ST303 Coursework 1

Candidate Number: 26617

Question 1

Method

Immediately, 2 solutions come to mind. First is generating **Y ~ Exp(1)** then approximating **E[tanh(Y)]** by calculating the sample mean. However, there are more accurate method which I have implemented instead.
The first step is to transform the integral using $\mathbf{x = e^{-y}}$ to obtain:

$$\int_0^\infty e^{-y} tanh(y) dy = \int_0^1 tanh(-log(x)) = \mathbb{E}[tanh(-logX)]$$

where **X ~ Uniform(0,1).**

The idea here is to treat the integrand as a function of a standard uniform random variable therefore the integral gives you the expectation of this function. Using this, we can then apply different variance reduction techniques to improve the accuracy. The most obvious would be to use an antithetic variable, **X2 = 1 - X ~ Uniform(0,1)**, which is very easy to do and doesn't require much computational power. This reduces the variance since **X** and **X2** have negative covariance hence reducing the variance of the estimation significantly.

Such antithetic variables should only be applied when the distribution is symmetric and when we are estimating a singular value only, for example an expectation or integral. The table below displays the results I obtained.

Confidence Intervals

The central limit theorem suggests that the distribution of the sample mean converges to a normal as the sample size increases. For any random variable X,

$$\overline{X} \sim \mathcal{N}(\mathbb{E}(X), \mathrm{Var(X)})$$

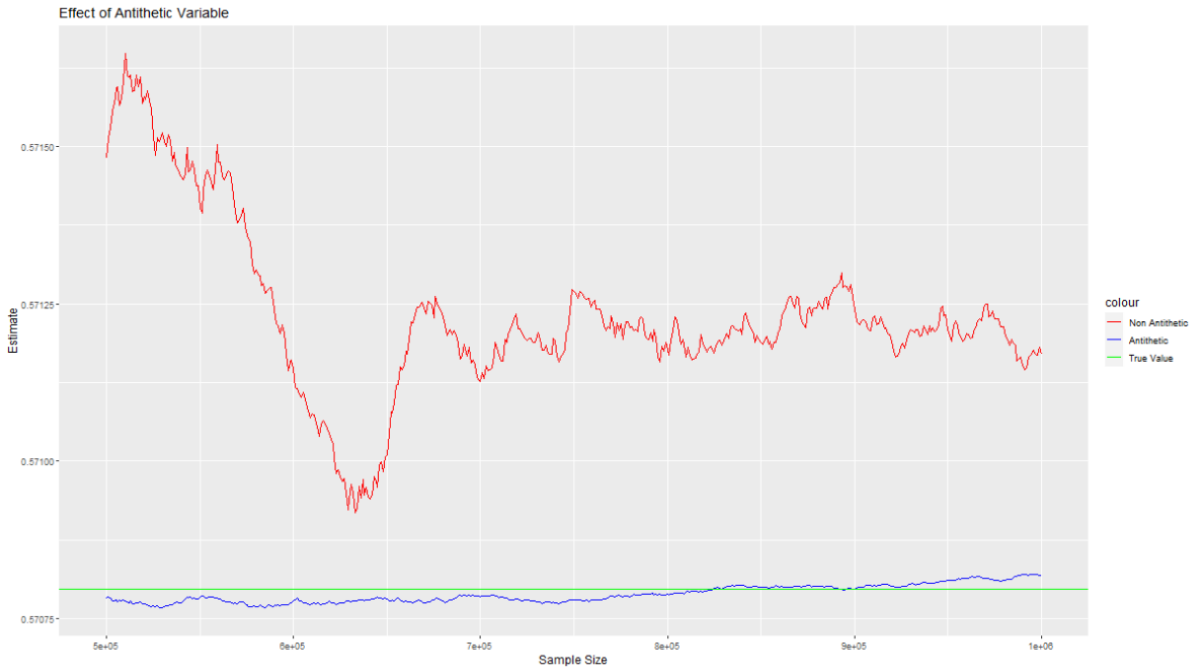
Therefore, the confidence interval for X is given by:

$$\left[\bar{X} - z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}},\ \bar{X} + z_{1-\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}\right]$$

The table below demonstrates the results I have obtained.

	Estimate	**Variance**	**CI Lower Bound**	**CI Upper Bound**
Monte-Carlo	0.5711698	0.1033277	0.5705397	0.5717998
Antithetic	0.5708179	0.0008333	0.5707614	0.5708745

Graphs



This graph demonstrates the effectiveness of applying antithetic variables. Even for a much smaller sample size, it is possible to obtain a very good estimation. In addition, it is much more beneficial to use this variance reduction method compared to increasing the sample size as it reduces the computational burden whilst providing higher accuracy.

Variance Reduction Methods

With some extra thinking, you could also apply both stratified sampling and antithetic variables to get almost a perfect estimation. The only drawback of using stratified sampling is that finding its variance is not as easy as applying the 'var' function in R and requires more thinking behind it. In this case, stratified sampling involves breaking (0,1) into smaller intervals of range 1/n and then sampling a uniform from these smaller intervals. This method exploits the inequality:

$$\mathbb{E}(\mathrm{Var}(X|Y)) \leq \mathrm{Var}(X)$$

to reduce the variance of my estimation significantly.

Since tanh(-log(X)) is a monotonically increasing function, we can also combine this method with an antithetic variable to obtain an almost perfect estimation.

```
# This chunk shows a method combining stratified sampling and antithetic v
ariable
j = 1:nSamples

X1 = tanh(-log((U+j-1)/nSamples))
X2 = tanh(-log((j-U)/nSamples))

Z2 = (X1+X2)/2

soln_strat = mean(Z2)
```

```
# It is very clear that the stratified sampling yields the most accurate s
olution by far
data.frame(true_value, soln_MC, soln_antithetic,soln_strat)

##   true_value   soln_MC soln_antithetic soln_strat
## 1  0.5707963 0.5711698       0.5708179  0.5707963

# However, identifying variance of stratified sample is not as easy as usi
ng the 'var' function
data.frame(variance_MC, variance_antithetic)

##   variance_MC variance_antithetic
## 1   0.1033277        0.0008333348
```

The following table displays the estimations

	Monte Carlo	**Antithetic**	**Stratified + Antithetic**	**True Value**
Estimation	0.5711698	0.5708179	0.5707963	0.5707963
Variance	0.1033277	0.0008333	N/A	N/A

As you can see, applying an antithetic variable alone significantly improves the estimate and the variance. However, applying both stratified and antithetic yields the true value up to 7 significant figures.

Extra: a control variable of **Y = 0.5 – X** could have also been used but we know stratified sampling is significantly better in this example.

Note: I have used antithetic variables and stratified sampling multiple times throughout this coursework and the principles above will apply the same way.

Question 2

Method

At first glance, the most obvious way to generate a sample with this distribution would be to use an Acceptance-Rejection method with envelope **Y ~ Exp(b)**.

$$\frac{2b}{\pi - 2} e^{-by} tanh(by) \leq Cbe^{-by}$$

and since **|tanh(x)| ≤ 1**, we can take **C = 2/(π−2)**.
This yields an acceptance rate of 57.1% which is relatively good considering the envelope is very easy to generate in R and no thinking needed to be done.

$$\frac{\frac{2b}{\pi-2}e^{-by}tanh(by)}{Cbe^{-by}} = tanh(by)$$

Therefore we generate **U ~ Uniform(0,1)** and **Y ~ Exp(b)** then accept **Y** if

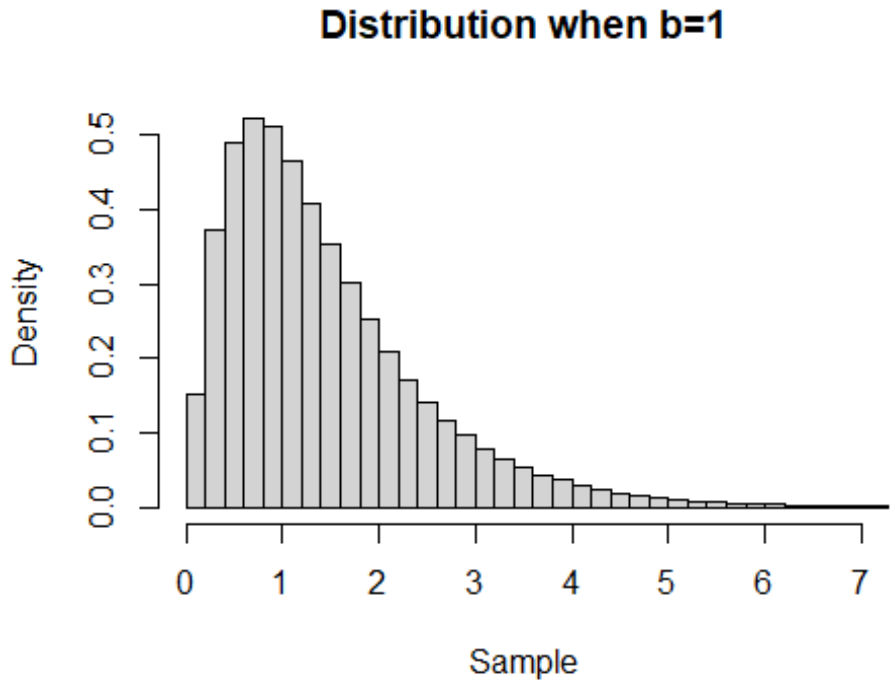
$$U \leq tanh(by)$$

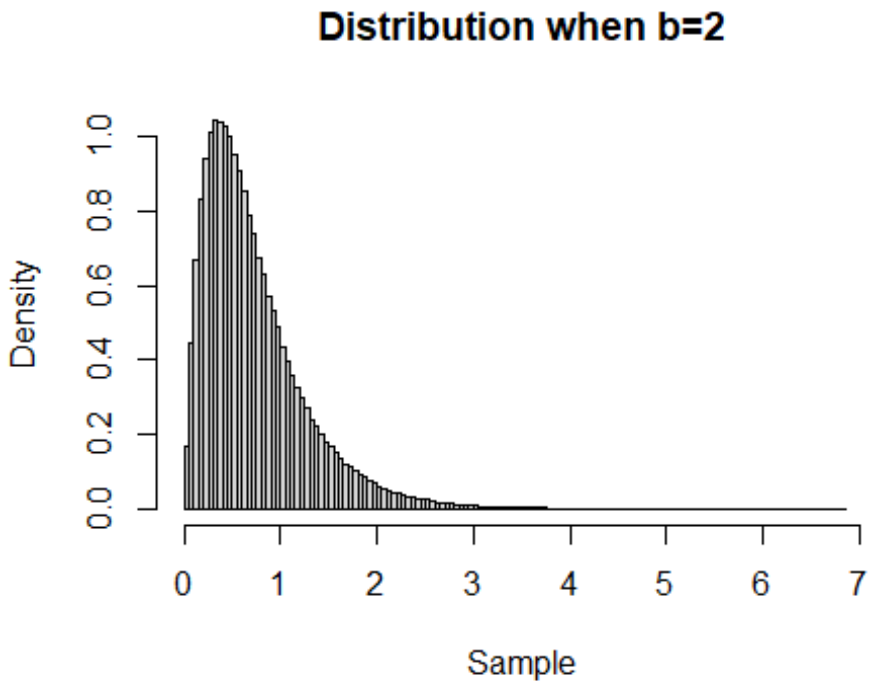
Results and Graphs

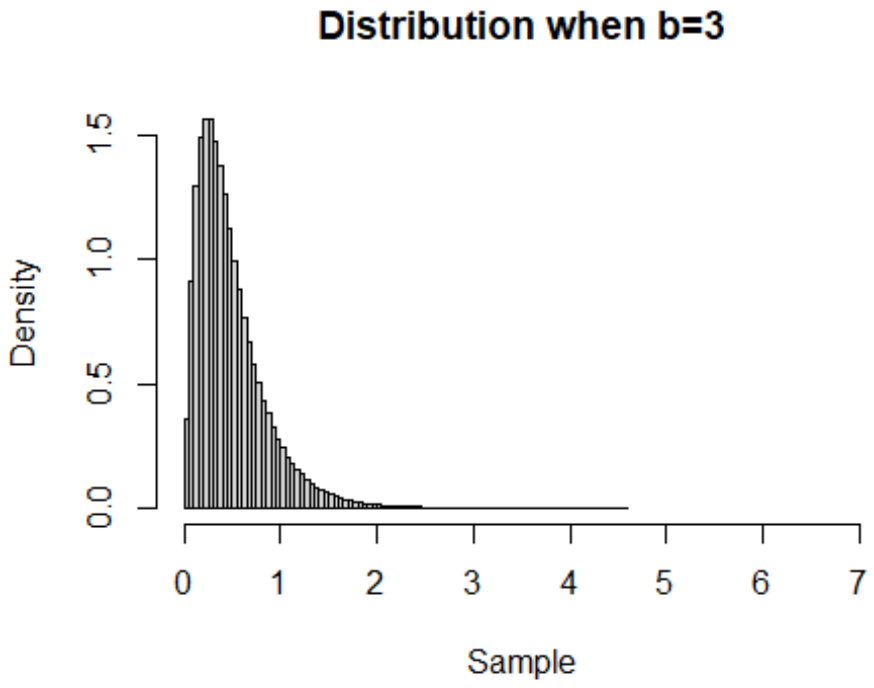
The following table is just a quick summary of my results for different values of b. My code allows you to easily investigate for different values of b.

b <dbl>	sample_mean <dbl>	sample_variance <dbl>	sample_acceptance_rate <dbl>	acceptance_rate <dbl>
1	1.4565628	1.1552528	0.571353	0.5707963
2	0.7292912	0.2908615	0.571735	0.5707963
3	0.4863874	0.1299616	0.571078	0.5707963

The following graphs show the distribution for each value of b.







The pdf seems to shift to the left as we increase b. The table above also supports this as the mean and variance both decrease as we increase b.

I decided to use Monte-Carlo methods to estimate the theoretical moments of the distribution since:

$$\mathbb{E}[Y] = \int_0^\infty y \frac{2b}{\pi - 2} e^{-by} tanh(by) dy$$
$$\mathbb{E}[Y^2] = \int_0^\infty y^2 \frac{2b}{\pi - 2} e^{-by} tanh(by) dy$$

and applying the transformation $\mathbf{x = e^{-by}}$ gives us:

$$\mathbb{E}[Y] = \int_0^1 \frac{2}{\pi - 2} tanh(-log(x))(-\frac{1}{b} log(x)) dx$$
$$= \mathbb{E}[\frac{2}{\pi - 2} tanh(-log(X))(-\frac{1}{b} log(X))]$$

and

$$\mathbb{E}[Y^2] = \int_0^1 \frac{2}{\pi - 2} tanh(-log(x))(-\frac{1}{b} log(x))^2 dx$$
$$= \mathbb{E}[\frac{2}{\pi - 2} tanh(-log(X))(-\frac{1}{b} log(X))^2]$$

where **X ~ Uniform(0,1).**
I then generated **X ~ Uniform(0,1)** and applied both stratified sampling and antithetic variable to obtain the following:

sample_mean <dbl>	Strat_Mean <dbl>	sample_variance <dbl>	Strat_Variance <dbl>
1.4565628	1.4574917	1.1552528	1.161970
0.7292912	0.7287459	0.2908615	0.290490
0.4863874	0.4858306	0.1299616	0.129108

The stratified estimate is a better estimate of the true value of the mean and variance hence I compared the sample mean and variance to this value. The sample estimates are reasonably close to the stratified estimates. This is a confirmation that the generated sample does follow the distribution as desired.

Question 3

Method

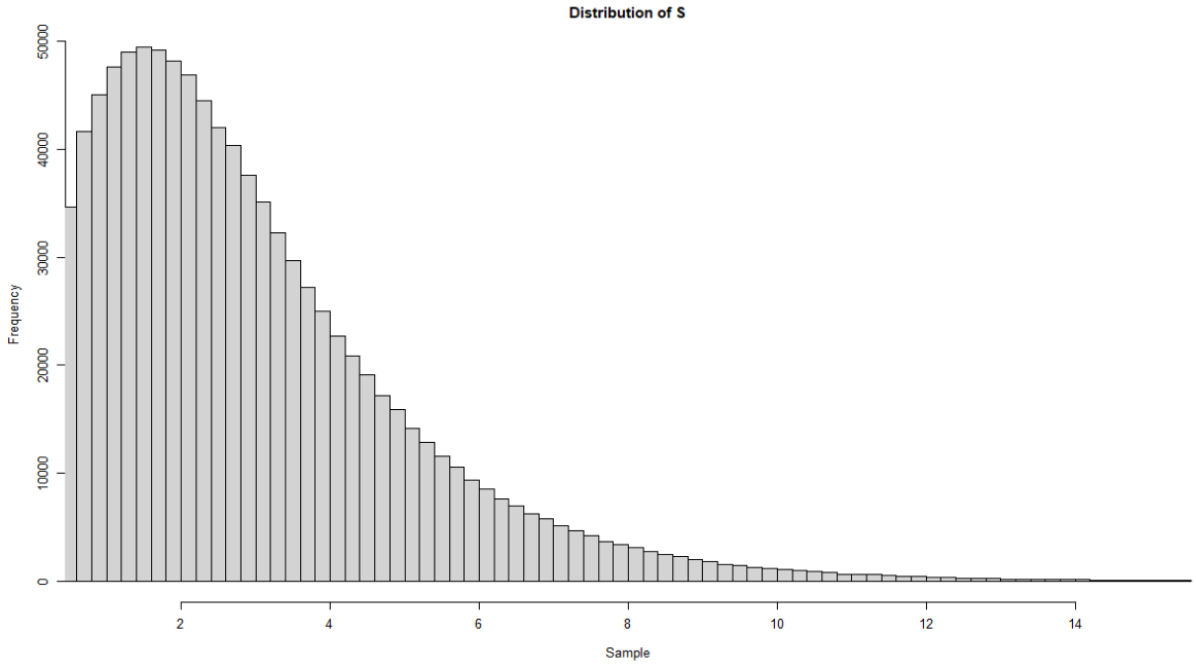
Before I started to do any work, I noticed that the distribution of N is a mixture of **Poisson(1)** and **Poisson(2)** with equal probability. The problem at hand is a random sum hence we must first generate N followed by N samples of Y. Since I am using A-R to generate Y like in question 2, it is important not to generate a random sample size as we did in question 2 but to generate a fixed size. To do this, I have used a while loop to repeat the A-R process until we accept a value. Finally, I sum all the Y values to obtain S as stated in the question.

```
# The following function repeats A-R until we obtain an accepted value. Th
e reason I did this is because this specific case requires me to generate
a sample of a fixed size
A_R <- function(v) {
  while (1) {
    U <- runif(1)
```

```
    Y <- rexp(1, 1)

    if (U<tanh(Y))
    return(Y)
  }
}

# Following function generates the poisson variable N first then uses this
to determine how many Ys we must generate.
RV_Generator <-function(x)  {
  q <-sample(c(1,2),size=1)
  N <-rpois(1,q)

  Y <- sapply(1:N, A_R)
  S = sum(Y)

  return(S)
}

S = sapply(1:nSamples, RV_Generator)
```

To estimate $\mathbb{P}(S>2.3)$, I calculated the average number of times S exceeds 2.3 since

$$\mathbb{P}(S > 2.3) = \mathbb{E}[\mathbf{1}_{[S>2.3]}(S)]$$



My estimate is $\mathbb{P}(S>2.3) = 0.530739$ which is a reasonable value based on the graph of the distribution.

Question 4

Method

For this question, I must firstly approximate C. To do this, I used the fact that the sum to infinity of the pdf must equal 1.

$$\sum_{even} C\frac{e^{-4}4^k}{2k!} + \sum_{odd} C\frac{e^{-4}4^k}{k!} = C(\sum_{even} \frac{e^{-4}4^k}{2k!} + \sum_{odd} \frac{e^{-4}4^k}{k!}) = 1$$

So, I sampled integers 0:200 and calculated the appropriate probability for each value depending on whether it's even or odd. I then summed up this probability. The value of C is 1 divided by the total probability. I obtained C = 1.33348.

Extra: I came up with the indicator function 1 – 0.5($\mathbf{1}_{[X \text{ is even}]}(X)$) to adjust for the different pdfs.

$$C\sum_{0}^{\infty} \frac{e^{-4}4^k}{k!}(1 - 0.5\mathbf{1}_{[X \text{ is even}]}(X)) = 1$$

After this, I generated the sample using the inverse transform method for a discrete variable. The method is very similar to that of generating a Poisson shown in the lecture notes but with a slight twist. Although we should start the algorithm with the most probable outcome to improve computational efficiency, the parameter of the function is small enough where the increase computational strain is negligible.

Algorithm

Step 1: Generate a U ~ Uniform(0,1)

Step 2: Generate base case (when j=0). Set

$$j = 0$$
$$p = 0.5C\frac{e^{-4}4^0}{2k!}$$
$$b = p$$
$$X = j$$

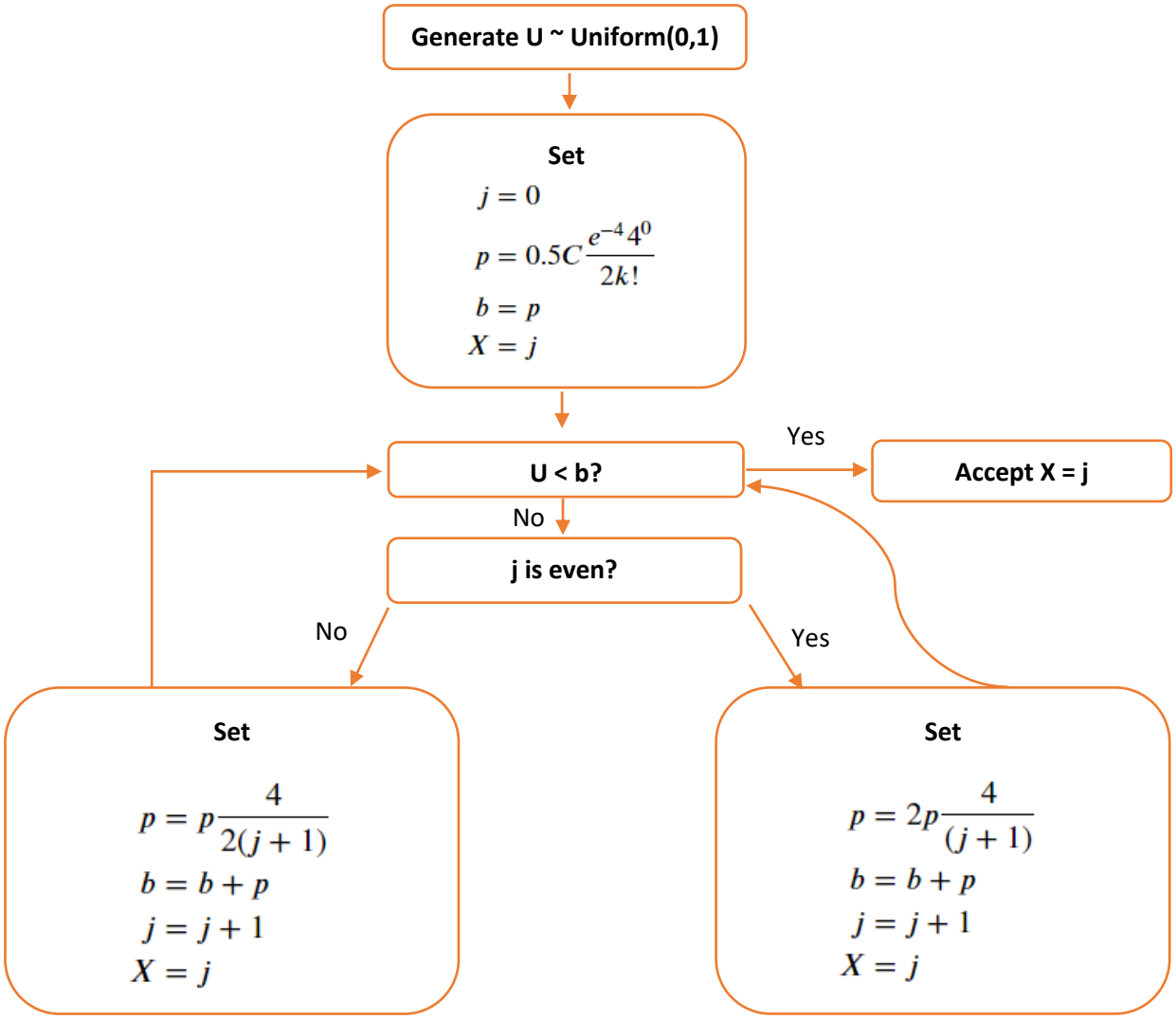
Step 3: If U<b, then we take X=j otherwise continue…

Step 4: Adjust the following

$$p = p\frac{4}{j+1}(0.5 + 1.5\mathbf{1}_{[X \text{ is even}]}(X))$$
$$b = b + p$$
$$j = j + 1$$
$$X = j$$

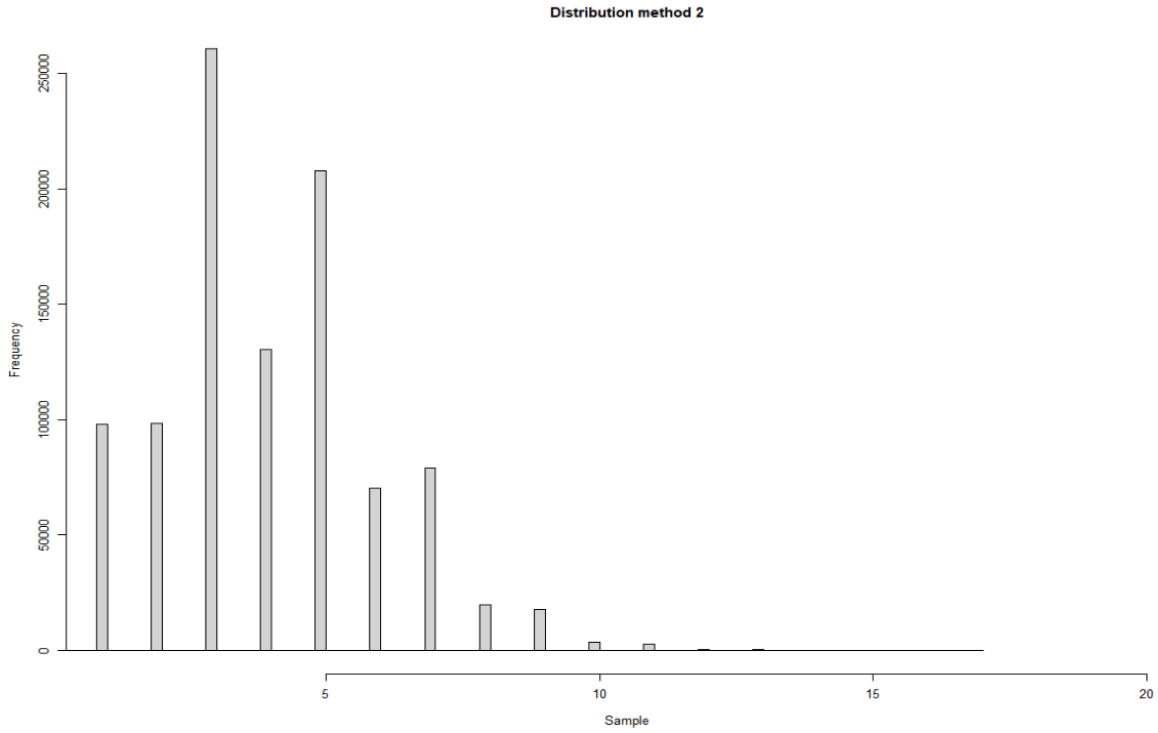
If U>b, go back to step 3 otherwise accept X.

Flowchart




```
RV_generator <- function(v) {
  j = 0
  U <- runif(1)
  p = 0.5*C*exp(-4)
  b = p
  X = j
  while (U>b) {
    p = p*(4/(j+1))*(0.5+1.5*(j %% 2 == 0)) # I have created this indicato
r function to adjust for the slight change in PDF depending on even or odd
j
    b = b + p
    j = j+1
    X = j
  }
  return(X)
}

X_method2 <- sapply(1:10^6, RV_generator)
```

Most of it follows what was demonstrated in the course notes. The important part I wanted to highlight was the use of the indicator function $0.5 + 1.5(\mathbf{1}_{[j \text{ is even}]}(j))$ which basically multiplies the pdf by 0.5 if we are switching from odd to even and multiplies the pdf by 2 if we are switching from even to odd.



Mean	Variance
3.987740	3.997029

As you can see, the mean and variance very similar. This is not surprising since the pdf looks very similar to a Poisson and we know that the mean and variance of a Poisson variable are both equal to the parameter of the distribution, in this case it is 4.
The following table compares the true probability and the sample proportions for k = 0, …, 5. Since the values are very similar, we can conclude that the sample generated comes from the given pdf.

Y <fctr>	Freq <dbl>	True_Probabilities <dbl>
0	0.012141	0.01221179
1	0.097698	0.09769433
2	0.097655	0.09769433
3	0.261446	0.26051822
4	0.129893	0.13025911
5	0.208564	0.20841457

Question 5

I decided to use a discrete uniform distribution for the scores because it is very easy to sample. I believe the choice of distribution does not matter because they will be IID for each candidate anyway. Again, I believe this question is easier to explain with the code hence I repeated the important parts below.

```
f5 <- function(k, N, a) {
  X <- sample(1:100, k, replace=TRUE)
  Y <- sample(1:100, N-k, replace=TRUE)

  # The following IF statement determines who is the selected candidate
  if (max(Y)>max(X)){
    winner = min(which(Y>max(X)))
  } else  {
    winner = N-k
  }

  # The following identifies whether the selected candidate is the highest
scoring candidate or not
  indicator = as.numeric(Y[winner]==max(X,Y))

  return(indicator)
}
```

This function consists of 3 arguments. k is the number of candidates that will be rejected, almost like the 'test set'. N is the number of candidates which will be important for the last part of the question. a is a dummy variable used for creating many samples later using the 'sapply' function. X and Y are discrete uniform samples.

The 'if' statement simply identifies the selected candidate. The variable 'indicator' takes a value of 1 if the selected candidate has the highest score amongst its cohort and 0 otherwise.

```
k = 1:11
X = c()

for (i in k) {
  X[i] = sum(sapply(1:nSamples, f5, k=i, N=12))
}

probability = X/nSamples

max_probability = max(probability)
optimal_k = which.max(probability)
```

Firstly, I defined the values of k I am interested in which is basically every possible value it can take. The 'for' loop calculates the total number of times the selected candidate achieves the highest score amongst its cohort out of **nSamples** for every value of k, keeping N constant.

The probability is simply the total number of times the selected candidate achieves the highest score divided by the total number of samples generated since

$$\mathbb{P}(Score_X = Max) = \mathbb{E}[\mathbf{1}_{[Score_X = Max]}(Score_X)]$$

where X is the selected candidate.

```
N = 10:18
Y = c()

for (i in N) {
  Y[i-9] = sum(sapply(1:nSamples, f5, k=optimal_k, N=i))
}


probability_part2 = Y/nSamples

max_probability_part2 = max(probability_part2)
optimal_N = which.max(probability_part2) + 9
```

I now relax the value of N and fix k to be the optimal k obtained in the previous part. This code is very similar to that above but varying N instead. My approximations suggest that the optimal value of k is 4 and N is 11. An important point I must point out is that this does not suggest it is the optimal combination to maximize the probability. They are only the optimal holding the other at a constant value. The probability of selecting the correct candidate when k=4 and N=11 is 0.40417.

Appendix

QUESTION 1

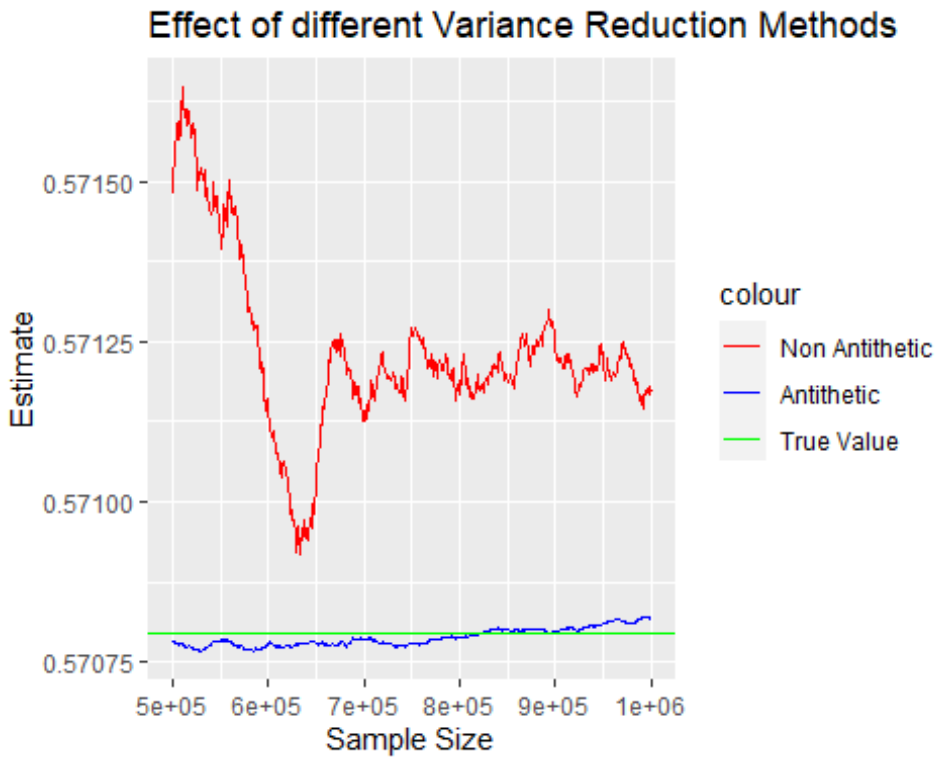
```
# I have cleared workspace at the start of each question to keep things ti
dy
rm(list = ls())
set.seed(56473)

nSamples = 10^6

# I am going to use the most obvious variance reduction method first, anti
thetic variable
U <- runif(nSamples)

X = tanh(-log(U))
Y = tanh(-log(1-U)) # Applying antithetic variable
Z = (X+Y)/2

soln_MC = mean(X) # This is the standard MC solution without any variance
reduction
soln_antithetic = mean(Z) # This is the solution to the integral after app
lying the antithetic variable

true_value = (pi-2)/2

variance_MC = var(X)
variance_antithetic = var(Z)
```

Calculating Confidence Intervals for each estimation

```
MC_CI = c(soln_MC-1.96*sqrt(variance_MC)/sqrt(nSamples),soln_MC+1.96*sqrt(
variance_MC)/sqrt(nSamples))
Antithetic_CI = c(soln_antithetic-1.96*sqrt(variance_antithetic)/sqrt(nSam
ples),soln_antithetic+1.96*sqrt(variance_antithetic)/sqrt(nSamples))
```

The next chunk plots a graph of the estimated solution for different sample sizes. This is to show the effectiveness of antithetic variables and how close the approximations are to the true value.

```
library(ggplot2)
# The following functions calculates the mean of the first v sample for ea
ch of the methods
f_X <- function(v) {
  XX = mean(X[1:v])
  return(XX)
}

f_Z <- function(v) {
  ZZ = mean(Z[1:v])
  return(ZZ)
}

x_seq = seq(5*10^5, 10^6, 1000) # Creates a sequence with an increment of
1000 starting at 500 000

estimates_MC = sapply(x_seq, f_X)
estimates_antithetic = sapply(x_seq, f_Z)

estimate_dataset = data.frame(x_seq, estimates_MC, estimates_antithetic)

# Code below uses the ggplot package to display the estimates for differen
t sample sizes
ggplot(data = estimate_dataset, aes(x = x_seq))+
  geom_line(aes(y = estimates_MC, colour="Non Antithetic"))+
  geom_line(aes(y = estimates_antithetic, colour="Antithetic"))+
  geom_hline(aes(yintercept = true_value,colour="True Value"))+
  labs(title = "Effect of different Variance Reduction Methods",
       y = "Estimate",
       x = "Sample Size")+
  scale_color_manual(values = c("Non Antithetic"="red", "Antithetic"="blue
", "True Value" = "green")) # Creates a legend
```

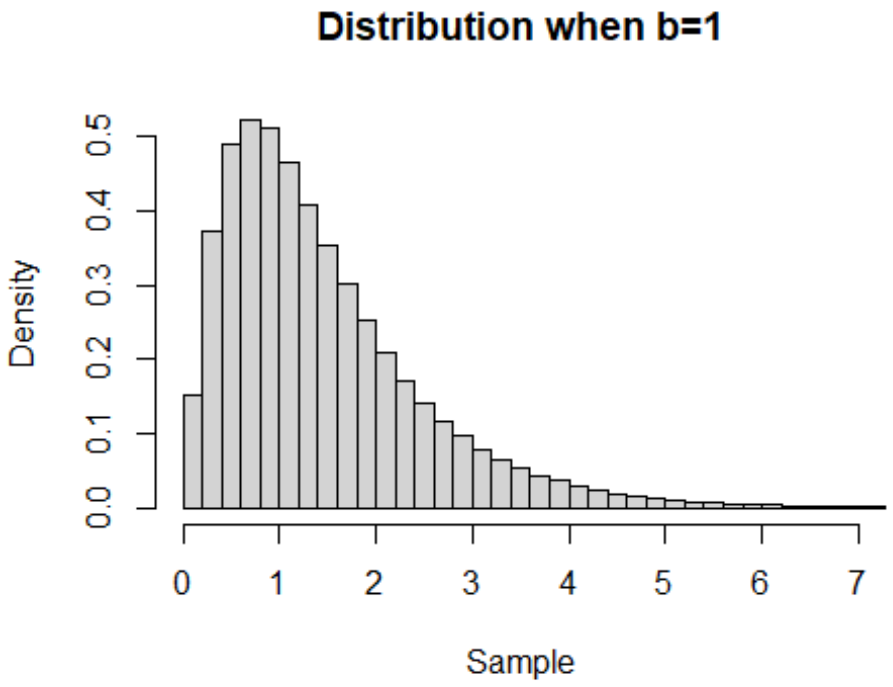



```
# This chunk shows a method combining stratified sampling and antithetic v
ariable
j = 1:nSamples

X1 = tanh(-log((U+j-1)/nSamples))
X2 = tanh(-log((j-U)/nSamples))

Z2 = (X1+X2)/2

soln_strat = mean(Z2)

# It is very clear that the stratified sampling yields the most accurate s
olution by far
data.frame(true_value, soln_MC, soln_antithetic,soln_strat)

##   true_value   soln_MC soln_antithetic soln_strat
## 1  0.5707963 0.5711698       0.5708179  0.5707963

# However, identifying variance of stratified sample is not as easy as usi
ng the 'var' function
data.frame(variance_MC, variance_antithetic)

##   variance_MC variance_antithetic
## 1   0.1033277        0.0008333348
```

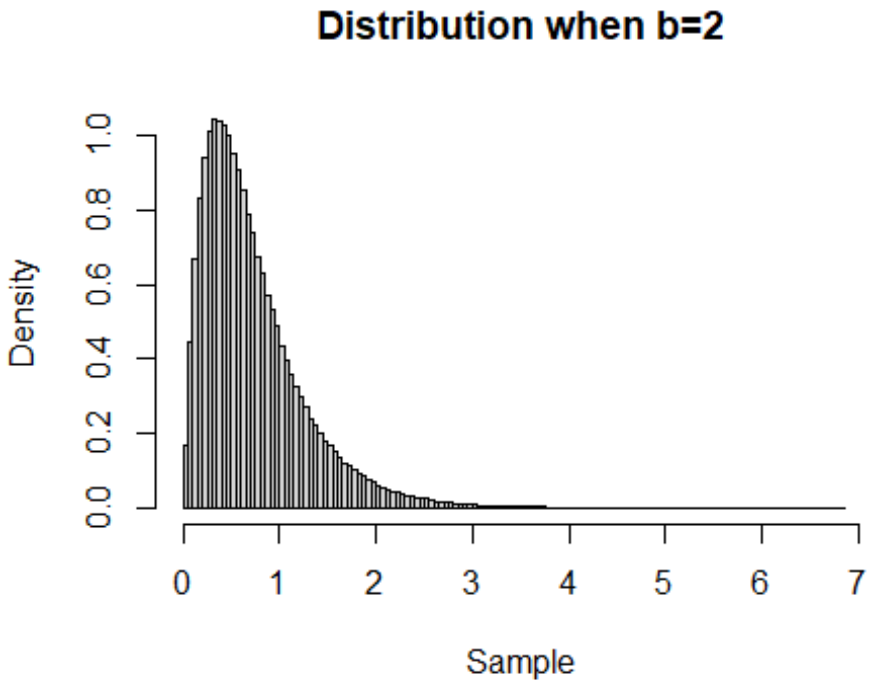
QUESTION 2

```
rm(list = ls())
set.seed(56473)

nSamples = 10^6
```

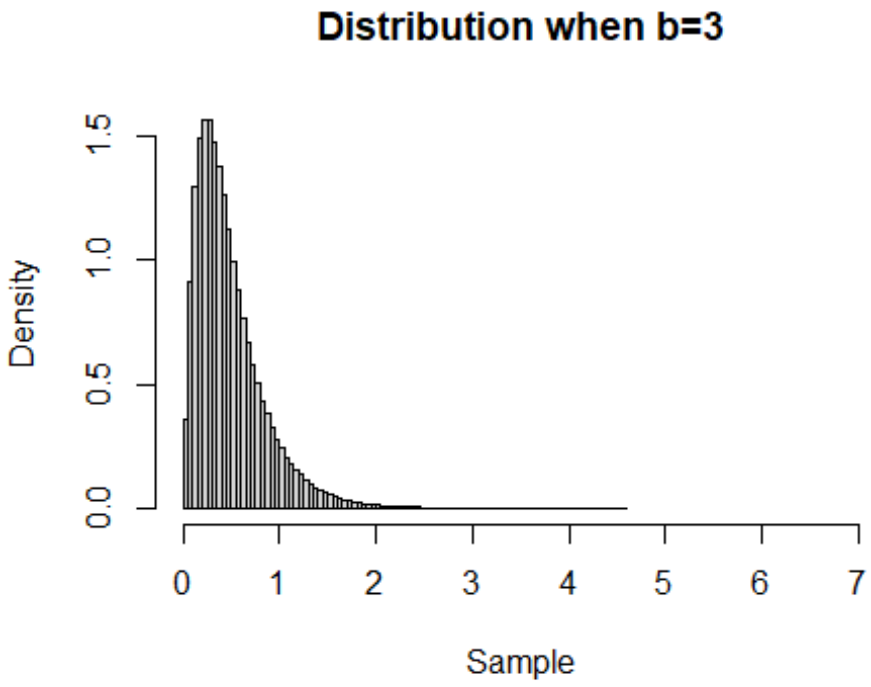
```r
c = 2/(pi-2)  # This is the constant C used in my A-R method
b = c(1,2,3)  # The different values of b I am interested in

# The following function generates a sample for the distribution using A-R
for your chosen value of b
generator <- function(v) {
  U <- runif(nSamples)
  Y <- rexp(nSamples, v)

  Y = Y*(U<tanh(v*Y)) # Only accepts Y if U<tanh(bY)
  Y = Y[!Y %in% c(0)] # Removes all the zeros

  return(Y)
}

X = sapply(b, generator)

# Here I am simply separating the 3 different distributions
sample_b1 = unlist(X[1])
sample_b2 = unlist(X[2])
sample_b3 = unlist(X[3])

mean_1 = mean(sample_b1)
mean_2 = mean(sample_b2)
mean_3 = mean(sample_b3)

var_1 = var(sample_b1)
var_2 = var(sample_b2)
var_3 = var(sample_b3)

actual_acceptance_rate_1 = length(sample_b1)/nSamples
actual_acceptance_rate_2 = length(sample_b2)/nSamples
actual_acceptance_rate_3 = length(sample_b3)/nSamples

sample_mean = c(mean_1,mean_2,mean_3)
sample_variance = c(var_1,var_2,var_3)
sample_acceptance_rate = c(actual_acceptance_rate_1,actual_acceptance_rate
_2,actual_acceptance_rate_3)

acceptance_rate = 1/c

data.frame(b,sample_mean,sample_variance, sample_acceptance_rate, acceptan
ce_rate)

##   b sample_mean sample_variance sample_acceptance_rate acceptance_rate
## 1 1   1.4565628       1.1552528               0.571353       0.5707963
## 2 2   0.7292912       0.2908615               0.571735       0.5707963
## 3 3   0.4863874       0.1299616               0.571078       0.5707963

# The following code plots the distribution so it is easy to see how the v
alue of b affects the distribution
hist(sample_b1, main = paste("Distribution when b=1"), xlab = "Sample", xl
im = range(0:7), freq= FALSE, breaks = 100)
```




```
hist(sample_b2, main = paste("Distribution when b=2"), xlab = "Sample", xl
im = range(0:7), freq= FALSE, breaks = 100)
```




```
hist(sample_b3, main = paste("Distribution when b=3"), xlab = "Sample", xl
im = range(0:7), freq= FALSE, breaks = 100)
```



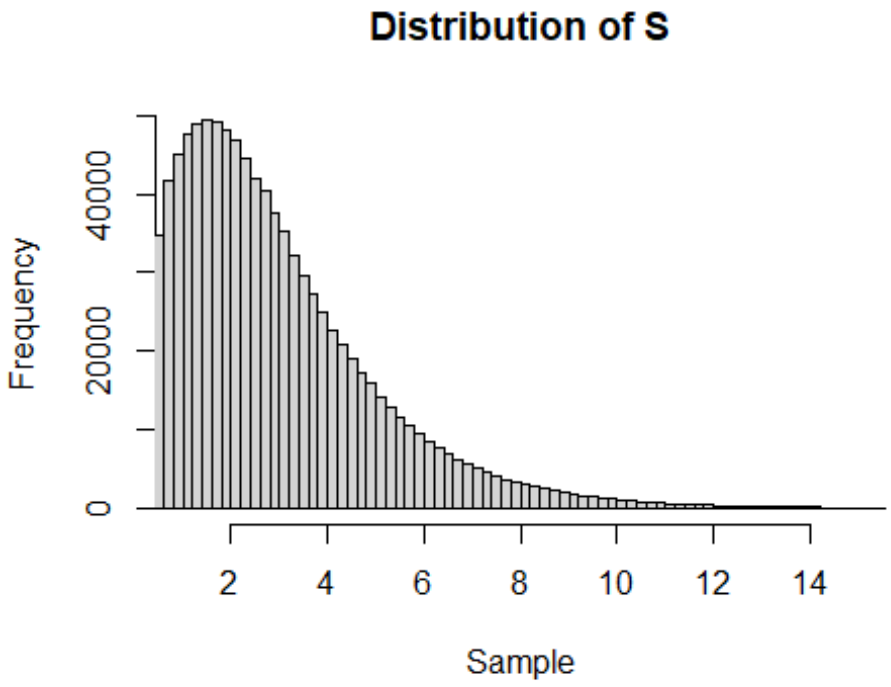

```
# A quick trend you can spot is that higher b leads to the sample being sm
aller. It is less likely to obtain larger numbers. The peak of the pdf shi
fts to the left.
```

Here I calculate the mean and variance of the distribution using Monte-Carlo method and applying antithetic variable to obtain better estimates of the moments.

```
# The following function calculates the moments of your choice of the dist
ribution, although only tested for the first 2 moments.
# I am combining stratified sampling and antithetic variables again
moments <- function(b,v) {
  U <- runif(nSamples)
  j = 1:nSamples

  X = (2/(pi-2)) * tanh(-log((U+j-1)/nSamples)) * (-log((U+j-1)/nSamples)/
b)^v
  Y = (2/(pi-2)) * tanh(-log((j-U)/nSamples)) * (-log((j-U)/nSamples)/b)^v
  Z = 0.5*(X+Y)
  return(mean(Z))
}

# This calculates the mean and variance using MC
Strat_Mean = sapply(b,moments,v=1)
Strat_Variance = sapply(b, moments, v=2) - Strat_Mean^2

data.frame(sample_mean, Strat_Mean, sample_variance, Strat_Variance)

##   sample_mean Strat_Mean sample_variance Strat_Variance
## 1   1.4565628  1.4574917       1.1552528      1.1619630
## 2   0.7292912  0.7287461       0.2908615      0.2904940
## 3   0.4863874  0.4858307       0.1299616      0.1291069
```

QUESTION 3

```
rm(list = ls())
set.seed(56473)

nSamples = 10^6

# The following function repeats A-R until we obtain an accepted value. Th
e reason I did this is because this specific case requires me to generate
a sample of a fixed size
A_R <- function(v) {
  while (1) {
    U <- runif(1)
    Y <- rexp(1, 1)

    if (U<tanh(Y))
    return(Y)
  }
}

# Following function generates the poisson variable N first then uses this
to determine how many Ys we must generate.
RV_Generator <-function(x)  {
  q <-sample(c(1,2),size=1)
  N <-rpois(1,q)

  Y <- sapply(1:N, A_R)
  S = sum(Y)

  return(S)
}

S = sapply(1:nSamples, RV_Generator)

mean = mean(S)
variance = var(S)
# Since the probability is equal to the expectation of the indicator funct
ion
probability = sum(S>2.3)/nSamples

cbind(mean, variance, probability)

##          mean variance probability
## [1,] 2.918674  4.42613    0.530739

hist(S, main = paste("Distribution of S"), xlim = range(1:15), xlab = "Sam
ple", breaks = 100)
```

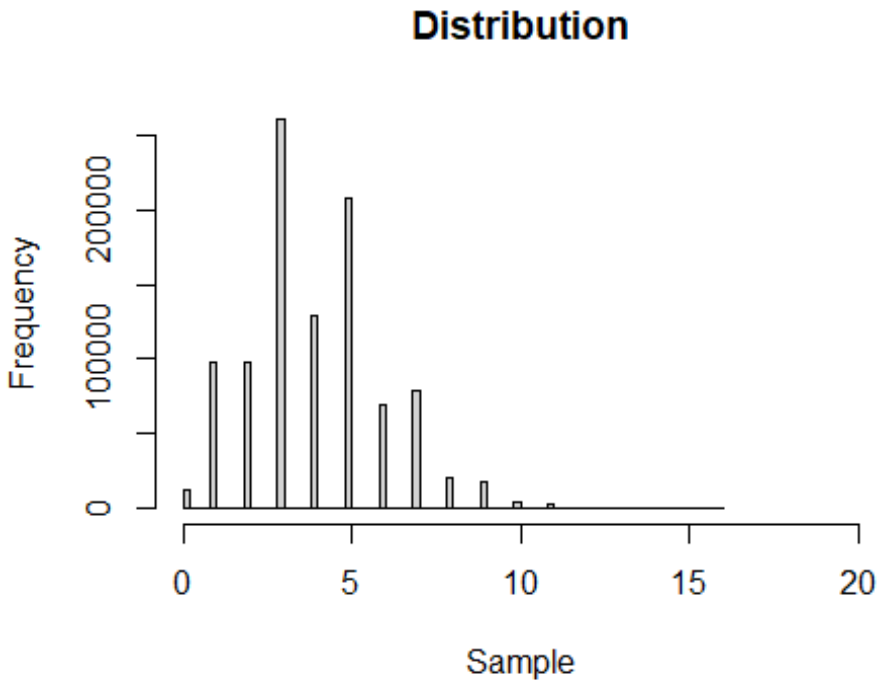



```
          QUESTION 4

rm(list = ls())
set.seed(56473)

# In this question, I have used the fact that the sum of the PDF equals 1
by definition. To approximate C, I decided to sum the probabilities for 0:
200 and therefore use this to calculate C.
nSamples = 10^6
sample = 0:200  # This is the sample used for approximating C. We only nee
d a small sample since the probability tends to zero very fast

# The following function identifies if v is odd or even then applies the c
orrect PDF.
f_C <- function(v) {
  X = (exp(-4)*4^v)/factorial(v)*(1-0.5*(v%%2==0))
  return(X)
}

probabilities = sapply(sample, f_C)
total_prob = sum(probabilities)

# For the envelope I have used, I have taken the constant used in the A-R
method to be equal to our approximated value of C
C = 1/total_prob
```

The next block uses the inverse transform method for a discrete r.v. to generate the sample instead. In general, inverse transform methods are preferred over A-R methods.

```
# This function generates a sample using the inverse transform method for
a discrete
```

```
RV_generator <- function(v) {
  j = 0
  U <- runif(1)
  p = 0.5*C*exp(-4)
  b = p
  X = j
  while (U>b) {
    p = p*(4/(j+1))*(0.5+1.5*(j %% 2 == 0))
    # I have created this indicator function to adjust for the slight chan
ge in PDF depending on even or odd j
    b = b + p
    j = j+1
    X = j
  }
  return(X)
}

Y <- sapply(1:10^6, RV_generator)
mean = mean(Y)
variance = var(Y)


cbind(mean,variance)

##          mean variance
## [1,] 3.998927 3.989162

hist(Y, main = paste("Distribution"), xlim = range(0:20), xlab = "Sample",
breaks = 100)
```



Testing whether algorithm works.

```
Sample_proportions = head(prop.table(table(Y)))
True_Probabilities = C*probabilities[1:6]
data.frame(Sample_proportions, True_Probabilities)

##   Y     Freq True_Probabilities
## 1 0 0.012141         0.01221179
## 2 1 0.097698         0.09769433
## 3 2 0.097655         0.09769433
## 4 3 0.261446         0.26051822
## 5 4 0.129893         0.13025911
## 6 5 0.208564         0.20841457

         QUESTION 5

rm(list = ls())
set.seed(56473)

nSamples = 10^5

f5 <- function(k, N, a) {
  X <- sample(1:100, k, replace=TRUE)
  Y <- sample(1:100, N-k, replace=TRUE)

  # The following IF statement determines who is the selected candidate
  if (max(Y)>max(X)){
    winner = min(which(Y>max(X)))
  } else  {
    winner = N-k
  }

  # The following identifies whether the selected candidate is the highest
scoring candidate or not
  indicator = as.numeric(Y[winner]==max(X,Y))

  return(indicator)
}
```

The next block finds the optimal k

```
k = 1:11
X = c()

for (i in k) {
  X[i] = sum(sapply(1:nSamples, f5, k=i, N=12))
}

probability = X/nSamples

max_probability = max(probability)
optimal_k = which.max(probability)
```

The following block investigates the effects of changing the number of candidates

```
# Lets now relax the value of N and fix k = optimal_k
N = 10:18
```

```
Y = c()

for (i in N) {
  Y[i-9] = sum(sapply(1:nSamples, f5, k=optimal_k, N=i))
}

probability_part2 = Y/nSamples

max_probability_part2 = max(probability_part2)
optimal_N = which.max(probability_part2) + 9

# For optimal k and optimal N, the probability of choosing the correct can
didate is...
Optimal_probability = probability_part2[optimal_N-9]

cbind(optimal_k, optimal_N, Optimal_probability)

##      optimal_k optimal_N Optimal_probability
## [1,]         4        10             0.40417
```

Challenges

Looking back at all the challenges I submitted, I could've improved the method significantly using the skills I have developed throughout this course! However, I have included the original copies I submitted with some tidying up and removing obvious inefficiencies.

Challenge 1

For challenge 1, I simulated 5 talented and 5 untalented candidates using the 'rnorm' function. I then checked if the 3rd highest scoring talented candidate is greater than the 1st highest scoring untalented candidate. If this is true, then all 3 selected candidates are talented hence we set the indicator function to be 1. Since the probability is the expectation of the indicator function, I estimated the probability by calculating the average of the indicator function.

```
nSample = 10^6
hit_seq = c(0)

for (i in 1:nSample){
  talented <- rnorm(5, 70, 10) #Simulating random normal
  untalented <- rnorm(5,50,10)
  talented = sort(talented) #sorting from smallest to highest
  untalented = sort(untalented)

  if (talented[3] >= untalented[5]) {
    hit_seq[i] = 1
  }else{
    hit_seq[i] = 0
  }
}
```

```
probability_1 = sum(hit_seq)/nSample
probability_1
```

```
## [1] 0.836604
```

Challenge 2

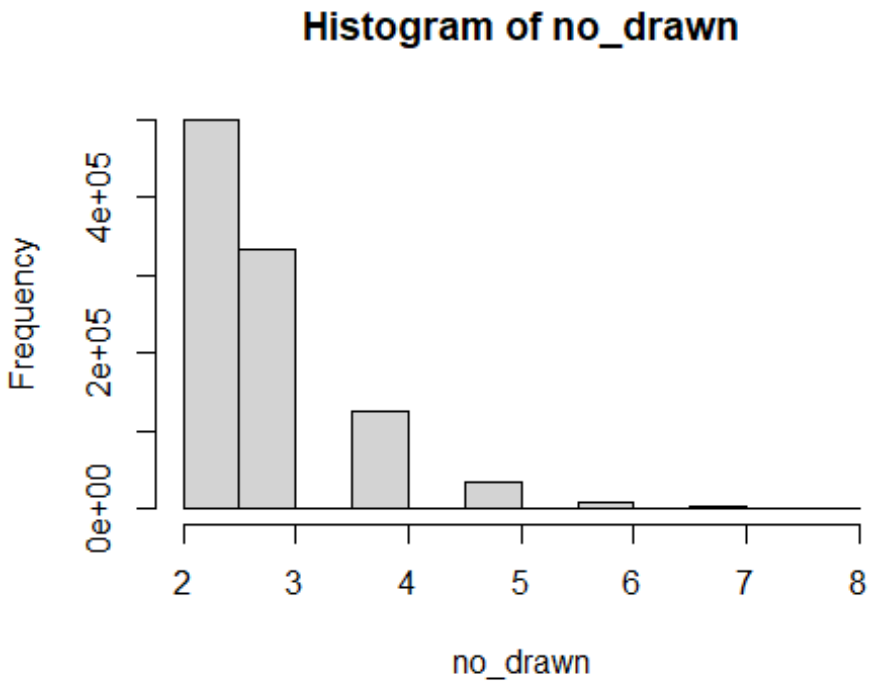
```
nSample = 10^6
hit_seq = rep(0,nSample)

for (i in 1:nSample){
  No_Tal <- rbinom(1, 6, 0.8) #Simulating binomial to determine how many t
alented individuals there are

  if (No_Tal > 2) { #If number of talented is less than 3, then it is not
possible for all 3 selected to be talented. So we can skip the next steps.
    talented <- rnorm(No_Tal, 70, 10)
    untalented <- rnorm(10-No_Tal, 50, 10)

    talented = sort(talented)
    untalented = sort(untalented)

    if (talented[No_Tal-2] > untalented[10-No_Tal]) {
      hit_seq[i] = 1
    } else {
      hit_seq[i] = 0
    }
  }
}

probability_1 = sum(hit_seq)/nSample
probability_1
```

```
## [1] 0.774515
```

Challenge 3

```
nSample = 1000000

# Defining a function to simulate many samples of draws
f1 = function(x, nCards) {
  return(sample(nCards, nCards, replace = FALSE))
}

draw = sapply(1:nSample, f1, nCards = 8)    # Creating many draws
difference = diff(draw, 1)    # Calculating difference between consecutive
draws



# Defining a function which identifies the first time you draw a lower num
ber
f2 = function(y) {
  which(difference[,y]<0)[1]
```

```
}

first_negative = sapply(1:nSample, f2)
first_negative = na.omit(first_negative)    # Removing all 'NA' which is w
hen there are no winners and the game is void

odd = as.numeric(first_negative %% 2 != 0)  # If the first time you draw a
lower number is odd, player A wins otherwise player B wins

probability = sum(odd)/nSample
probability

## [1] 0.63207

no_drawn = first_negative + 1   # This is the distribution of the number o
f draws
hist(no_drawn)
```



Testing for different number of cards

```
set.seed(127)
nSample = 100000

nCards = 8:12


probability2 = c()

# Defining a function to simulate many samples of draws
f1 = function(x, nCards) {
  sample = sample(nCards, nCards, replace=FALSE)
```

```
  difference = diff(sample, 1)
  first_negative = which(difference<0)[1]

  if (length(first_negative) != 0) {
    odd = as.numeric(first_negative%%2 != 0)
  } else {
    odd = NULL
  }
  return(odd)
}

for (i in nCards) {
  X <- sum(sapply(1:nSample, f1, nCards=i))
  probability2[i-7] = X/nSample
}

probability2
```

```
## [1] 0.63122 0.63210 0.63276 0.63270 0.63400
```

```
# As you can see, the probability does not change much at all as you chang
e the number of cards
```