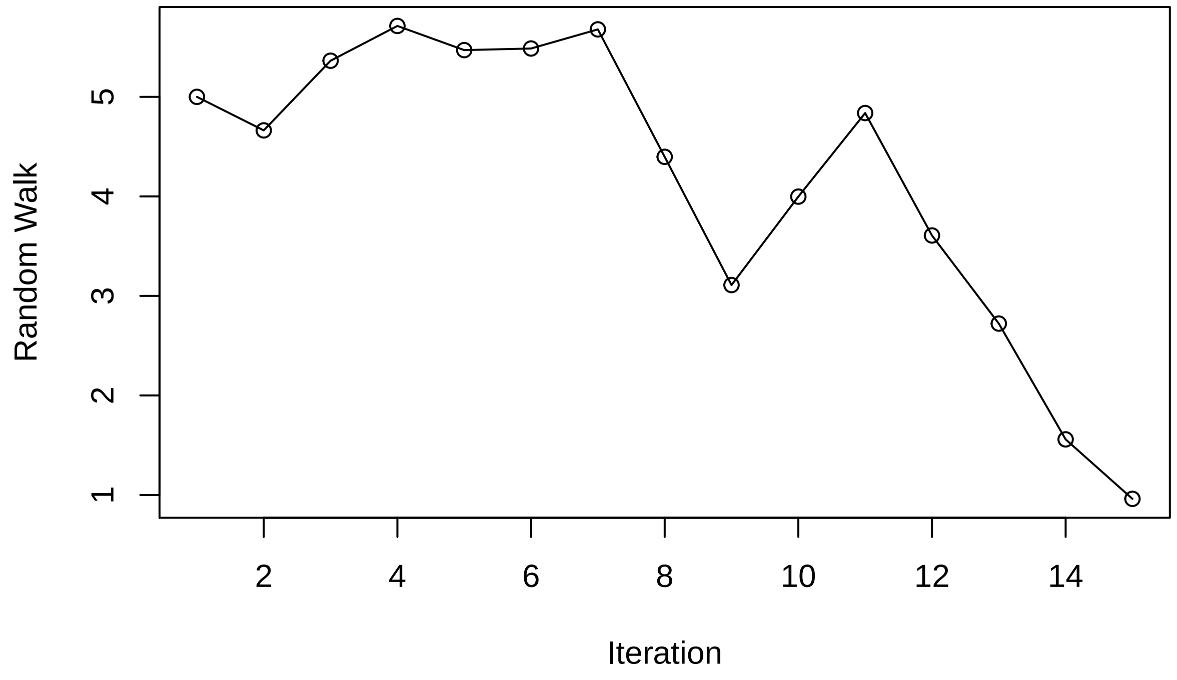
16. We'd like to answer the following question using simulation: if we start our random walk process, as defined above, at $x = 5$, what is the expected number of iterations we need until it terminates? To estimate the solution produce 10,000 such random walks and calculate the average number of iterations in the 10,000 random walks you produce. You'll want to turn the plot off here.

```r
total.steps <- 0
for (i in 1:10000){
  total.steps <- total.steps+random.walk(5,F)$num.steps
}
total.steps/10000
```

```
## [1] 11.2022
```

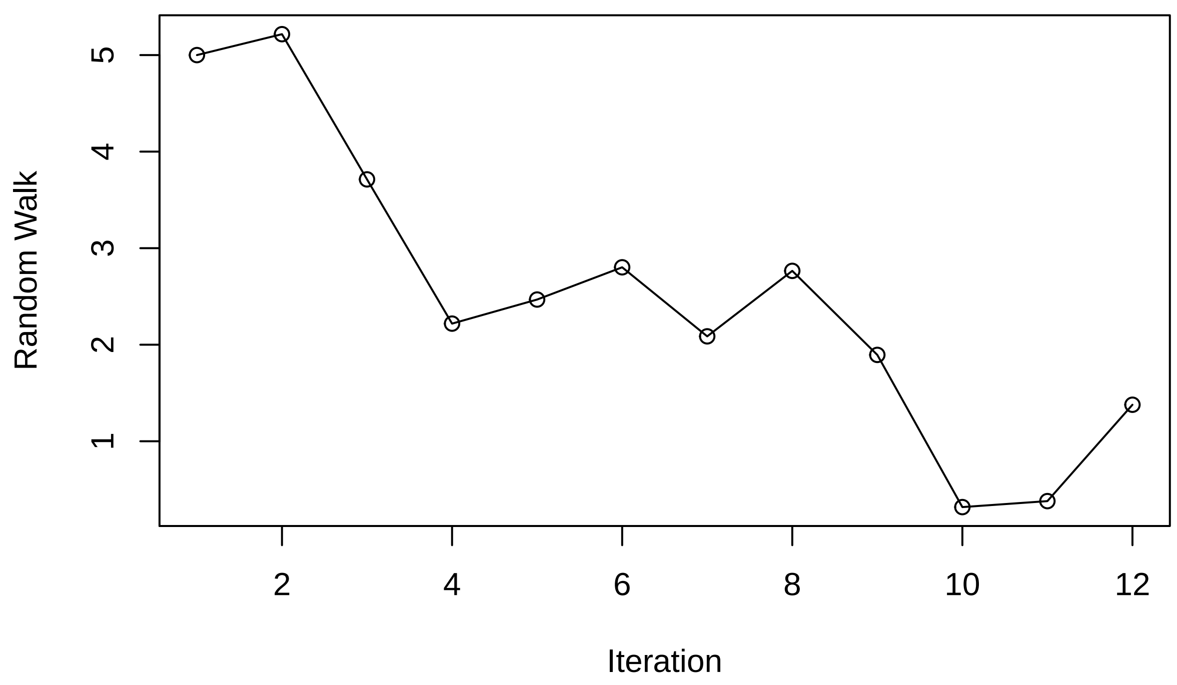
17. Modify your function `random.walk()` defined previously so that it takes an additional argument seed: this is an integer that should be used to set the seed of the random number generator, before the random walk begins, with `set.seed()`. But, if seed is `NULL`, the default, then no seed should be set. Run your modified function `random.walk()` function twice with the default inputs, then run it twice with the input seed equal to (say) 33 and `plot.walk = FALSE`.

```r
random.walk <- function(x.start=5,plot.walk=TRUE,seed=NULL){
  if (!is.null(seed)){
    set.seed(seed)
  }
  x <- x.start # initiate
  k <- 0  # counter
  x.vals <- NULL
  x.vals[1] <- x
  while (x>0){
    k <- k+1
    r <- runif(1,-2,1)
    x <- x+r
    if (x>0)
      x.vals[k+1] <- x
  }
  if (plot.walk==TRUE){
    plot(x.vals, type = "o", xlab = 'Iteration', ylab = 'Random Walk')
  }
  return(list(x.vals=x.vals,num.steps=k))
}

random.walk()
```



```
## $x.vals
##  [1] 5.0000000 4.6630875 5.3630111 5.7122392 5.4695190 5.4855825 5.6778722
##  [8] 4.3973235 3.1091635 3.9974965 4.8372049 3.6072386 2.7218109 1.5584614
## [15] 0.9608364
##
## $num.steps
## [1] 15
```

```r
random.walk()
```



```
## $x.vals
##  [1] 5.0000000 5.2164191 3.7126381 2.2185513 2.4678568 2.8018661 2.0866992
##  [8] 2.7647447 1.8945273 0.3182150 0.3801195 1.3782397
##
## $num.steps
## [1] 12
```

```r
random.walk(plot.walk=F,seed=33)
```

```
## $x.vals
##  [1] 5.0000000 4.3378214 3.5217724 2.9729590 3.7295869 4.2612312 3.8132800
##  [8] 3.1246550 2.1542497 0.2008006
##
## $num.steps
## [1] 10
```

```r
random.walk(plot.walk=F,seed=33)
```

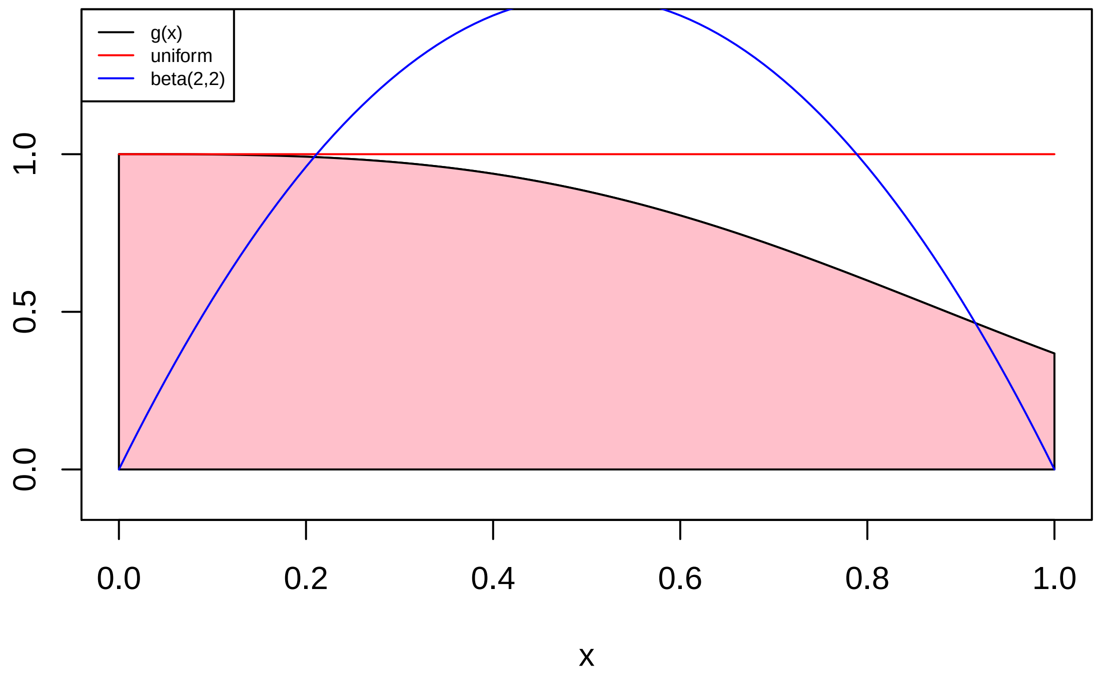
```
## $x.vals
##  [1] 5.0000000 4.3378214 3.5217724 2.9729590 3.7295869 4.2612312 3.8132800
##  [8] 3.1246550 2.1542497 0.2008006
##
## $num.steps
## [1] 10
```

Part 4: Monte Carlo Integration

18. Run the following code:

```r
g <- function(x) {
  return(exp(-x^3))
}

x <- seq(0,1,.01)
```

```r
alpha <- 2
beta <- 2

plot(x,g(x),type="l",xlab="x",ylab="",ylim=c(-.1,1.4))
polygon(c(0,seq(0,1,0.01),1),c(0,g(seq(0,1,0.01)),0) ,col="pink")
lines(x,rep(1,length(x)),col="red")
lines(x,dbeta(x,shape1=alpha,shape2=beta),col="blue")

legend("topleft",legend=c("g(x)","uniform","beta(2,2)"),
       lty=c(1,1,1),col=c("black","red","blue"),cex=.6)
```



19. Using **Monte Carlo Integration**, approximate the integral$\int_0^1 e^{-x^3} dx$ using n=1000^{2}random draws from the distribution uniform(0,1).

```r
g.over.p <- function(x){
  return(exp(-x^3))
}
mean(g.over.p(runif(1000^2)))
```

```
## [1] 0.8077283
```

20. Using **Monte Carlo Integration**, approximate the integral$\int_0^1 e^{-x^3} dx$ using n=1000^{2}random draws from the distribution beta($\alpha = 2, \beta = 2$).

```r
mean(g.over.p(rbeta(1000^2,shape1=2,shape2=2)))
```

```
## [1] 0.8352311
```