Homework 5

Maxime Grossman

6/18/2021

```
library(png)
library(stats)
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.6.3
```

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

Part I: Inverse Transform

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

```
u <- runif(1, min=0, max=1)
x <- tan(pi*(u-1/2))


pp <- readPNG("cauchy.png")
plot.new()
rasterImage(pp,0,0,1,1)
```

$$f(x) = \frac{1}{\pi(1+x^2)} \qquad x \epsilon R$$

$$F(x) = 0.5 + \frac{\tan^{-1} x}{\pi} \qquad x \epsilon R$$

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.

```
cauchy.sim <- function(n){

  u <- runif(n)
  return(tan(pi*(u-1/2)))

}

cauchy.sim(10)
```

```
##  [1] -0.9070131 -0.4248394  0.2329576  3.3710605 -1.3611982  3.0255173
##  [7]  5.6954298  0.5530230  0.4294381 -5.0869244
```

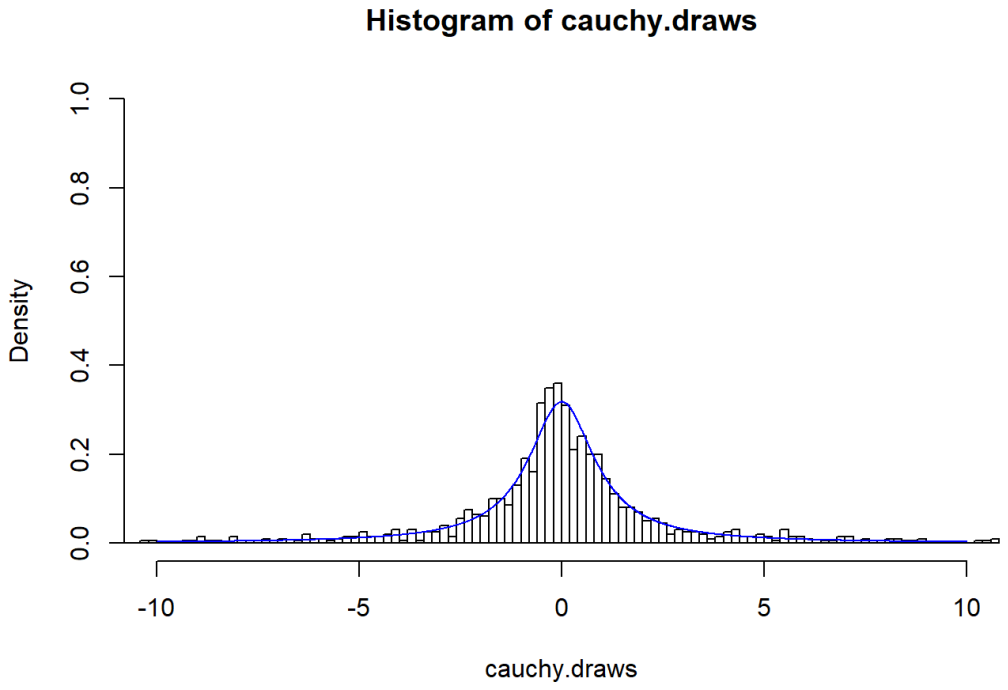
```
# we can pull 10 random cauchy RVs and eyeball the similarity

rcauchy(10)
```

```
##  [1]   0.75561999   0.61954832  -1.50147326   2.62405381  -0.88250440
##  [6] 138.34760311  -1.22737450  -0.02543323   2.52652746  -0.84088155
```

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) 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).

```
cauchy.sim <- function(n){

  u <- runif(n)
  return(tan(pi*(u-1/2)))

}


cauchy.draws <- cauchy.sim(1000)


hist(cauchy.draws, prob=TRUE, breaks=30000, ylim=c(0,1), xlim=c(-10,10))

y<-seq(-10,10,.01)

lines(y,(1/pi)*(1/(1+y^2)), col="blue")
```



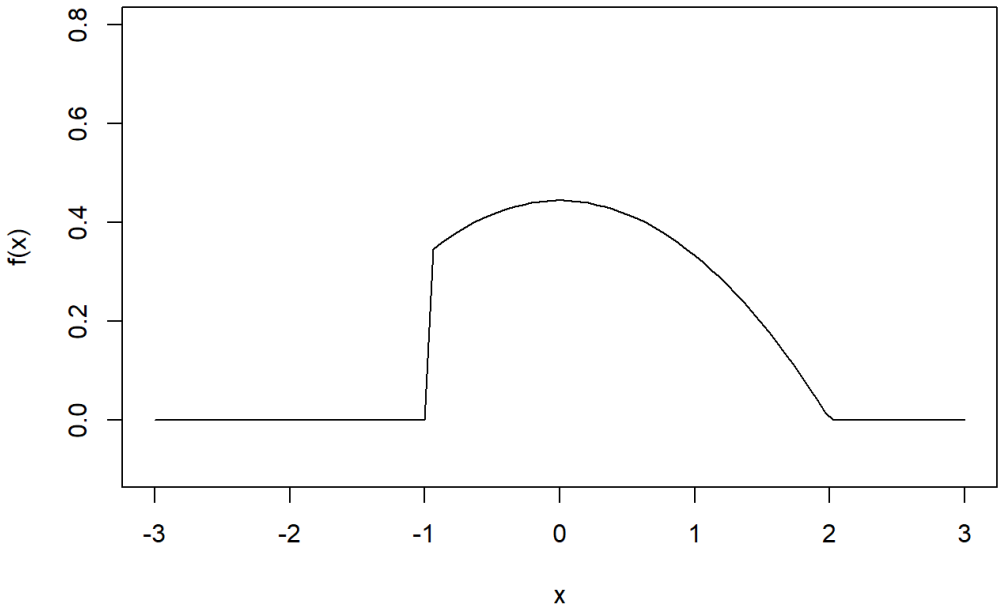
Part 2: Accept-Reject Method

Let random variable X denote the temperature at which a certain chemical reaction takes place. Suppose that X has probability density function shown.

Perform the following tasks:

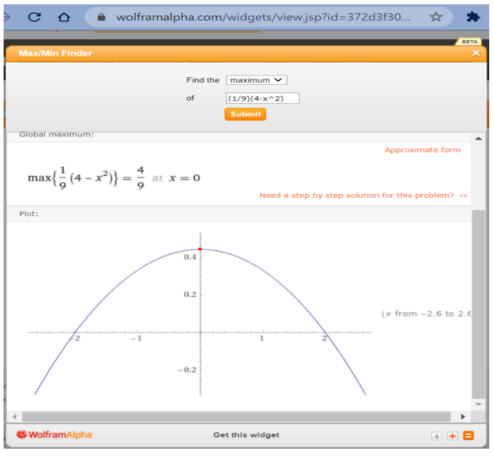
4. Write a function f that takes as input a vector x and returns a vector of f(x) values. Plot the function between −3 and 3. Make sure your plot is labeled appropriately.

```
f <-function(x)
  {return(ifelse((x<=-1|x>=2), 0, (1/9)*(4-x^2)))
  }

x <- seq(-3,3,length=100)

plot(x,f(x),type="l",ylab="f(x)", xlim = c(-3,3), ylim=c(-0.1, 0.8))
```



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.

```
pp <- readPNG("max.png")
plot.new()
rasterImage(pp,0,0,0.6,1)
```



Using Wolfram, we find that the maximum of the function is 4/9 and occurs at x = 0.

An appropriate envelope would be a horizontal line at 4/9. So y = 4/9.

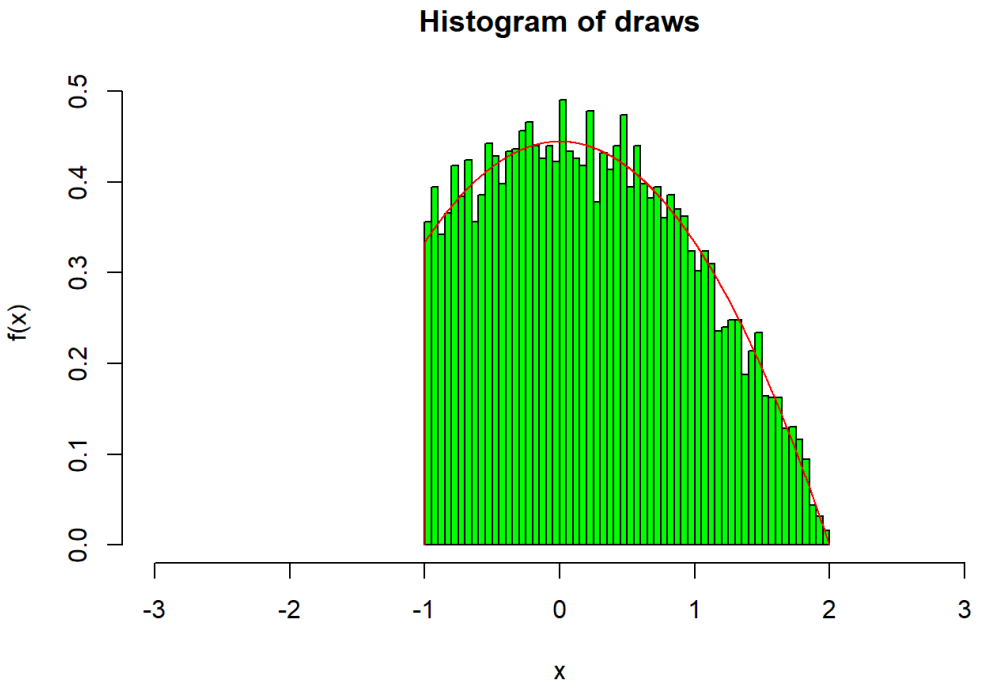
```
# NOTES:

# sample Y ~ g
# sample U ~ Unif(0,1)

# If U < f(Y)/e(Y), accept Y
# Set X = Y

# g can be our uniform distribution
# alpha = 1/max(f(x)) = 1/(4/9) = 9/4
# e(x) = 4/9

# Draw 2 uniforms, U1 and U2
# If U1 < f(U2)*9/4, accept U2

# function  is 1/9(4-x^2) for -1 <= x <= 2

# e(x) = g(x)/alpha = 1/(9/4) = 4/9 = f.max >= f(x)


x.max <- 4/9
f.max <- (1/9)*(4-(x.max)^2)

e <- function(x){
  return(ifelse((x<=-1|x>=2), Inf, f.max))
}
```

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.

```
x.max <- 4/9
f.max <- (1/9)*(4-(x.max)^2)

e <- function(x){
  return(ifelse((x<=-1|x>=2), Inf, f.max))
}



n.samps <- 10000  # we want 10,000 samples
n <- 0  # counter

f.draws <- numeric(n.samps)

while (n < n.samps) {
  y <- runif(1, min = -1, max = 2)
  u <- runif(1)

  if(u < f(y)/e(y)){      # f(y) = 1/9(4-x^2)
    n <- n + 1            # e(y) = 4/9
    f.draws[n] <- y         # if a random Uniform U1 is less than 1/4(4-U2^2)
  }                       # copy down the U2
}
```

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

```
x <- seq(-1,2, length = 1000)

hist(f.draws, probability = T, ylab = "f(x)", xlab = "x", main = "Histogram of draws", breaks = 50, xlim = c(-3,3), col =
"green")
lines(x, f(x), col = "red")
```



Problem 3: Accept-Reject Method Continued

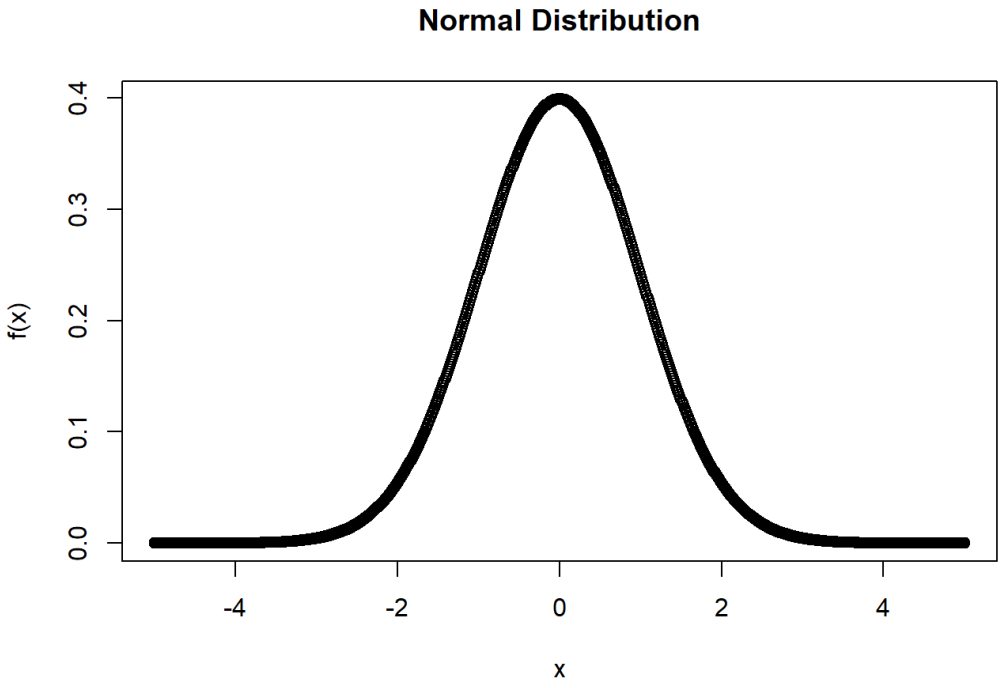
Consider the standard normal distribution X with probability density function shown.

In this exercise, we will write a function named normal.sim that simulates a standard normal random variable using the Accept-Reject Algorithm.

Perform the following tasks:

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. Make sure your plot is labeled appropriately.

```
st.norm <- function(x){
  return(1/sqrt(2*pi)*exp(-x^2/2))
}

s <- seq(-5,5, by=.01)

plot(s,st.norm(s), xlab = "x", ylab = "f(x)", main = "Normal Distribution")
```



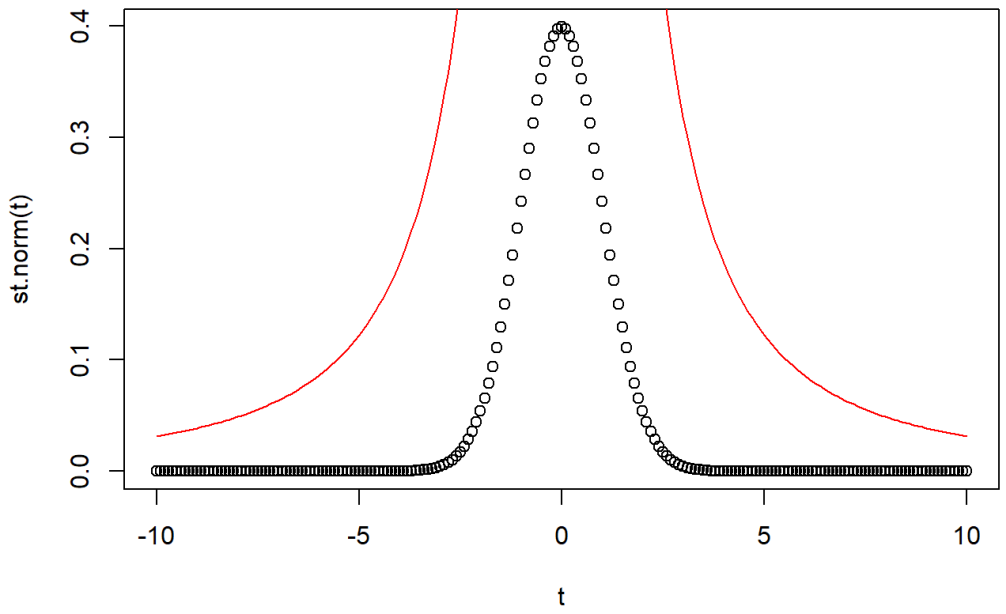
9. Let the known density g be the Cauchy density defined by pdf shown.

Write a function e that takes as input a vector x and constant alpha (0 < α < 1) and returns a vector of e(x) values. The envelope function should be defined as e(x) = g(x)/α.

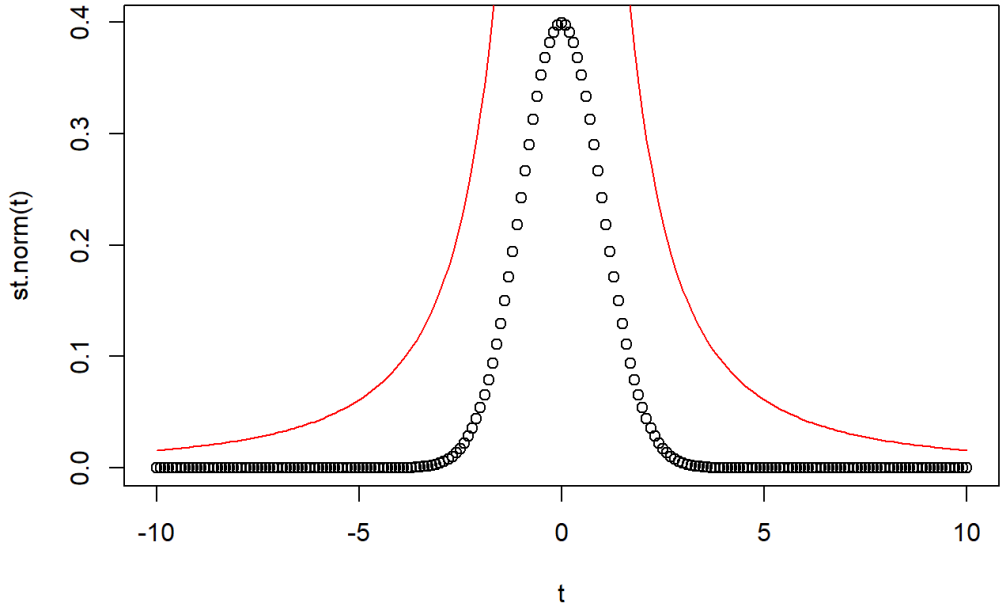
```
e <- function(x, alpha) {
  return(1/(alpha*pi*(1+x^2)))
  }
```

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].

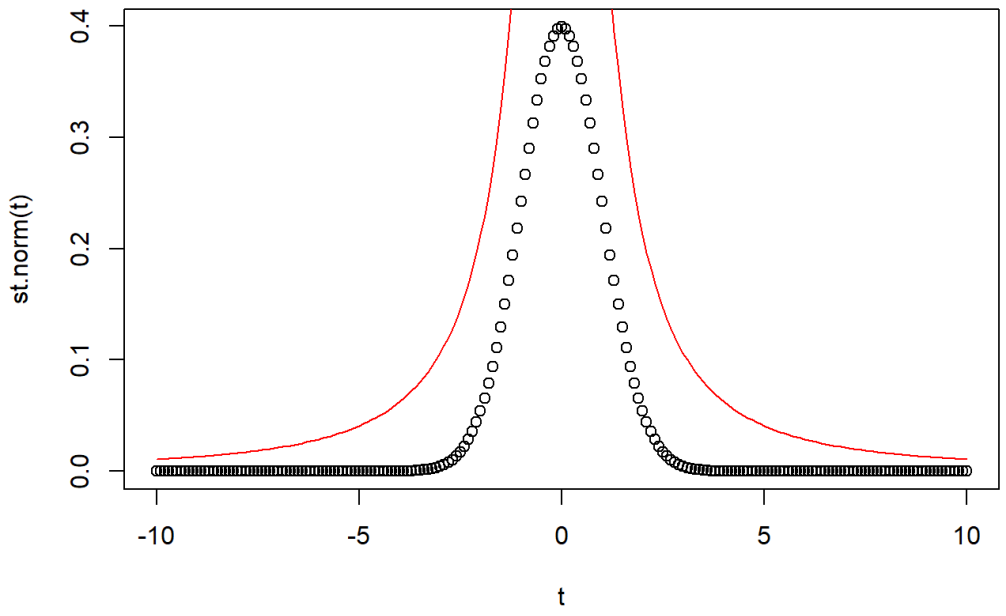
```
t <- seq(-10,10, by=0.1)



plot(t, st.norm(t))
lines(t,e(t,alpha=0.1), col="red")
```



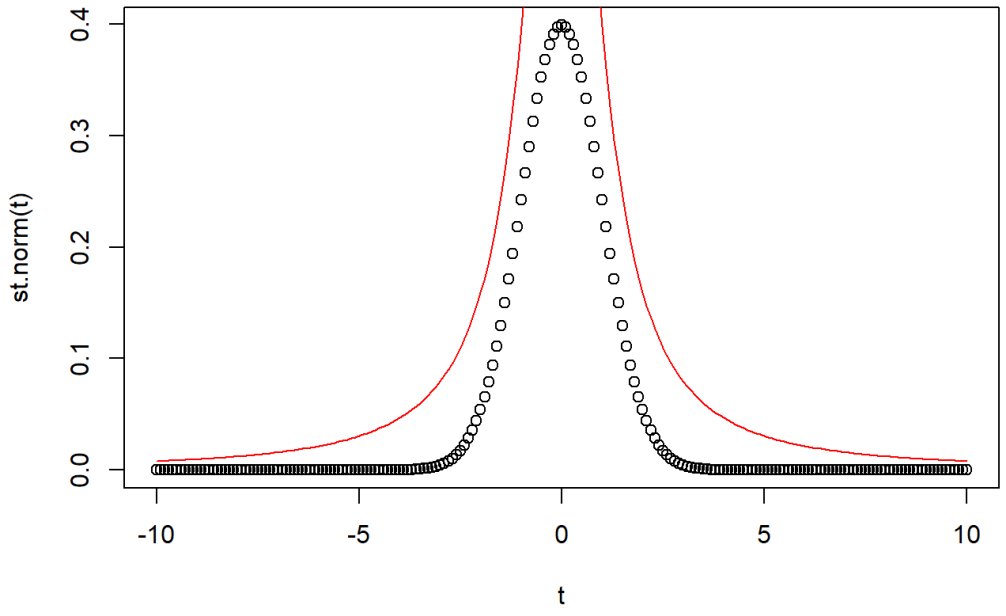

```
plot(t, st.norm(t))
lines(t,e(t,alpha=0.2), col="red")
```



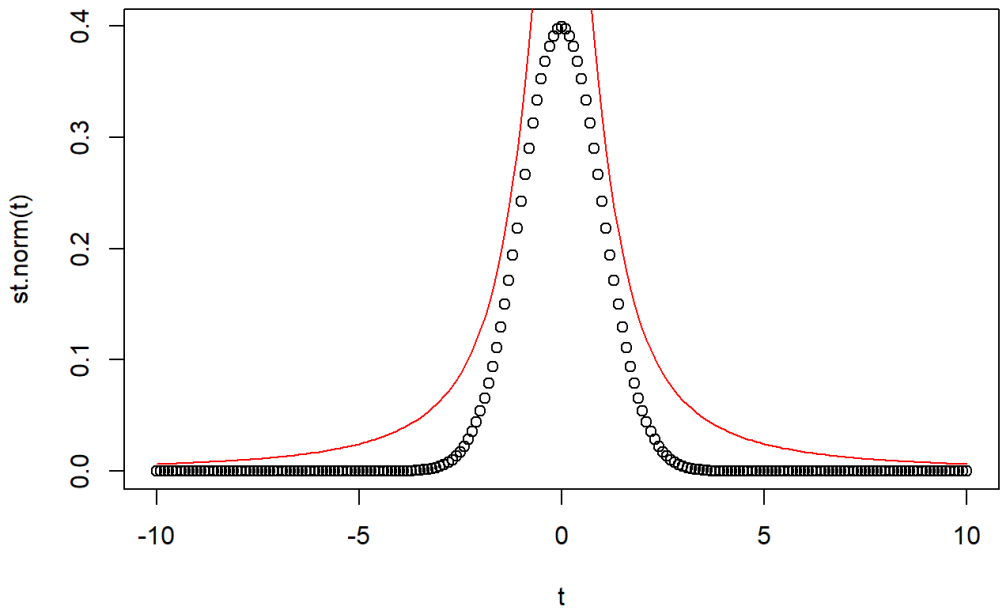

```
plot(t, st.norm(t))
lines(t,e(t,alpha=0.3), col="red")
```



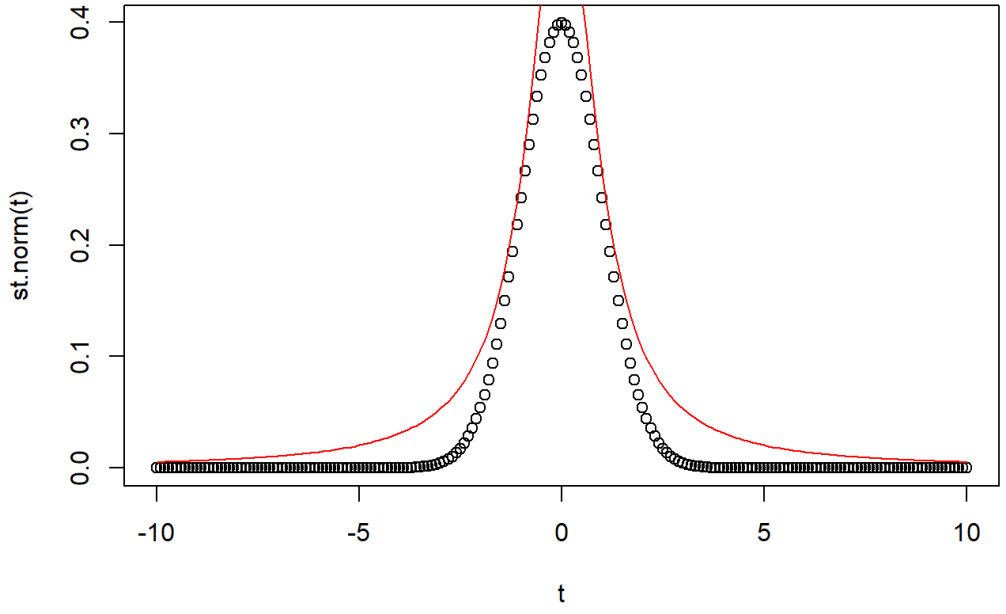

```
plot(t, st.norm(t))
lines(t,e(t,alpha=0.4), col="red")
```



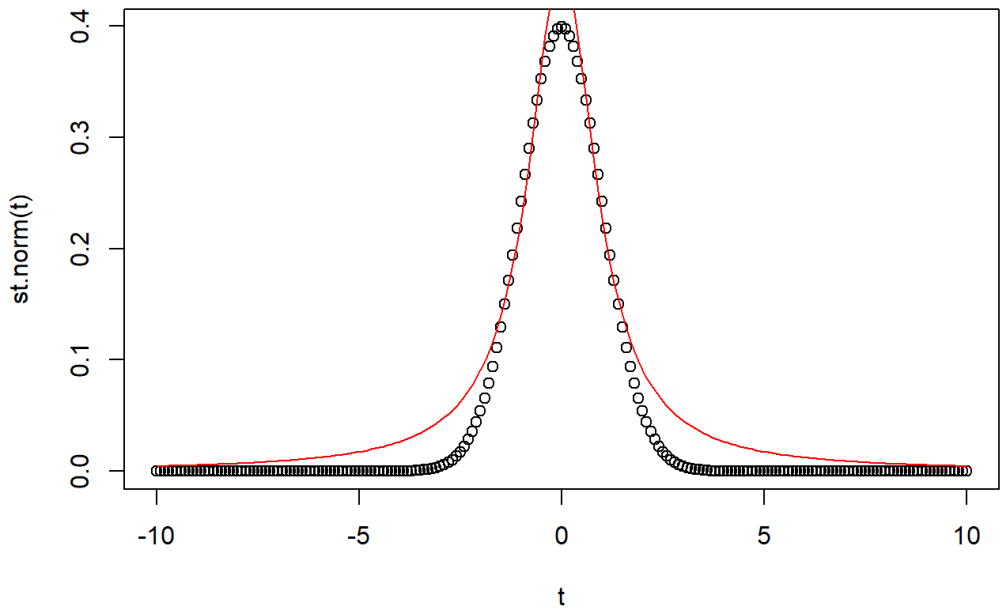

```
plot(t, st.norm(t))
lines(t,e(t,alpha=0.5), col="red")
```



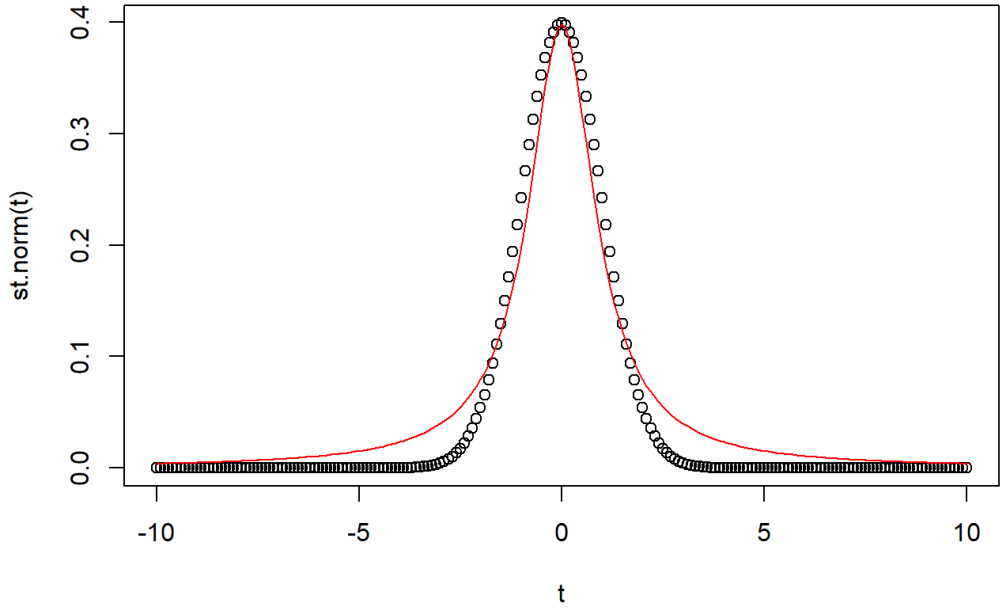

```
plot(t, st.norm(t))
lines(t,e(t,alpha=0.6), col="red")
```




```
plot(t, st.norm(t))
lines(t,e(t,alpha=0.7), col="red")
```




```
plot(t, st.norm(t))
lines(t,e(t,alpha=0.8), col="red")
```




```
all(e(t, alpha=0.5)>st.norm(t))
```

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

```
all(e(t, alpha=0.6)>st.norm(t))
```

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

```
all(e(t, alpha=0.7)>st.norm(t))
```

```
## [1] FALSE
```

```
all(e(t, alpha=0.8)>st.norm(t))
```

```
## [1] FALSE
```

```
## Answer is in between 0.6 and 0.7


all(e(t, alpha=0.65)>st.norm(t))
```

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

Choose the parameter alpha = 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.samps) {
  n.samps <- 10   # number of samples desired
  n       <- 0           # counter for number samples accepted
  samps   <- numeric(n.samps) # initialize the vector of output
  while (n < n.samps) {
    y <- cauchy.sim(1)    #random draw from g
    u <- runif(1)
    if (u < st.norm(y)/e(y, alpha=0.65)) {
      n        <- n + 1
      samps[n] <- y
    }
  }
  return(samps)
}



normal.sim(10)
```

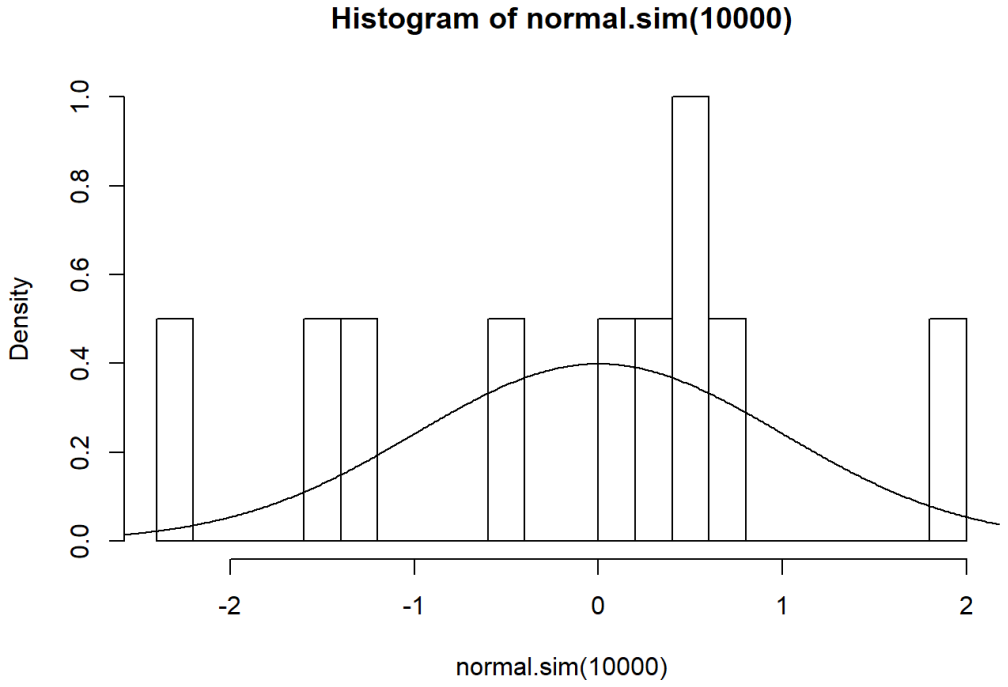
```
##  [1]  1.210029198  0.001927655  0.003139828 -0.231729614 -1.062687247
##  [6] -1.389677309  0.655417851  0.637201362 -0.950268734  0.046590990
```

```
#hist(normal.sim(10000),prob=T,breaks=30,xlim=c(-10,10))
#lines(x,normal.sim(x),col="blue")
```

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(s, normal.draws, breaks=20, prob = TRUE)
#lines(s, st.norm(s))


hist(normal.sim(10000), breaks=20, prob = T)
lines(s, st.norm(s))
```



Part 3: Simulation with Built-in R Functions

Consider the following “random walk” procedure:

i. Start with x= 5

ii. Draw a random number r uniformly between−2 and 1.

iii. Replace x with x + r

iv. Stop if x ≤ 0

v. Else repeat

Perform the following tasks:

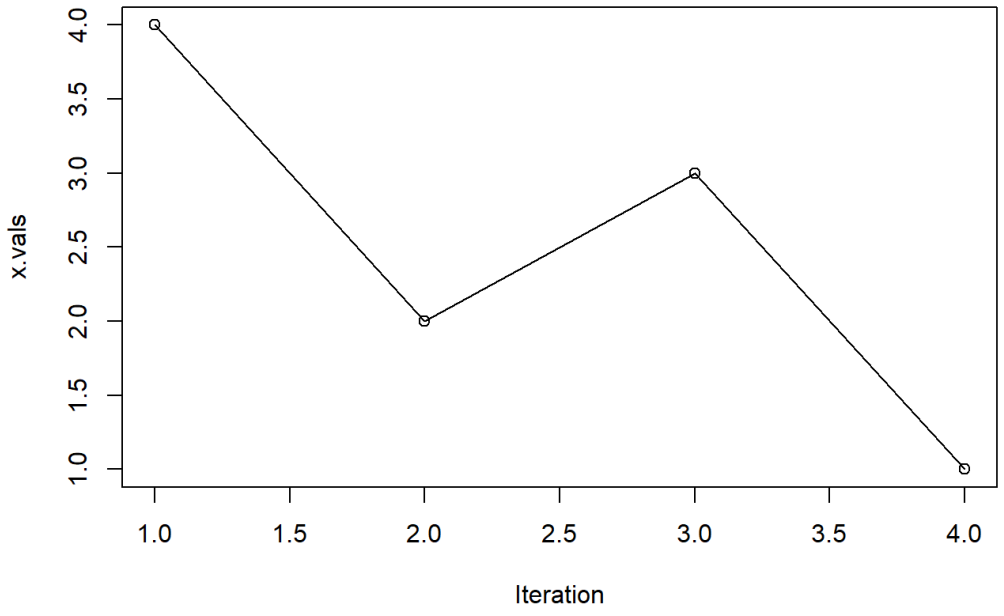
13. Write a while() loop to implement this procedure. Importantly, save all the positive values of x that were visited in this procedure in a vector called x.vals, and display its entries.

```
x <- 5
x.vals <- NULL
n <- 1

while(x > 0){

  x <- x + sample(-2:1, 1, replace = TRUE)

    if(x>0){
    x.vals[n] <- x
    n <- n+1
  }
}

x.vals
```

```
## [1] 4 2 3 1
```

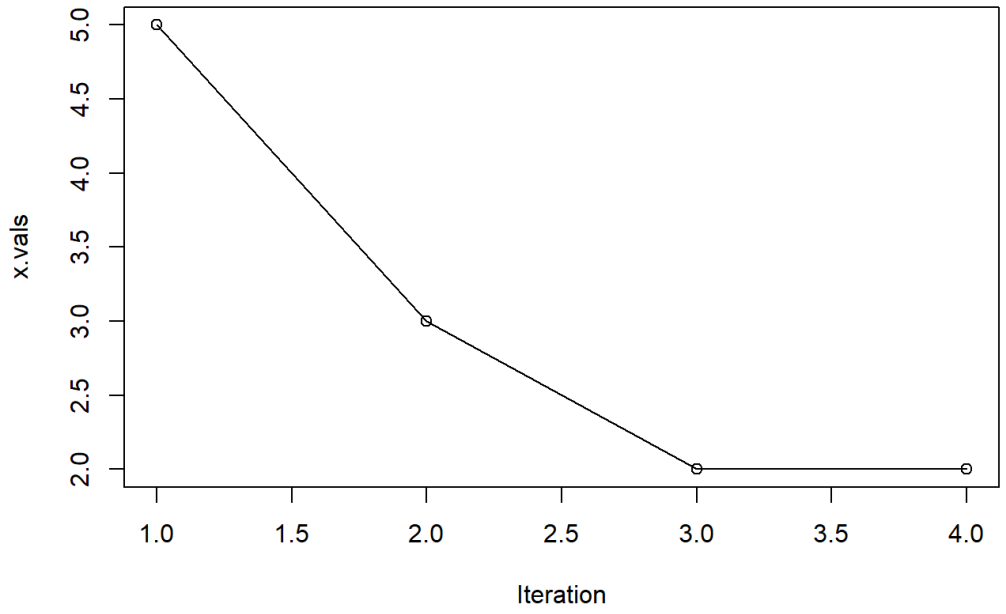
14. Produce a plot of the random walk values x.vals from above versus the iteration number. Make sure the plot has an appropriately labeled x-axis and and y-axis. Also use type="o" so that we can see both points and lines.

```
plot(x.vals, type = "o", xlab="Iteration")
```



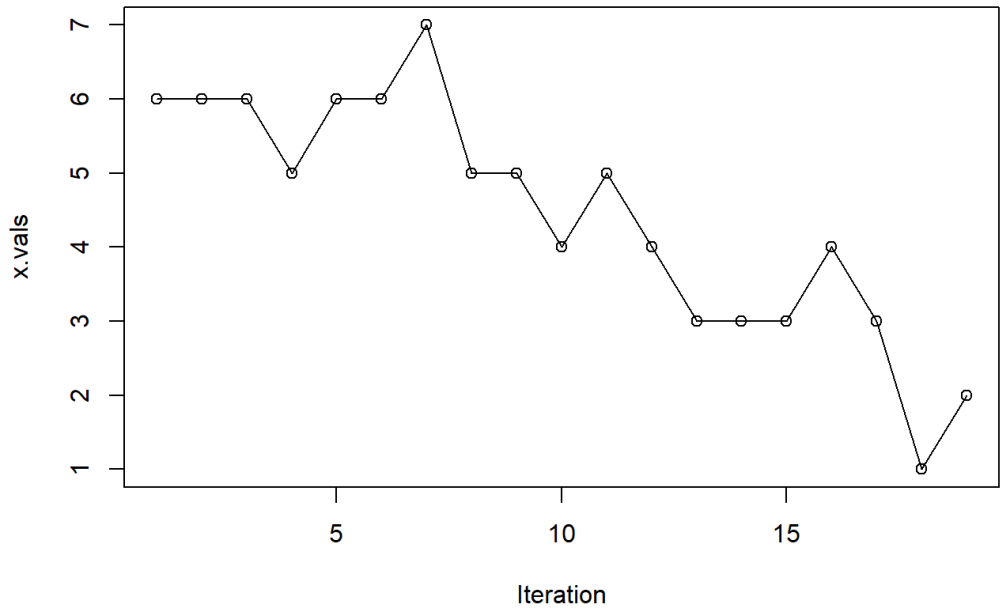
15. Write a function random.walk() to perform the random walk procedure that you implemented in question (13). Its inputs should be: x.start, a numeric value at which we will start the random walk, which takes a default value of 5; and plot.walk, a boolean value, indicating whether or not we want to 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 withthe default inputs, and then twice times with x.start equal to 10 and plot.walk = FALSE.

```
random.walk <- function(x.start=5, plot.walk){

x.vals <- NULL
n <- 1

while(x.start > 0){

  x.start <- x.start + sample(-2:1, 1, replace = TRUE)

    if(x.start>0){
    x.vals[n] <- x.start
    n <- n+1
    }}


  if(plot.walk=="TRUE"){
    plot(x.vals, type = "o", xlab="Iteration")
    }

#x.vals

#num.steps <- length(x.vals)
#num.steps

cat("The values of x are:", x.vals, "\n")
cat("The number of steps are:", n-1, "\n")

}


random.walk(x.start=5, plot.walk="TRUE")
```




```
## The values of x are: 5 3 2 2
## The number of steps are: 4
```

```
random.walk(x.start=5, plot.walk="TRUE")
```




```
## The values of x are: 6 6 6 5 6 6 7 5 5 4 5 4 3 3 3 4 3 1 2
## The number of steps are: 19
```

```
random.walk(x.start=10, plot.walk="FALSE")
```

```
## The values of x are: 8 7 6 6 6 7 8 9 9 7 5 5 3 3 4 4 4 3 3 3 1 2
## The number of steps are: 22
```

```
random.walk(x.start=10, plot.walk="FALSE")
```

```
## The values of x are: 8 7 8 8 9 10 11 12 12 13 14 12 10 10 8 6 6 4 4 2
## The number of steps are: 20
```

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.

```
# We slightly modify our former function to add a "counter" and a "sum"


random.walk.count <- function(x.start=5, plot.walk){

x.vals <- NULL
n <- 1
counter <- 0
sum <- 0

while(counter <= 10000){
  while(x.start > 0){

    x.start <- x.start + sample(-2:1, 1, replace = TRUE)

    if(x.start>0){
    x.vals[n] <- x.start
    n <- n+1
    }
  }

counter <- counter+1
sum <- sum + (n-1)

             if(plot.walk=="TRUE"){
             plot(x.vals, type = "o", xlab="Iteration")
               }

#x.vals

#num.steps <- length(x.vals)
#num.steps



}
return(sum/10000)
}

random.walk.count(x.start=5, plot.walk="FALSE")
```

```
## [1] 13.0013
```

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.

```
random.walk.count <- function(x.start=5, plot.walk, seed){

x.vals <- NULL
n <- 1
counter <- 0
sum <- 0

if(seed!="NULL"){
  set.seed(seed)
}


while(counter <= 10000){
  while(x.start > 0){

    x.start <- x.start + sample(-2:1, 1, replace = TRUE)

    if(x.start>0){
    x.vals[n] <- x.start
    n <- n+1
    }
  }

counter <- counter+1
sum <- sum + (n-1)

             if(plot.walk=="TRUE"){
             plot(x.vals, type = "o", xlab="Iteration")
               }

#x.vals

#num.steps <- length(x.vals)
#num.steps



}
return(sum/10000)
}




random.walk.count(x.start=5, plot.walk="FALSE", seed=33)
```

```
## [1] 9.0009
```

```
random.walk.count(x.start=5, plot.walk="FALSE", seed=33)
```

```
## [1] 9.0009
```

```
random.walk.count(x.start=5, plot.walk="FALSE", seed=33)
```

```
## [1] 9.0009
```

```
random.walk.count(x.start=5, plot.walk="FALSE", seed=33)
```

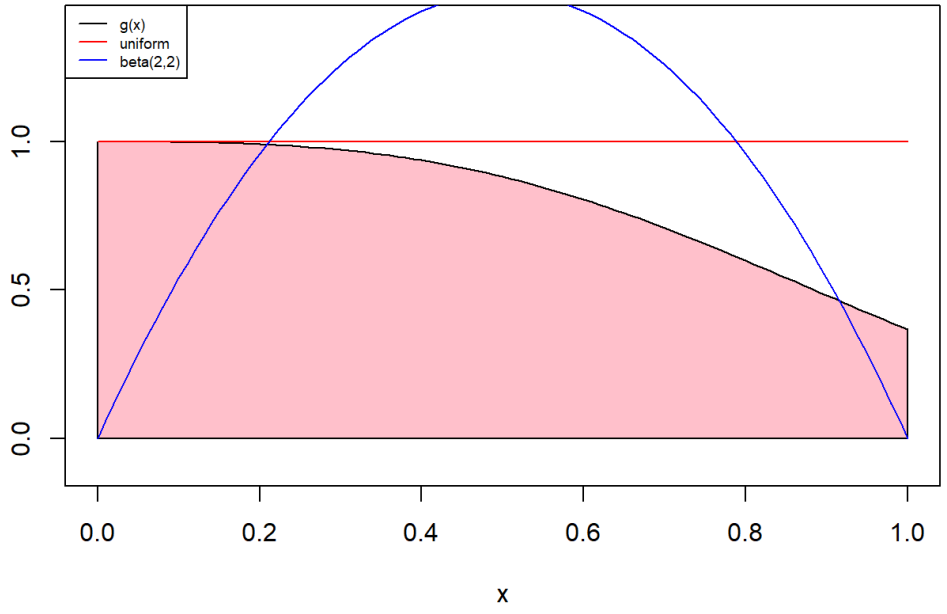
```
## [1] 9.0009
```

Part 4: Monte Carlo Integration

Perform the following tasks:

18. Run the code given.

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

x <- seq(0,1,.01)
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 with n = 1000^2 random draws from the distribution uniform(0,1).

First, we can find the actual value of the integral:

```
q <- function(x){
  (exp(-x^3))
}

integrate(q, 0, 1)
```

```
## 0.8075112 with absolute error < 9e-15
```

Now we move onto the question:

```
# To estimate, draw from p and take the sample mean of

# f(x) = g(x) / p(x)


g.over.p <- function(x){
  return(exp(-x^3))
  }

mean(g.over.p(runif(1000^2))) #Tryn=10000
```

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

The result is consistent with our known answer. We know that the integral is about 0.807511

20. Using Monte Carlo Integration, approximate the integral with n= 1000^2 random draws from the distribution beta(α= 2, β= 2).

```
# To estimate, draw from p and take the sample mean of

# f(x) = g(x) / p(x)

# Here, we can say p(x) is Uniform because our integral spans from 0 to 1

# We know that the integral is about 0.807511


g.over.p <- function(x){
  return((exp(-x^3))/(6*(x-x^2)))
  }

mean(g.over.p(rbeta(1000^2, 2, 2))) #Tryn=10000
```

```
## [1] 0.8061686
```

It appears that we get a superior result from the uniform distribution as our p(x).