PSTAT 194CS HW 2
Random Number Generation and Sampling

Ostapenko, Vasiliy (vostapenko, 774 970 8)
Collaborated with: N/A

Contents

Exercise 1

(a) For each of the following commands, either explain why they should be errors, or explain the non-erroneous result.

```
vector1 = c("8", "12", "7", "32")
min(vector1)
sort(vector1)
sum(vector1)
```

The first two commands execute without error, while the third command throws an invalid argument error. However, conceptually, the results of the first two commands are wrong. Because the vector has character strings instead of a float or integer, the min() and sort() functions use lexicographic ordering, character by character. Thus the minimum element of "12" is a minimum because it is the only one which starts with a 1. Also, the sorted order is such that "12" comes first, due to the aforementioned reason, and "32" comes next because it starts with a 3.

(b) For the next series of commands, either explain their results, or why they should produce errors.

```
vector2 <- c("5",7,12)
vector2[2] + vector2[3]
```

The attempt to add the second and third elements of vector2 causes an error. This is because R sees that we passed both strings and integers into a vector, and coerces the integers into strings. Thus there is an error - addition operator is not defined for strings.

```
dataframe3 <- data.frame(z1="5",z2=5,z3=12)
dataframe3[1,2] + dataframe3[1,3]
```

On the other hand, here there is no error when adding elements of a data.frame. This is because the data.frame constructor does not coerce elements of different columns to be the same type. Thus the second two columns, elements of which are added, stay as numeric, and the addition operator is defined.

```
list4 <- list(z1="6", z2=42, z3="49", z4=126)
list4[[2]]+list4[[4]]
list4[2]+list4[4]
```

The first addition attempt works because double brackets access the actual numeric element of a named list. The second addition attempt does not work because using single brackets produces a smaller named list with just that element. Addition is not defined for list objects and thus the attempt fails.

Exercise 2

Working with functions and operators. (a) The colon operator will create a sequence of integers in order. It is a special case of the function `seq()`. Using the help command `?seq` to learn about the function, design an expression that will give you the sequence of numbers from 1 to 20000 in increments of 582. Design another that will give you a sequence between 1 and 20000 that is exactly 52 numbers in length.

```
print(seq(1, 20000, 582))
```

```
##  [1]     1   583  1165  1747  2329  2911  3493  4075  4657  5239  5821  6403
## [13]  6985  7567  8149  8731  9313  9895 10477 11059 11641 12223 12805 13387
## [25] 13969 14551 15133 15715 16297 16879 17461 18043 18625 19207 19789
```

```
print(seq(1, 20000, length.out=52))
```

```
##  [1]     1.0   393.1   785.3  1177.4  1569.5  1961.7  2353.8  2746.0  3138.1
## [10]  3530.2  3922.4  4314.5  4706.6  5098.8  5490.9  5883.1  6275.2  6667.3
## [19]  7059.5  7451.6  7843.7  8235.9  8628.0  9020.2  9412.3  9804.4 10196.6
## [28] 10588.7 10980.8 11373.0 11765.1 12157.3 12549.4 12941.5 13333.7 13725.8
## [37] 14117.9 14510.1 14902.2 15294.4 15686.5 16078.6 16470.8 16862.9 17255.0
## [46] 17647.2 18039.3 18431.5 18823.6 19215.7 19607.9 20000.0
```

(b) The function `rep()` repeats a vector some number of times. Explain the difference between `rep(1:5, times=6)` and `rep(1:5, each=6)`.

```
print(rep(1:5, times=6))
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```
print(rep(1:5, each=6))
```

```
##  [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 5
```

Using the times argument, the sequence is repeated 6 times in the specific order we passed it in. Using the each argument, each of the elements is repeated 6 times one after another.

Exercise 3

Writing R functions. Write an `R` function that accepts an $n \times n$ matrix A as an argument and returns the trace of A, $trace(A) = \sum_{i=1}^{n} a_{ii}$. Include code that verifies that the function's argument is indeed a square matrix. Also include comments that describe the function's purpose, argument and output. The following code generates a list of 25 matrices of different dimensions. Apply your trace function to each matrix in this list.

```
matrix.list <- lapply(1:25, FUN=function(x) {matrix(1:x^2, nrow=x, ncol=x)})
trace_func = function(A) {
  # Function which computes the trace of A and returns it,
  # after checking that A is a square matrix
  # trace_func accepts matrix A and outputs numeric type sigma
  dims = dim(A)
```

```
  sigma = 0
  if(dims[[1]] != dims[[2]]) {
    return(-1)
  } else {
    for(i in 1:dims[[1]]) {
      sigma = sigma + A[i, i]
    }
    return(sigma)
  }
}
```

```
print(sapply(matrix.list, trace_func))
```

```
##  [1]    1    5   15   34   65  111  175  260  369  505  671  870 1105 1379 1695
## [16] 2056 2465 2925 3439 4010 4641 5335 6095 6924 7825
```

Exercise 4

[**Question**] Which method is better? Answer this by responding to the following parts. **(a)** Write a program that simulates 10,000 Beta(3,4) random variables using both methods.

```
x = list()
i=0
while(length(x) != 10000) {
  y = runif(n=1, min=0, max=1)
  u = runif(n=1, min=0, max=1)
  if(u < ((y^2 * (1-y)^3)/(0.0346*1))) {
    x = append(x, y)
  }
  i = i+1
}
x = unlist(x)
print(length(x)/i*100)
```

```
## [1] 48.28
```

```
print(1/0.0346)
```

```
## [1] 28.9
```

```
z = list()
i=0
while(length(z) != 10000) {
  y = mean(runif(n=2, min=0, max=1))
  u = runif(n=1, min=0, max=1)
  if(u < ((y^2 * (1-y)^3)/(0.0264*(2-abs(4*y-2))))) {
    z = append(z, y)
  }
  i = i+1
}
z = unlist(z)
print(length(z)/i*100)
```

```
## [1] 63.07
```

```
print(1/0.0264)
```

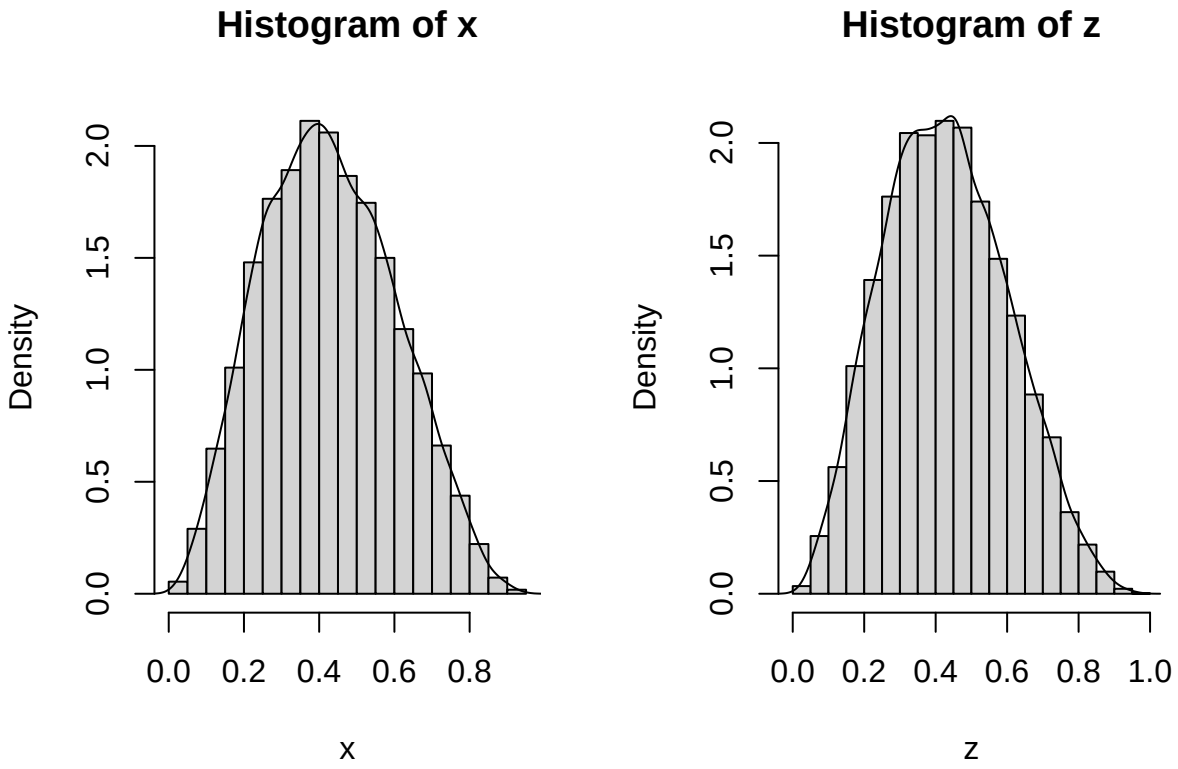
```
## [1] 37.88
```

(b) Compare the acceptance rates and decide which one is the better algorithm. Acceptance rate of the second algorithm is about 64% versus that of the first algorithm, which is about 49%. The second algorithm is better using the acceptance rate criterion.

(c) Is it correct to use theoretical acceptance rate $\frac{1}{M}$? Why or why not? The theoretical acceptance rate is a rule of thumb, or an approximation. However, here, it gives poor results. Using 1/M gives 28.9% and 37.88% acceptance rates respectively. What we have experimentally is 48.64% and 63.98% respectively.

(d) Graph histograms of your results.

```
par(mfrow=c(1, 2))
{
  {
    hist(x, freq=FALSE)
    lines(density(x))
  }
  {
    hist(z, freq=FALSE)
    lines(density(z))
  }
}
```



(e) Summarize your findings. We are able to generate samples from the kernel of a Beta(3, 4) distribution by using uniform random variables. Experimentally, the second algorithm yields better results due to higher acceptance. The histograms of both sets of random variables are very similar to what Beta(3, 4) random variables might look like.

Exercise 5

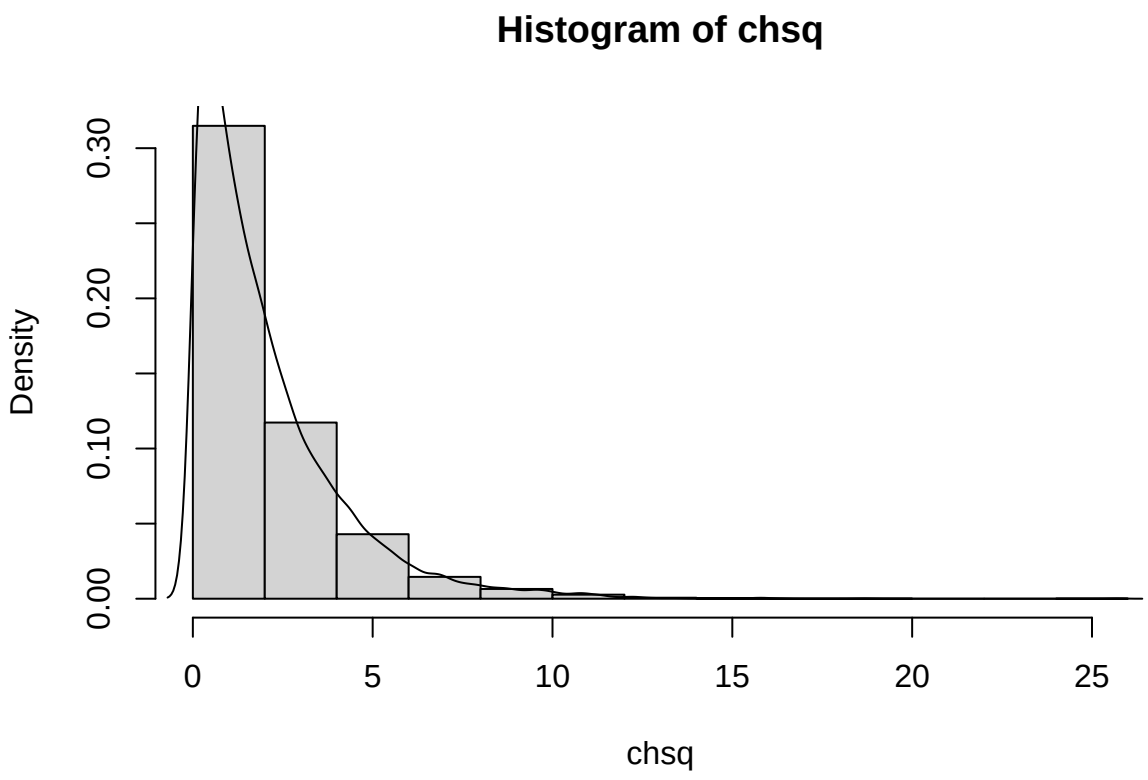
Simulating a mixture distribution.

```
n = 1000
r = 4
b = 2
lambda = rgamma(n=n, shape=r, rate=b)
x = rexp(n=n, rate=lambda)
```

Exercise 6

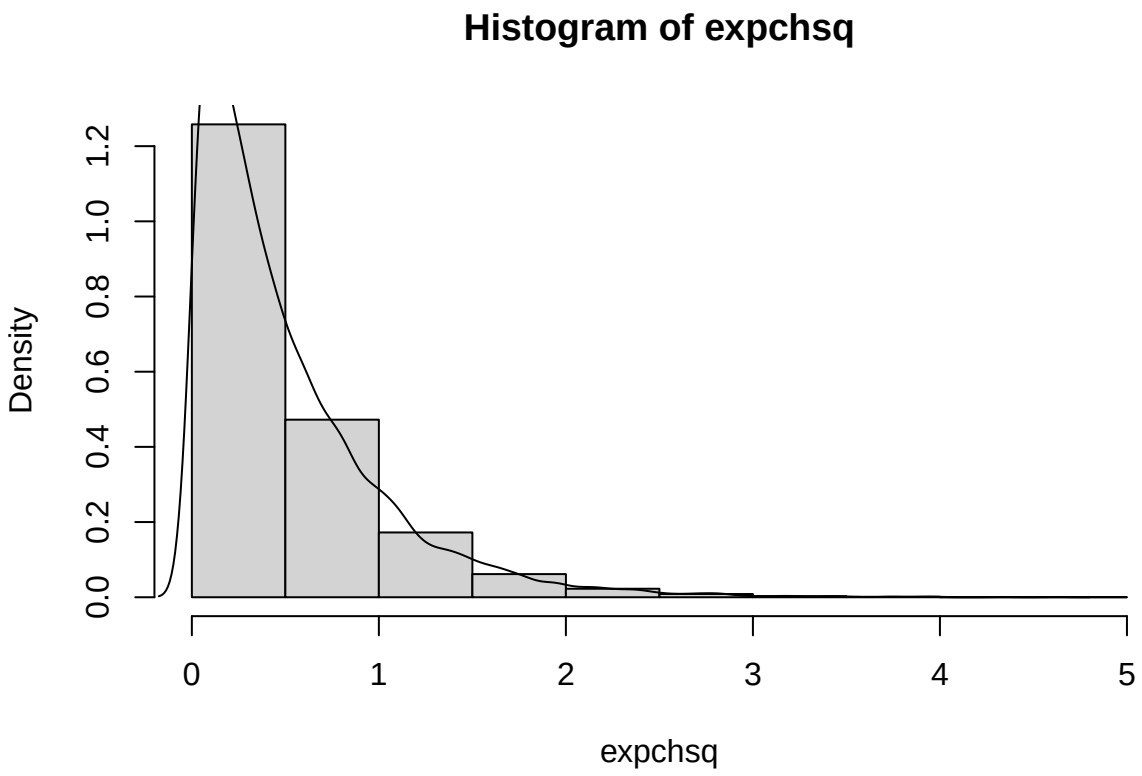
(a) Simulate 10000 pairs of random variables (Z_1, Z_2), both from $\mathcal{N}(0, 1)$ using the transformation method 4 under "3.4: Transformation methods" in the book. Use these pairs to generate 10000 $\chi^2(2)$ random variables.

```
z1 = rnorm(n=10000, mean=0, sd=1)
z2 = rnorm(n=10000, mean=0, sd=1)
chsq = z1^2 + z2^2
hist(chsq, freq=FALSE)
lines(density(chsq))
```



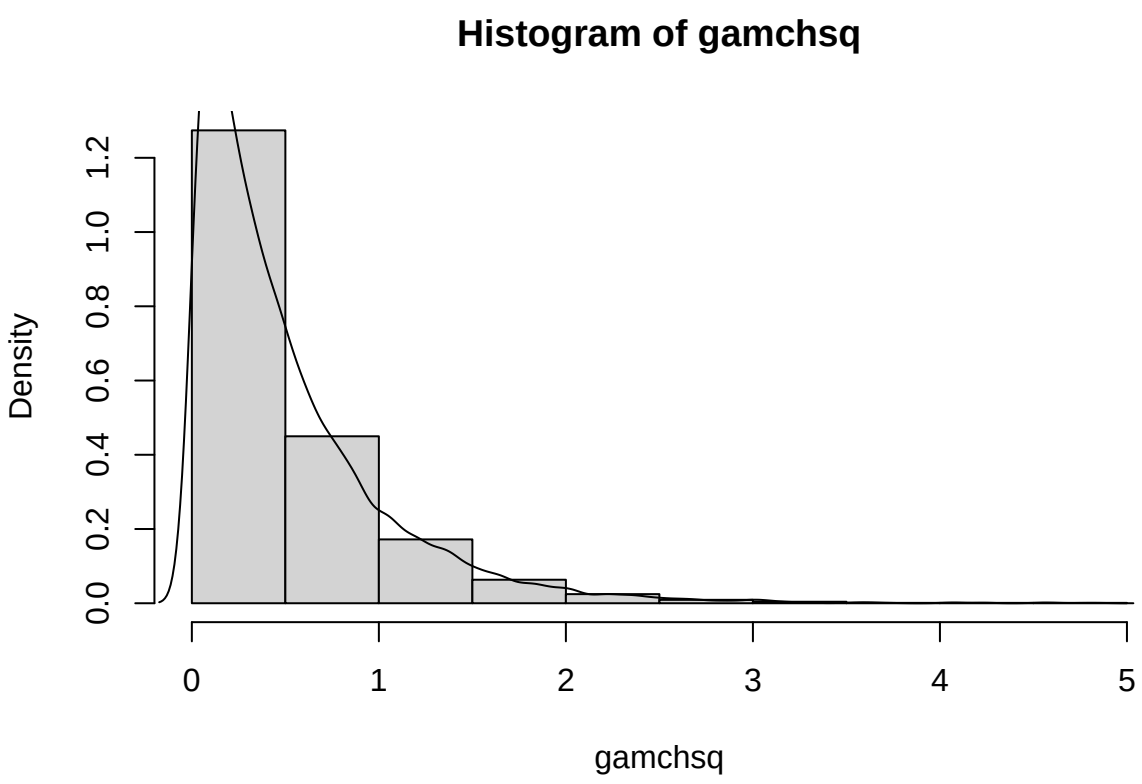
(b) How does the exponential distribution relate to a $\chi^2(2)$ distribution? (Do a little research!) Generate 10000 random variables from a useful exponential distribution using CDF inversion for this case. Use them to calculate 10000 random variables from a $\chi^2(2)$ distribution.

```
expchsq = rexp(n=10000, rate=2)
hist(expchsq, freq=FALSE)
lines(density(expchsq))
```



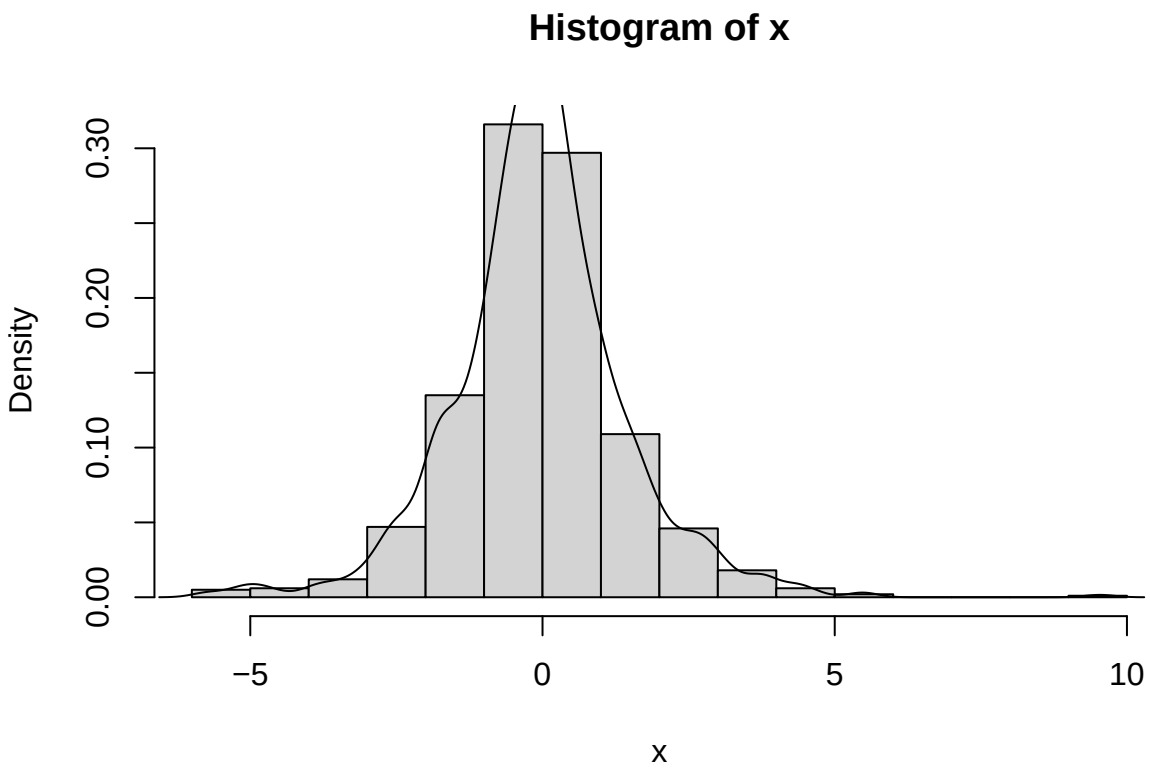
(c) Generate 10000 random variables from a gamma distribution such that the resulting variables will be $\chi^2(2)$. (You can use an r function to generate the gamma variables.)

```
gamchsq = rgamma(n=10000, shape=1, rate=2)
hist(gamchsq, freq=FALSE)
lines(density(gamchsq))
```



For Exerise 7-10, do the following problems from the book 3.2, 3.3, 3.5, 3.9. ## Exerise 7 (3.2) f(x) = .5exp(-abs(x)) u ~ Unif(0, 1) x = invF(u) = -log(2u-1) if .5 <= u < 1; log(2u) if 0 < u < .5.

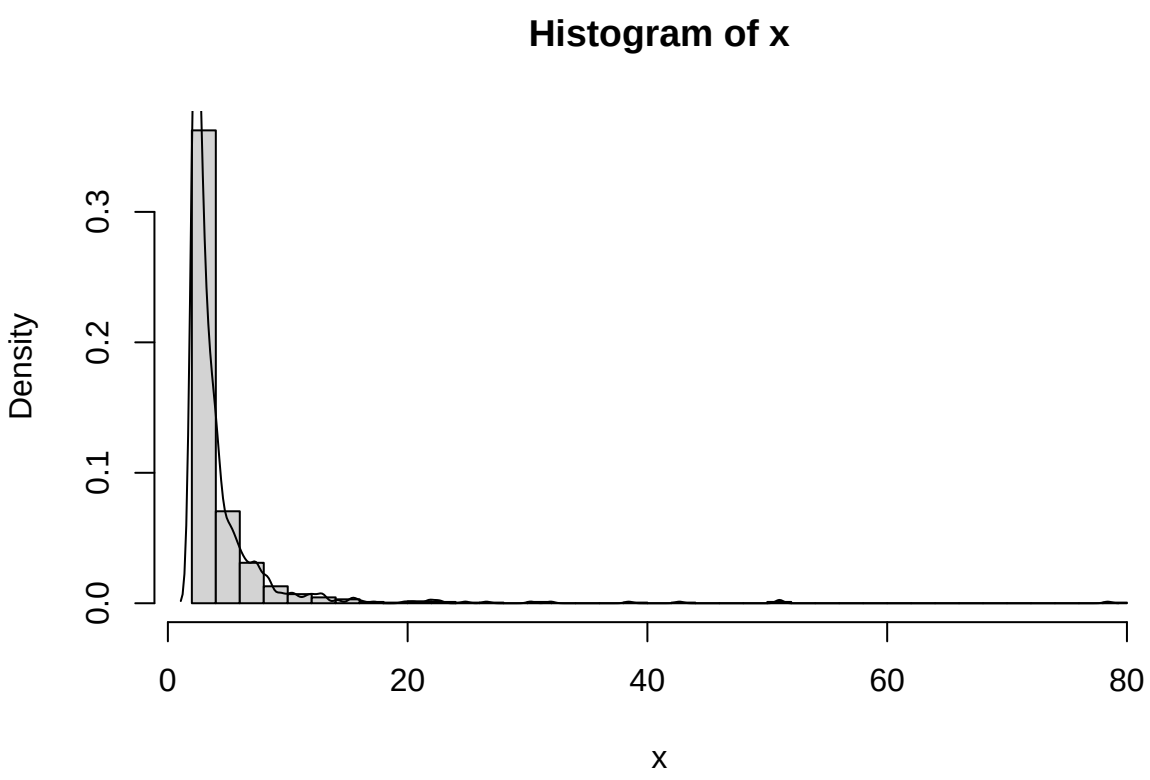
```
u = runif(n=1000, min=0, max=1)
x = c(log(2*u[u < .5]), -log(2*u[u >= .5]-1))
hist(x, freq=FALSE)
lines(density(x))
```



Exerise 8 (3.3)

F(x) = 1 - (b/x)^a u ~ Unif(0, 1) x = invF(u) = b(1-u)^(-1/a) = b(u)^(-1/a)

```
u = runif(n=1000, min=0, max=1)
x = 2*(u)^(-1/2)
hist(x, breaks="Scott", prob=TRUE)
lines(density(x))
```



Exerise 9 (3.5)

X ~ p(x) = 0.1 if x=0; 0.2 if x=1; 0.2 if x=2; 0.2 if x=3; 0.3 if x=4.

```
x = c(0, 1, 2, 3, 4)
p = c(0.1, 0.2, 0.2, 0.2, 0.3)
z = sample(x=x, size=1000, replace=TRUE, prob=p)
print(rbind(table(z)/1000, p))
```

```
##       0     1     2     3     4
##   0.102 0.216 0.207 0.181 0.294
## p 0.100 0.200 0.200 0.200 0.300
```

Exercise 10 (3.9)

f(x) = .75(1-x^2) if abs(x) <= 1

```
epanechnikov = function(n) {
  u1 = runif(n, -1, 1)
  u2 = runif(n, -1, 1)
  u3 = runif(n, -1, 1)
  mask = (abs(u3) >= abs(u2)) & (abs(u3) >= abs(u1))
  return(ifelse(mask, u2, u3))
}
```

```
z = epanechnikov(10000)
hist(z, breaks="Scott", prob=TRUE)
```

```
x = seq(-1, 1, 0.001)
fx = 0.75*(1-x^2)
lines(x, fx)
```