# Homework 5

Shreya Rao sr3843

6/7/2021

## Part 1: Inverse Transform Method

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

$$f(x) = \frac{1}{\pi}\frac{1}{1-x^2}$$
$$F(x) = \int_{-\infty}^{x} \frac{1}{\pi}\frac{1}{1-x^2}\,dx = \frac{arctan(x) + \frac{\pi}{2}}{\pi}$$
$$u = \frac{arctan(x) + \frac{\pi}{2}}{\pi} \Rightarrow x = tan - \left(\pi.u - \frac{\pi}{2}\right)$$

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.

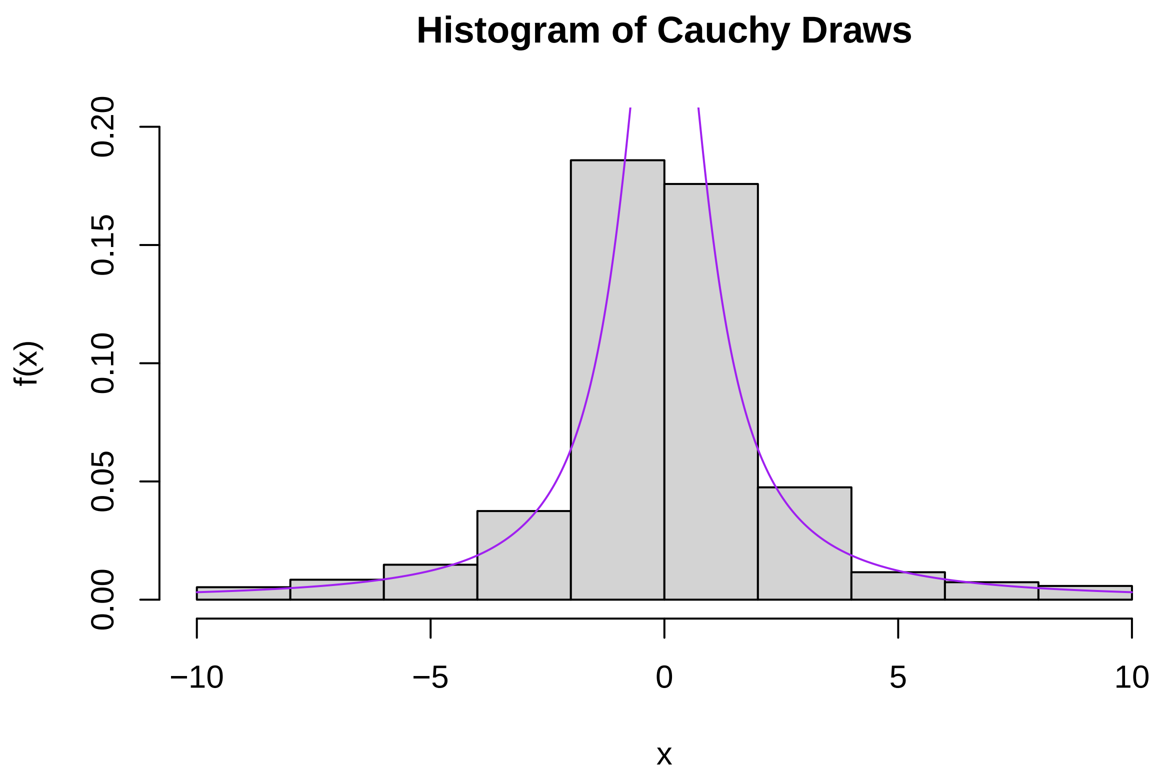
```
cauchy.sim <- function(n) {
    u <- runif(n)
    x <- tan(pi * u - (pi/2))
    return(x)
}
cauchy.sim(10)
```

```
##  [1]  1.43452007  1.84500355  0.05913753  0.95139660  2.63150959  9.07465540
##  [7] -0.72409924 -1.39995415  1.12513356 -0.30992732
```

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 fX(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.draws <- cauchy.sim(1000)

hist(cauchy.draws[cauchy.draws > -10 & cauchy.draws < 10], prob = TRUE,
    main = "Histogram of Cauchy Draws", ylab = "f(x)", xlab = "x",
    ylim = c(0, 0.2))
```

```r
y <- seq(-10, 10, by = 0.01)
cauchy.func <- function(x) {
    (1/pi) * (1/(1 + x^2))
}
lines(y, cauchy.func(y), col = "purple")
```

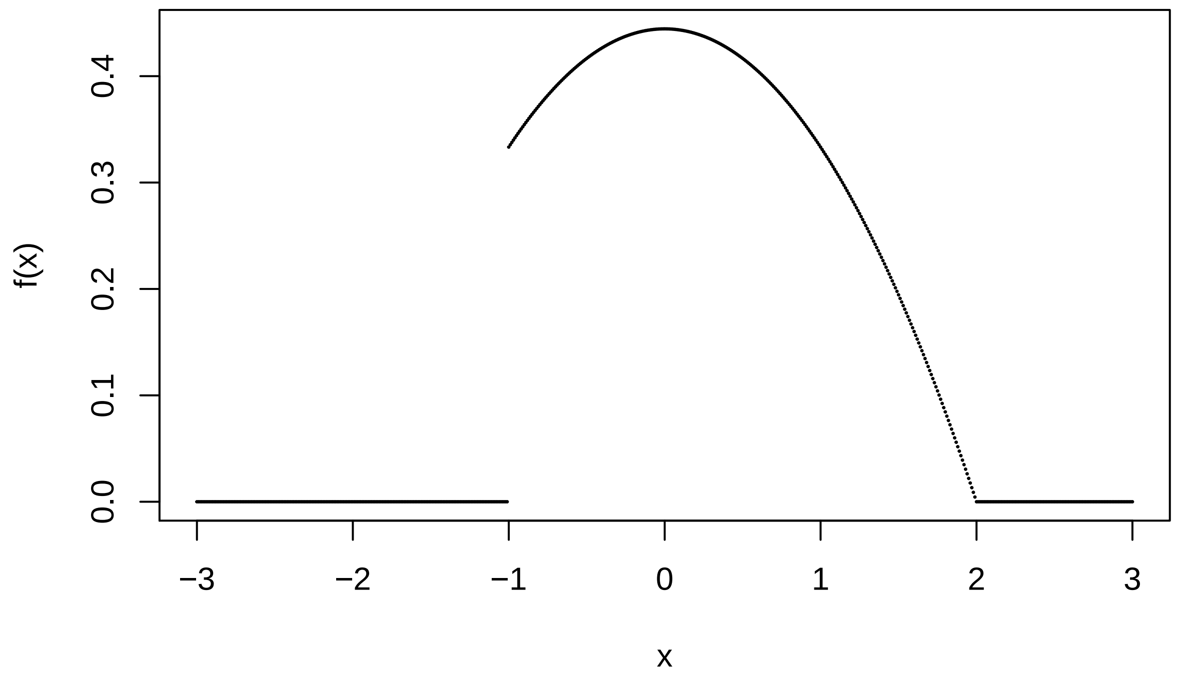


## Part 2: Accept-Reject Method

### 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. Make sure your plot is labeled appropriately

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

x <- seq(-3, 3, by = 0.01)
plot(x, f(x), cex = 0.1)
```



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.

$$f(x) = \frac{1}{9}\left(4 - x^2\right)$$
$$MaximumValue : f'(x) = 0 \Rightarrow x = 0$$

```
x.max = 0
f.max = f(0)
f.max
```

```
## [1] 0.4444444
```

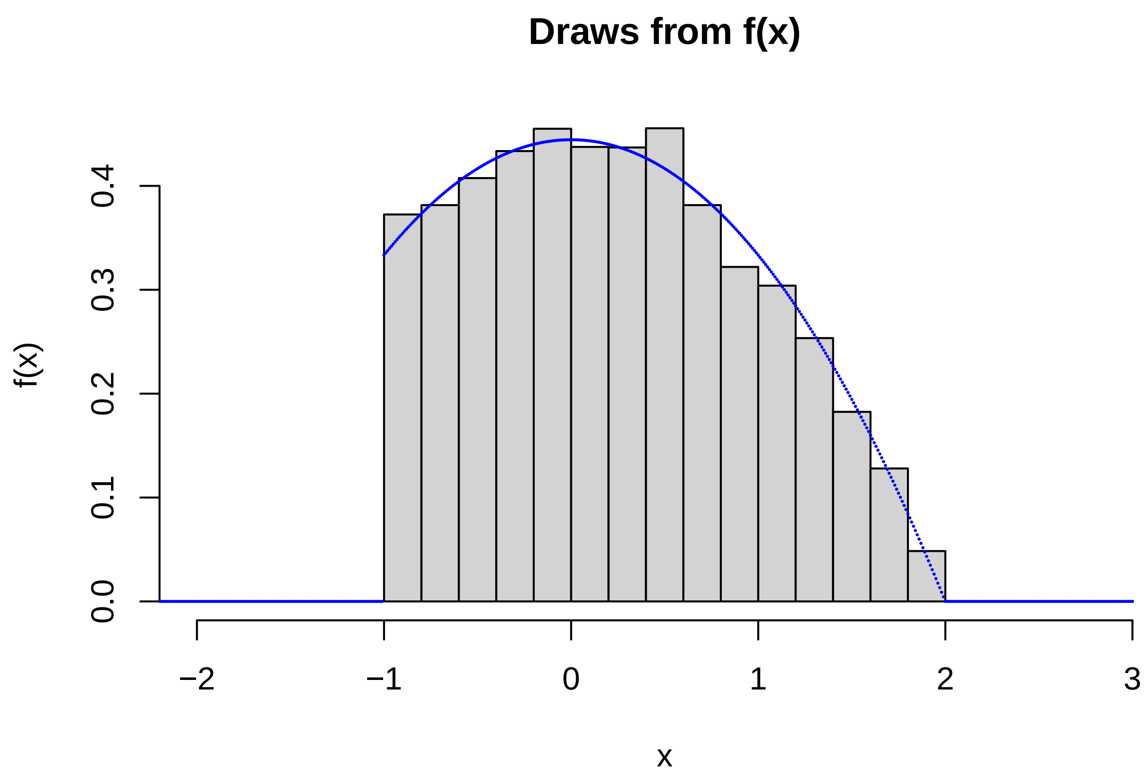
```
e <- function(x) {
    return(ifelse(x >= -1 & x <= 2, f.max, Inf))
}
```

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_samples <- 10000
n <- 0

f.draws <- numeric(n_samples)

while (n < n_samples) {
    # random draw from g
    y <- runif(1, min = -1, max = 2)
    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.

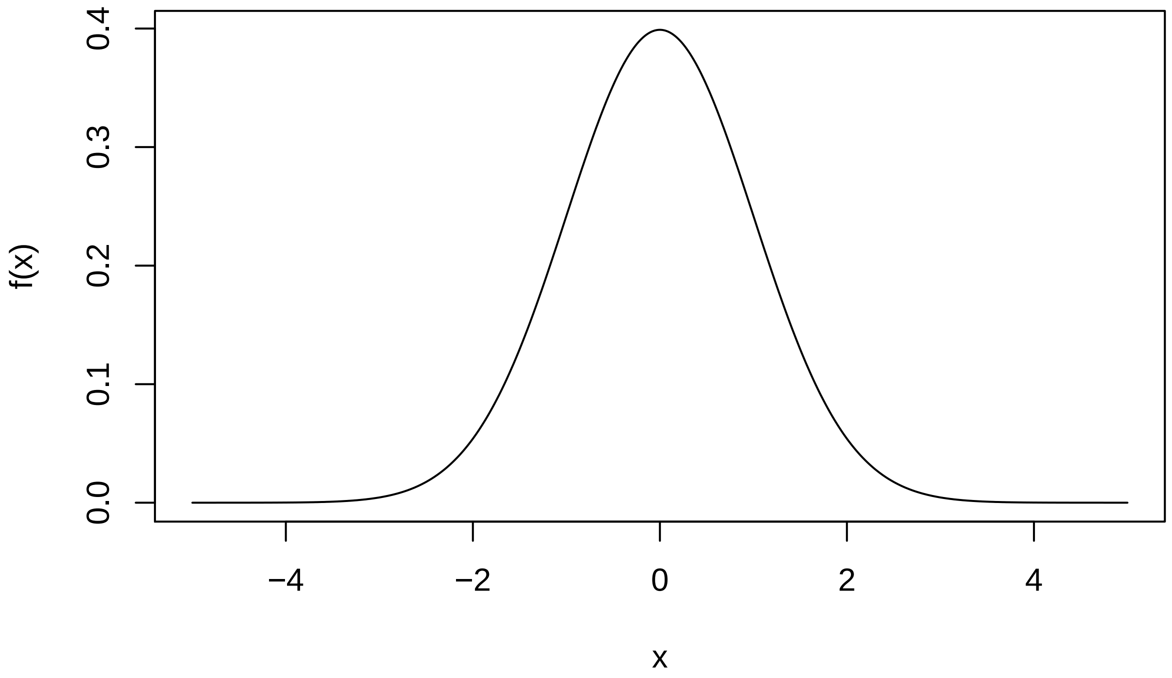
```
hist(f.draws, prob = T, ylab = "f(x)", xlab = "x", xlim = c(-2,
    3), main = "Draws from f(x)")
points(x, f(x), cex = 0.08, col = "blue")
```



**Problem 3: Accept-Reject 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. Make sure your plot is labeled appropriately.

```r
f <- function(x) {
    normal_f <- function(x) {
        (1/sqrt(2 * pi)) * exp(-(1/2) * x^2)
    }
    return(normal_f(x))
}

x <- seq(-5, 5, by = 0.01)
plot(x, f(x), type = "l")
```

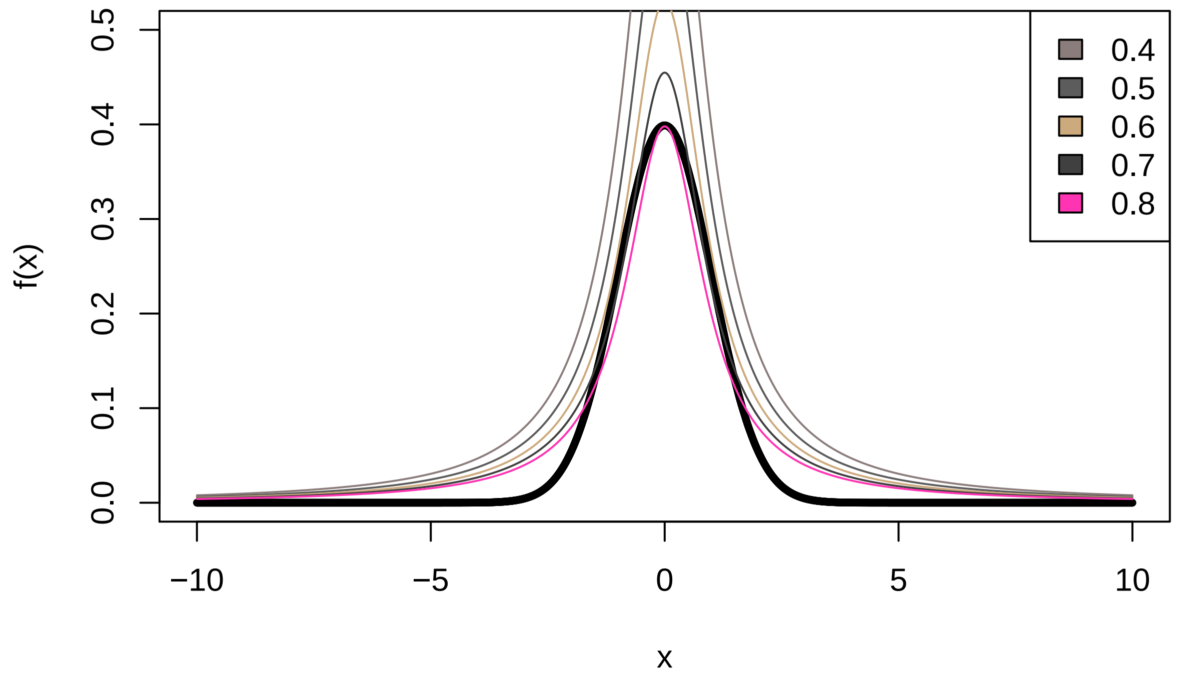


9. Let the known density g be the Cauchy density defined by pdf. Write a function e that takes as input a vector x and constant alpha (0 <   < 1) and returns a vector of e(x) values.

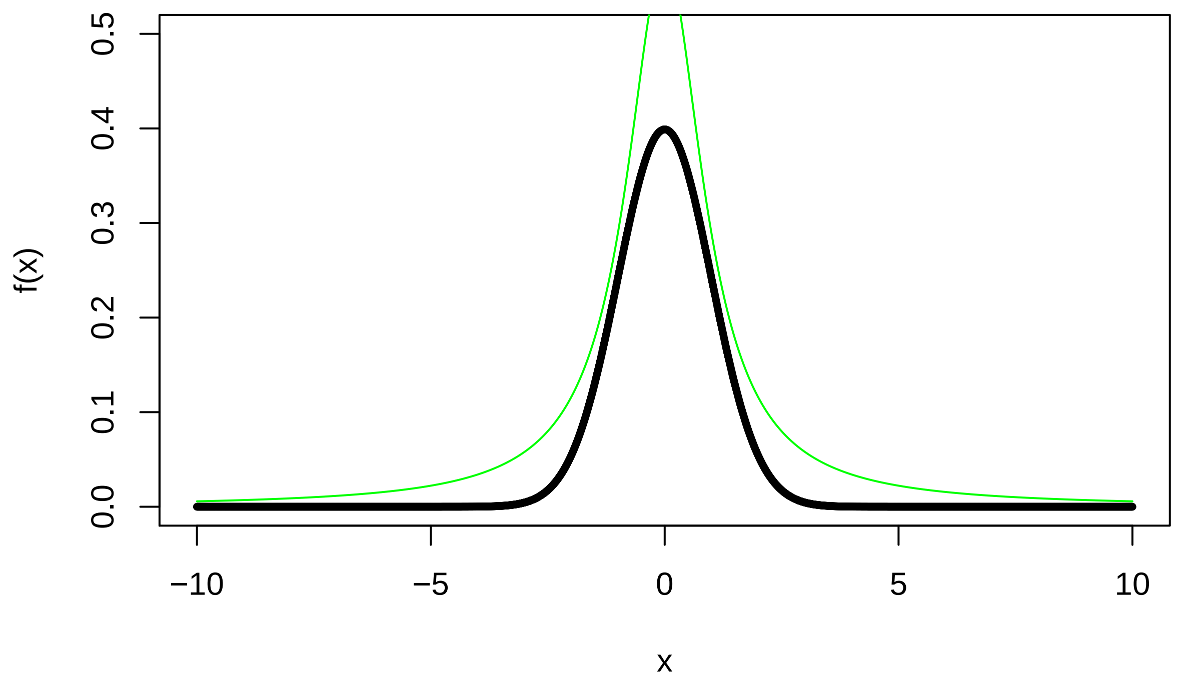
```r
e <- function(x, alpha) {
    cauchy_f <- function(x) {
        1/pi * (1/(1 + x^2))
    }
    # return(ifelse(x >= -5 & x <= 5, cauchy_f(x)/alpha, Inf))
    return(cauchy_f(x)/alpha)
}
```

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

```r
x <- seq(-10, 10, 0.01)
alphas <- seq(0.4, 0.8, by = 0.1)
plot(x, f(x), cex = 0.4, ylim = c(0, 0.5))
alpha_colors <- colors()[sample(2:657, length(alphas))]

for (i in 1:length(alphas)) {
    lines(x, e(x = x, alpha = alphas[i]), col = alpha_colors[i])
}
legend("topright", legend = alphas, fill = alpha_colors)
```



```r
plot(x, f(x), cex = 0.4, ylim = c(0, 0.5))
lines(x, e(x = x, alpha = 0.55), col = "green")
```



```r
all(e(x = x, alpha = 0.55) > f(x))
```

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

0.55 seems like a good alpha value.

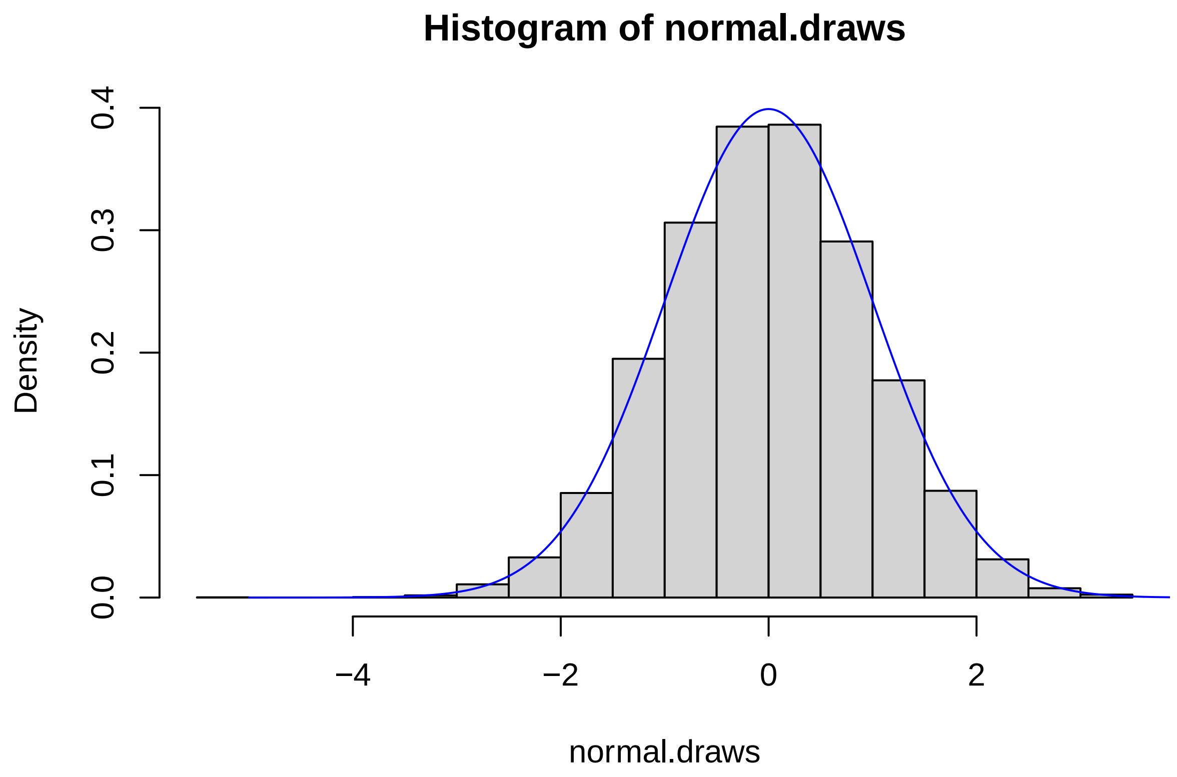
11. Write a function named normal.sim that simulates n standard normal random variables using the Accept-Reject Algorithm. The function should also use the InverseTransformation from Part 1. Test your function using n=10 draws

```r
normal.sim <- function(n) {
    n_count <- 0
    samples_normal <- numeric(n)
    a <- 0.55
    while (n_count < n) {
        # random draw from g
        y <- cauchy.sim(1)
        u <- runif(1)
        if (u < f(y)/e(x = y, alpha = a)) {
            n_count <- n_count + 1
            samples_normal[n_count] <- y
        }
    }
    return(samples_normal)
}

normal.sim(10)
```

```
##  [1] -1.2804255 -1.1416261  0.1206478 -0.6076093 -1.2407055  1.2158861
##  [7]  0.2197108  0.9662568 -0.2145704  1.1158182
```

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)
x <- seq(-5, 5, by = 0.01)
lines(x, f(x), col = "blue")
```



## Part 3: Simulation with Built-in R Functions

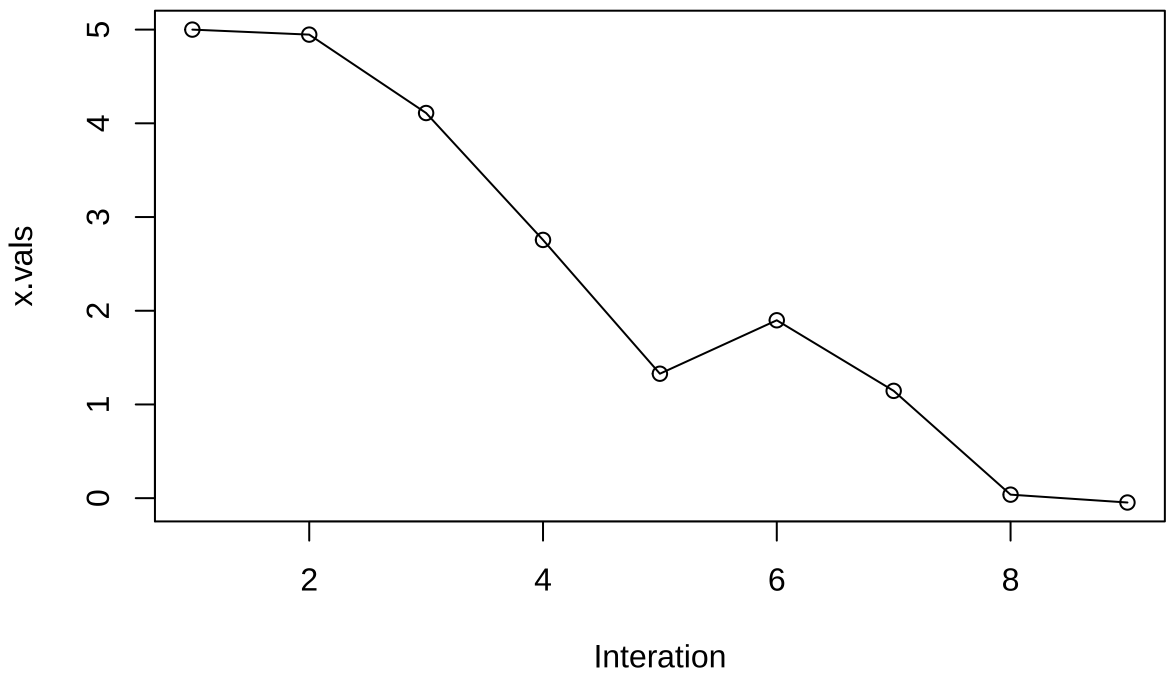
13. Write a while() loop to implement this procedure. Importantly, save all the positive values of x that were visited in this procedure in a vector called x.vals, and display its entries.

```r
x.vals <- NULL
x.vals[1] <- 5
c <- 1
while (x.vals[c] > 0) {
    c = c + 1
    x.vals[c] <- x.vals[c - 1] + runif(1, min = -2, max = 1)
}

x.vals
```

```
## [1]  5.00000000  4.94656201  4.11001060  2.75581959  1.32797064  1.89837003
## [7]  1.14497915  0.03748566 -0.04625892
```

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.

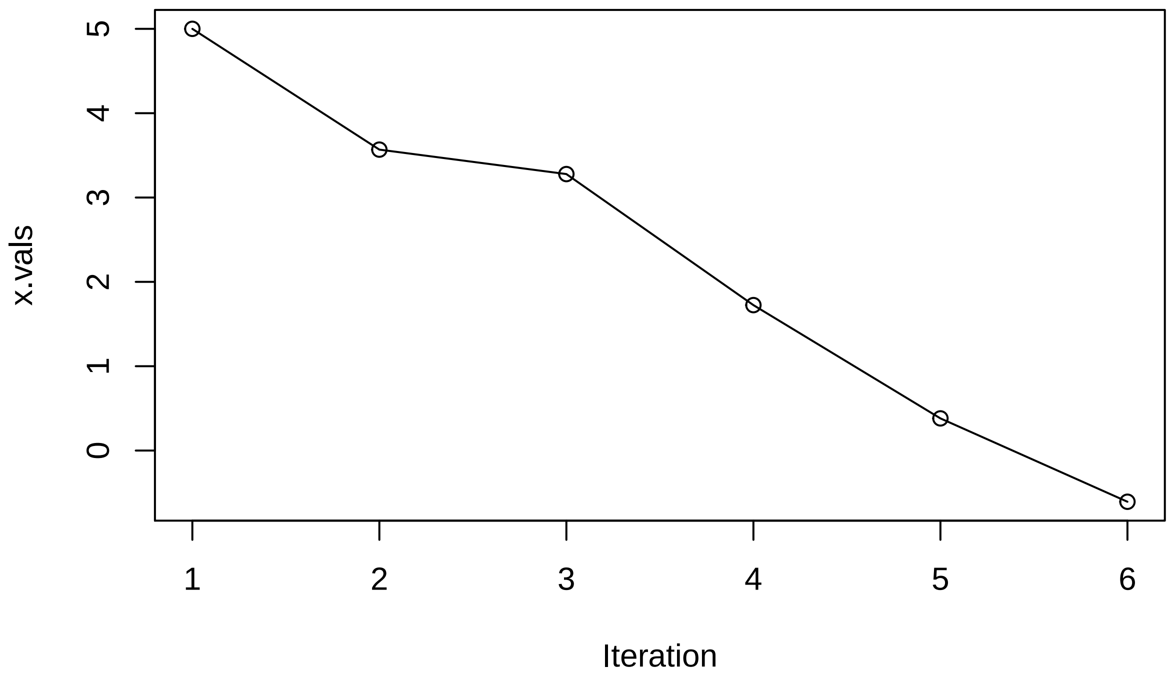
```r
plot(1:length(x.vals), x.vals, type = "o", xlab = "Interation")
```



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

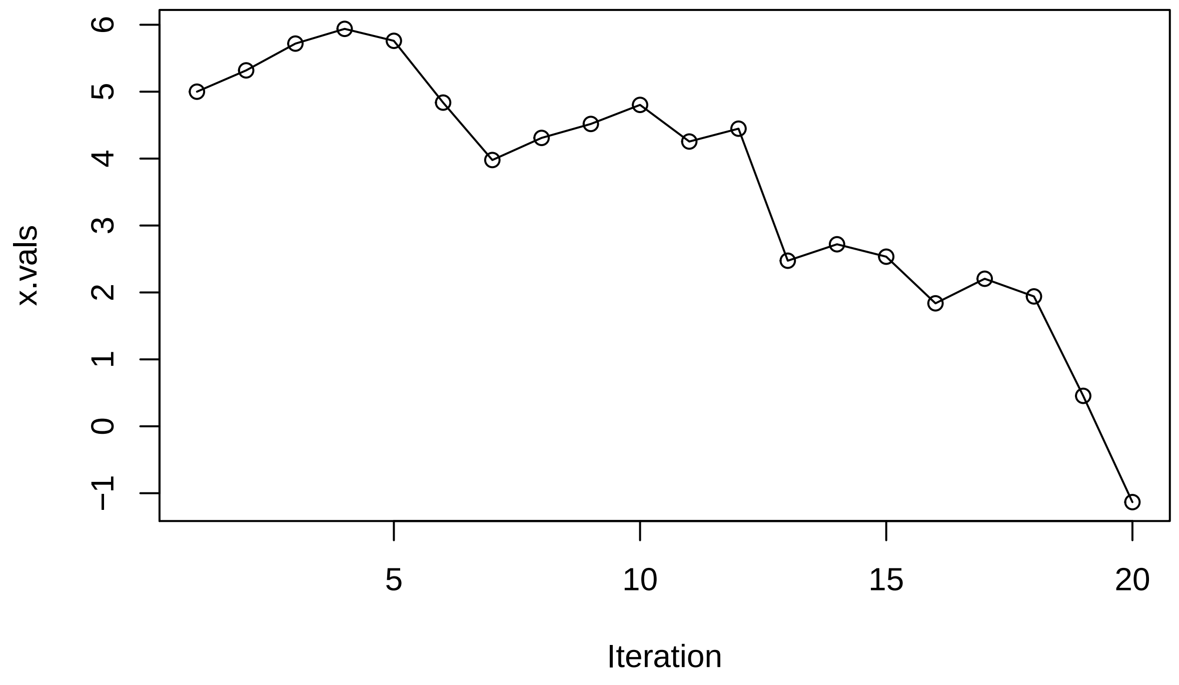
vector of the random walk values as computed above; and num.steps, the number of steps taken by the random walk before terminating. Run your function twice with the default inputs, and then twice times with x.start equal to 10 and plot.walk = FALSE.

```r
x.start = 5
plot.walk = TRUE

random.walk <- function(x.start = 5, plot.walk = TRUE) {
    x.vals <- NULL
    x.vals[1] <- x.start
    c <- 1

    while (x.vals[c] > 0) {
        c = c + 1
        x.vals[c] <- x.vals[c - 1] + runif(1, min = -2, max = 1)
    }

    num_steps <- c

    if (plot.walk) {
        plot(1:num_steps, x.vals, type = "o", xlab = "Iteration")
    }

    return(list(Iterations = num_steps, RandomWalk.Value = x.vals))
}

random.walk()
```



```
## $Iterations
## [1] 6
##
## $RandomWalk.Value
## [1]  5.0000000  3.5679822  3.2780522  1.7243894  0.3812190 -0.6070728
```

```
random.walk()
```



```
## $Iterations
## [1] 20
##
## $RandomWalk.Value
##  [1]  5.0000000  5.3182096  5.7185024  5.9384405  5.7599500  4.8368816
##  [7]  3.9785967  4.3091366  4.5176035  4.8026705  4.2549931  4.4471250
## [13]  2.4741220  2.7198324  2.5343971  1.8374984  2.2047375  1.9404231
## [19]  0.4549444 -1.1329324
```

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

```
## $Iterations
## [1] 21
##
## $RandomWalk.Value
##  [1] 10.00000000  9.76111200 10.45393318 10.45595344  9.73926306  7.93428487
##  [7]  6.60914242  5.55040856  6.05698037  5.90542088  6.25006744  6.55223395
## [13]  5.70114559  5.90320054  6.80228943  6.27804023  4.36134233  2.63209455
## [19]  1.62243386  0.02029864 -1.63999855
```

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

```
## $Iterations
## [1] 27
```

```
##
## $RandomWalk.Value
##  [1] 10.00000000 10.18599719  9.35907558  7.72179760  6.86647975  5.16801713
##  [7]  5.04966017  3.68644339  3.18277106  3.61973740  1.65396287  2.61635082
## [13]  2.18564942  1.72879613  2.50334751  2.94084153  3.47157611  3.02827340
## [19]  1.17706278  2.06487842  2.35051197  2.09439617  0.74882594  0.08757258
## [25]  0.15332139  0.63832951 -0.26846976
```

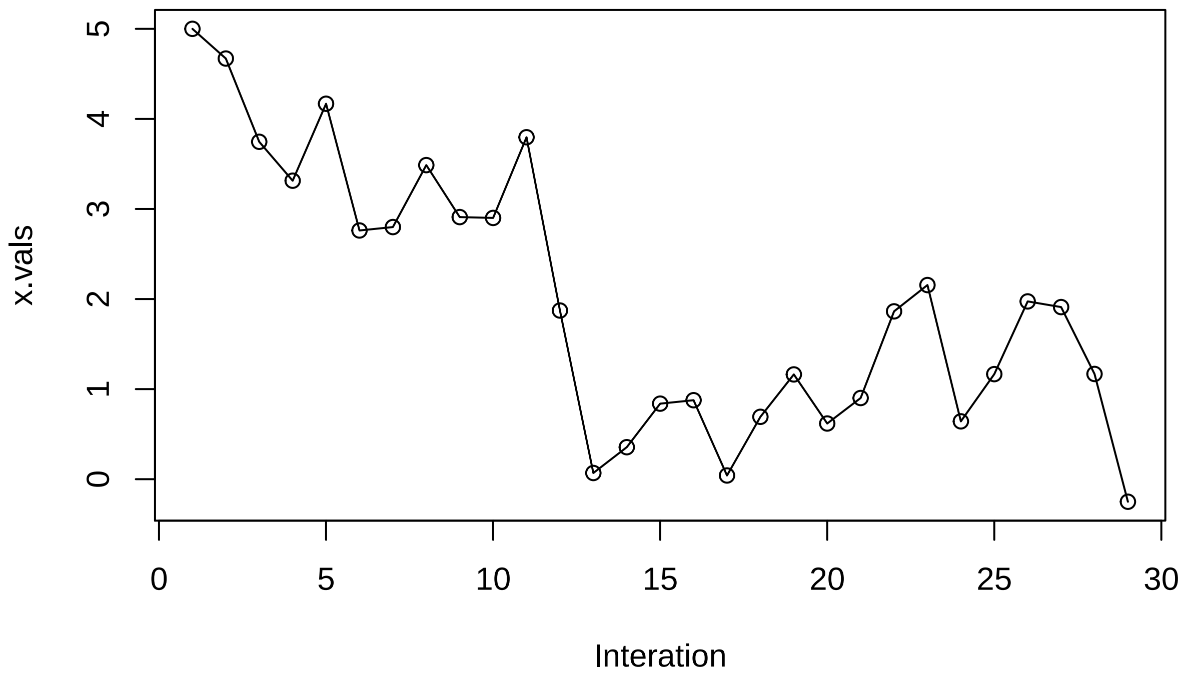
16. We'd like to answer the following question using simulation: if we start our random walk process, as defined above, at x = 5, what is the expected number of iterations we need until it terminates? To estimate the solution produce 10,000 such random walks and calculate the average number of iterations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

```
exp_iter <- rep(NA, 10000)
for (i in 1:10000) {
    temp_rw <- random.walk(x.start = 5, plot.walk = FALSE)
    exp_iter[i] <- temp_rw$Iterations
}

# Expected number of iterations
mean(exp_iter)
```

```
## [1] 12.2995
```

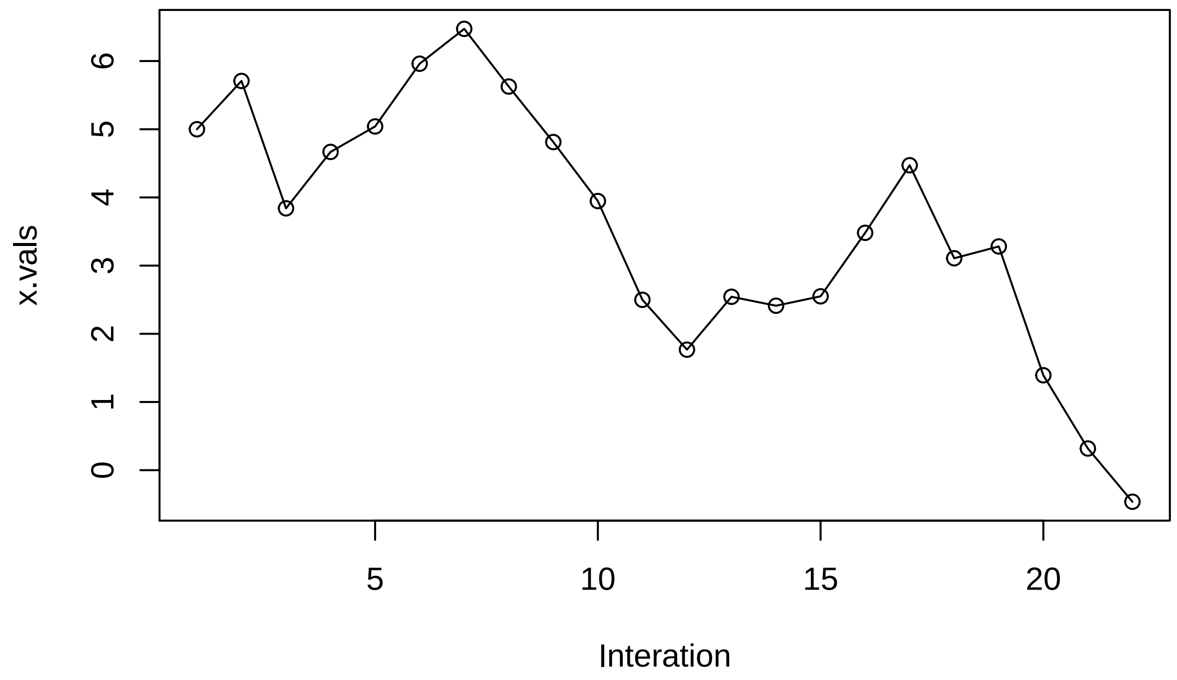
17. Modify your function random.walk() defined previously so that it takes an additional argument seed: this is an integer that should be used to set the seed of the random number generator, before the random walk begins, with set.seed(). But, if seed is NULL, the default, then no seed should be set. 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 <- function(x.start = 5, plot.walk = TRUE, seed = NULL) {
    set.seed(seed)

    x.vals <- NULL
    x.vals[1] <- x.start
    c <- 1

    while (x.vals[c] > 0) {
        c = c + 1
        x.vals[c] <- x.vals[c - 1] + runif(1, min = -2, max = 1)
    }

    num_steps <- c

    if (plot.walk) {
        plot(1:num_steps, x.vals, type = "o", xlab = "Interation")
    }

    return(list(Iterations = num_steps, RandomWalk.Value = x.vals))
}

random.walk()
```



```
## $Iterations
## [1] 29
##
## $RandomWalk.Value
##  [1]  5.00000000  4.67026975  3.74601559  3.31401580  4.16847682  2.76130512
##  [7]  2.79928347  3.48713794  2.91006837  2.90067697  3.79656171  1.87252413
## [13]  0.06859207  0.35523762  0.83859934  0.87696425  0.04104363  0.69254395
## [19]  1.16313682  0.61809148  0.90109078  1.86332631  2.15536431  0.64212667
## [25]  1.16648437  1.97393261  1.91075497  1.16826409 -0.25048962
```

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



```
## $Iterations
## [1] 22
##
## $RandomWalk.Value
##  [1]  5.0000000  5.7081327  3.8401446  4.6674193  5.0408268  5.9607847
##  [7]  6.4721805  5.6262172  4.8122613  3.9465726  2.4984692  1.7674660
## [13]  2.5433559  2.4118695  2.5498082  3.4806938  4.4719870  3.1078809
## [19]  3.2811381  1.3916787  0.3179442 -0.4628840
```

```
random.walk(plot.walk = FALSE, seed = 33)
```

```
## $Iterations
## [1] 11
##
## $RandomWalk.Value
##  [1]  5.0000000  4.3378214  3.5217724  2.9729590  3.7295869  4.2612312
##  [7]  3.8132800  3.1246550  2.1542497  0.2008006 -1.4452259
```

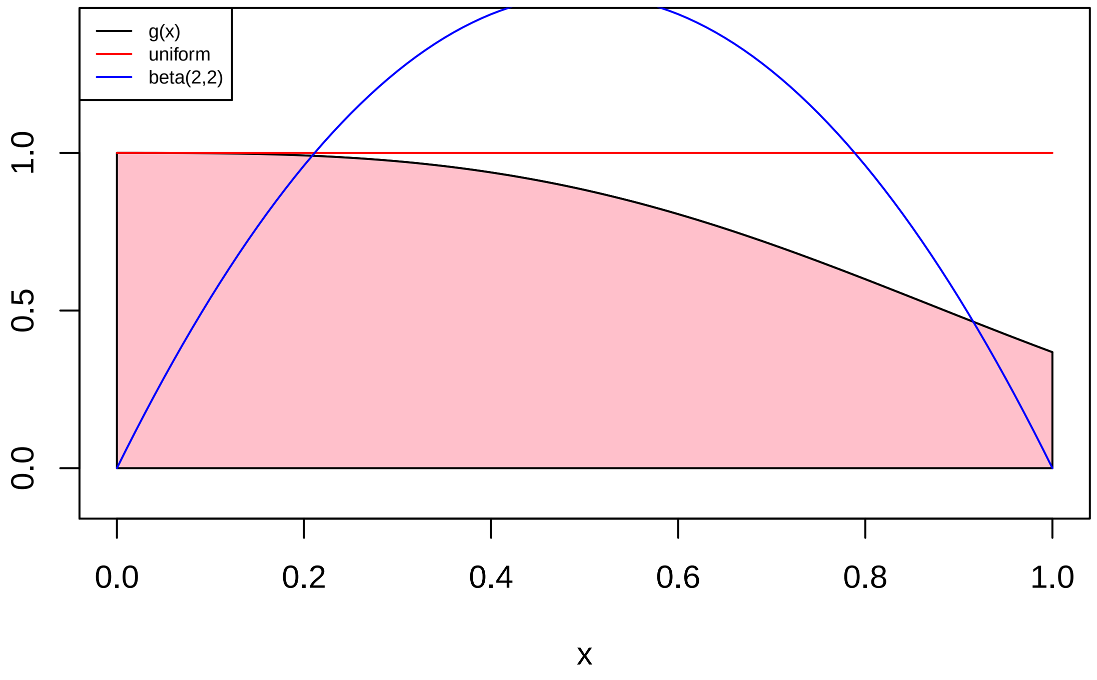
```
random.walk(plot.walk = FALSE, seed = 33)
```

```
## $Iterations
## [1] 11
##
## $RandomWalk.Value
##  [1]  5.0000000  4.3378214  3.5217724  2.9729590  3.7295869  4.2612312
##  [7]  3.8132800  3.1246550  2.1542497  0.2008006 -1.4452259
```

## Part 4: Monte Carlo Integration

18. Run the following code:

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

x <- seq(0, 1, 0.01)
alpha <- 2
beta <- 2
plot(x, g(x), type = "l", xlab = "x", ylab = "", ylim = c(-0.1,
    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 = 0.6)
```



19. Using Monte Carlo Integration, approximate the integral with n = 1000^2 random draws from the distribution uniform(0,1).

```r
n <- 1000^2
g.x_over_unif <- function(x) {
    return(exp(-x^3))
}

mean(g.x_over_unif(runif(n, min = 0, max = 1)))
```

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

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

#Estimate

$$\int g(x)\,dx$$

by taking the sample mean of

$$\frac{g(x)}{p(x)}$$

where

$$p(x) = 6x(x-1)$$

and

$$\frac{g(x)}{p(x)} = \frac{e^{-x^3}}{6x(1-x)}$$

```r
# g.x_over_beta <- function(x) {
# return(exp(-x^3)/(6*x*(1-x))) } mean(g.x_over_beta(rbeta(n,
# shape1 = 2, shape2 = 2)))

tgx <- function(x) {
    return(exp(-x^3)/dbeta(x, shape1 = 2, shape2 = 2))
}

mean(tgx(rbeta(n, shape1 = 2, shape2 = 2)))
```

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