Coursework

Alix Vermeulen

Friday 11th December 2020

```
# Installing packages if required (specifying a mirror to avoid possible errors)
suppressPackageStartupMessages({
if(!require(ggplot2)) install.packages("ggplot2",repos = "http://cran.us.r-project.org")
library(ggplot2,verbose=FALSE,warn.conflicts=FALSE,quietly=TRUE)
})
suppressPackageStartupMessages({
if(!require(pracma)) install.packages("pracma",repos = "http://cran.us.r-project.org")
library(pracma,verbose=FALSE,warn.conflicts=FALSE,quietly=TRUE)
})
suppressPackageStartupMessages({
if(!require(knitr)) install.packages("pracma",repos = "http://cran.us.r-project.org")
library(pracma,verbose=FALSE,warn.conflicts=FALSE,quietly=TRUE)
})
suppressPackageStartupMessages({
if(!require(dplyr)) install.packages("pracma",repos = "http://cran.us.r-project.org")
library(pracma,verbose=FALSE,warn.conflicts=FALSE,quietly=TRUE)
})
suppressPackageStartupMessages({
if(!require(tidyverse)) install.packages("pracma",repos = "http://cran.us.r-project.org")
library(pracma,verbose=FALSE,warn.conflicts=FALSE,quietly=TRUE)
})
```

Statement of the Problem

1. Implementation of either a rejection or ratio-of-uniform scheme to generate random variates from the probability density function $f_X(\cdot)$ given below. You should include both descriptions of your methodology and the commented R code, you should include a discussion of the acceptance rate of your scheme and of the computational time.

2. Verification of your scheme using at least two diagnostic plots and at least two statistical tests. You should include a discussion of the methods and interpret your results.

3. Implementation of a Monte Carlo procedure to estimate the normalizing constant associated with $f_X(\cdot)$. You should include a description of the approach that you have taken and compare your procedure to hit-or-miss Monte Carlo.

$$f_X(x) \propto \begin{cases} \frac{1}{(x-a)(b-x)} exp\left\{ \frac{-1}{c}\left(d + \log\left(\frac{x-a}{b-x}\right)\right)^2 \right\} & \text{for } x \in (a,b); \\ 0 & \text{otherwise.} \end{cases}$$

Introduction and Mathematical Background

In this report, I will be using the rejection method to simulation from the given pdf. I will first compare a well-matched envelope that I came up with, to the standard uniform uniform envelope. In this section, I will also be adding a pretest-squeezing condition to try and improve the efficiency of my rejection scheme. The next section consists of a series of diagnostic plots and goodness of fit tests to evaluate the performance of my chosen envelope. Lastly, I will compare 2 different methods used to estimate the normalizing constant of my pdf: Crude Monte Carlo and Hit-or-Miss.

The largest prime of my CID is 16301 which gives me the following parameters for my pdf: $a = 0, b = 5, c = 7$ and $d = 0.00$. Hence in my case, the goal will be to simulate from:

$$f_X(x) \propto \begin{cases} \frac{1}{x(5-x)} exp\left\{ \frac{-1}{7} \left(\log\left(\frac{x}{5-x} \right) \right)^2 \right\} & \text{for } x \in (0,5) \\ 0 & \text{otherwise} \end{cases}$$

Generating from $f_X(x)$ using Rejection

The general rejection sampling algorithm to simulate a random variable $X \sim f_X(x)$, requires specification of an envelope Mg_Y that mimics the shape of $f_X^*(x)$ and that satisfies the two following assumptions:

1. The support of g_Y encompasses that of f_X, i.e. $f_X(x) > 0 \implies g_Y(x) > 0$.
2. There exists an $M > 0$ such that $\forall x$ such that $f_X(x) > 0$, we have:

$$\frac{f_X(x)}{g_Y(x)} \leq M < \infty$$

where $M = \sup_x \frac{f_X(x)}{g_Y(x)}$. M can also be any other number greater than the supremum, as long as it satisfies $f_X(x) \leq Mg_Y(x)$, but the algorithm will be less efficient if a greater number is used.
Once, we have specified $g_Y(y)$ and M, the general rejection algorithm is given below.

General Rejection Algorithm

The rejection algorithm below also applies if we only know $f_X^*(x) \propto f_X(x)$ and/or $g_Y^*(y) \propto g_Y(y)$ for the specified ranges of the densities.

1. Generate $Y = y \sim g_Y(y)$.
2. Generate $U = u \sim U(0,1)$.
3. If $u \leq \frac{f_X(y)}{Mg_Y(y)}$, set $X = y$.
4. Otherwise GOTO 1.

In our case where we have $f_X^*(x)$, step 3 would be replaced by:

3'. If $u \leq \frac{f_X^*(y)}{Mg_Y(y)}$, set $X = y$.

We also know that the acceptance region is given by:

$$u \leq \frac{f_X^*(y)}{Mg_Y(y)}$$

And the acceptance probability, θ, of the algorithm is given by;

$$\theta = \frac{\int f_X^*(x)\,\mathrm{d}x}{M}$$

Step 1: Finding the envelope for f_X^*

I have picked g_Y to be:

$$g_Y(x) = \begin{cases} -0.04x + 0.25 & \text{for } x \in (0, 2.5] \\ 0.04x + 0.05 & \text{for } x \in (2.5, 5) \end{cases}$$

Indeed, g_Y satisfies the assumptions for the rejection scheme because g_Y is composed of two linear functions hence its support encompasses the support of f_X. Secondly, we are able to find a supremum M given by:

$$M = \sup_x \frac{f_X^*(x)}{g_Y(y)} = \sup_x \frac{1}{x(5-x)(-0.04x+0.25)} \exp\left\{\frac{-1}{7}(\log(\frac{x}{5-x}))^2\right\}.$$

We note that g_Y and f_X^* are symmetric about 2.5 hence we would get the same supremum when using $g(x) = 0.04x + 0.05$ i.e. $g(x)$ when $x \in (2.5, 5)$.

This choice of g_Y seems like a sensible choice to maximise the acceptance probability because it mimics f_X^*. Besides, g_Y will be easy to simulate from as it is composed of two linear functions that can be simulated from using inversion.

I will also be comparing this more optimal choice of g_Y to a uniform distribution, i.e. $g_{Y_U}(y) = \frac{1}{5}$ for $y \in (0, 5)$.

The following code chunk shows that using a uniform envelope $M_U g_{Y_U}$ yields a lower theoretical acceptance probability than my proposed g_Y. To find the supremum M_U (for the uniform envelope) and M (for g_Y), I will use the in-built optimize function in R. I also use the inbuilt integrate function to compute the integral of f_X^* from 0 to 5 that is needed for the theoretical acceptance probability. We indeed see that the theoretical acceptance probability is 76.2% hence we can do better.

```
library(knitr)
knitr::opts_chunk$set(warning=FALSE, message=FALSE)
opts_chunk$set(tidy.opts=list(width.cutoff=60),tidy=TRUE)
suppressMessages(library(tidyverse,verbose=FALSE,warn.conflicts=FALSE,quietly=TRUE))
suppressMessages(library(forecast,verbose=FALSE,warn.conflicts =FALSE,quietly=TRUE))

# Function to evaluate f*_X(x)
fstar <- function(x) 1/(x*(5-x))*exp((-1/7)*(log(x/(5-x)))^2)

gunif <- function(x) 1/5  # Function to evaluate a uniform g_Y_U(x) over x in 0 to 5

# Function to evaluate f*_X(x)/g_Y_U(x)
fstar_gunif <- function(x) (5/(x*(5-x)))*exp((-1/7)*(log(x/(5-x)))^2)

# I use the optimize function to find M_U, the supremum of f*_X(x)/g_Y_U(x):
M_U <- optimize(fstar_gunif, c(0, 5), maximum=TRUE)$objective

# Evaluating f*_X(x) numerically using the in-built integrate function to find the
# normalizing constant nc:
int = integrate(fstar,lower=0,upper=5)[1]$value
nc = 1/int

thetaU = int/M_U # Evaluating the acceptance probability thetaU: 76.2%
print(thetaU)
```

```
## [1] 0.7619621
```

I now use my envelope Mg_Y as it better mimics the shape of f_X^* to show that we get a higher acceptance probability. This code works out the theoretical acceptance probability using the suitable envelope g_Y. We find that it has an acceptance probability of 87.9%.

```
g <- function(y) # Function to evaluate g_Y(x)
{ifelse(y<=5/2, -0.04*y+0.25, 0.04*y+0.05)}

fstar_g <- function(x) # Function to evaluate f*_X(x)/g_Y(x)
{ifelse(x<=5/2, fstar(x)/(-0.04*x+0.25),fstar(x)/(0.04*x+0.05))}

# I use the optimize function again to find M, the supremum of f*/g:
M <- optimize(fstar_g, c(0, 5), maximum=TRUE)$objective

# Evaluating the acceptance probability theta:
theta = int/M
print(theta)
```

```
## [1] 0.879276
```

The following code plots a graph to visualise and compare f_X^*, g_Y and Mg_Y. We can clearly see from Figure 1 that the envelope Mg_Y is valid as it is above the graph of f_X^* and its support is equal to the one of f_X^*.

```
# Obtain all the data required for the plots:
x=seq(0,5,l=1000)
fgx = data.frame(x=x, fx=fstar(x), gx=g(x), Mgx = M*g(x))
mylabs=list(expression(fstar[X](x)),expression(g[Y](x)),expression(Mg[Y](x)))

# Plots a comparison of f*_X, g_Y and Mg_Y:
p <- ggplot(fgx)
p + geom_line(aes(x,fx,colour="fx"))+
  geom_line(aes(x,gx,colour="fy"))+
  geom_line(aes(x,Mgx,colour="fz"))+
  labs(y="y", title=expression("Comparison of"~fstar[X](x)~"," ~g[Y](x)~"and"~Mg[Y](x)))+
  scale_colour_manual("", values=c("fx"="red","fy"="blue","fz"="black"), labels=mylabs)
```

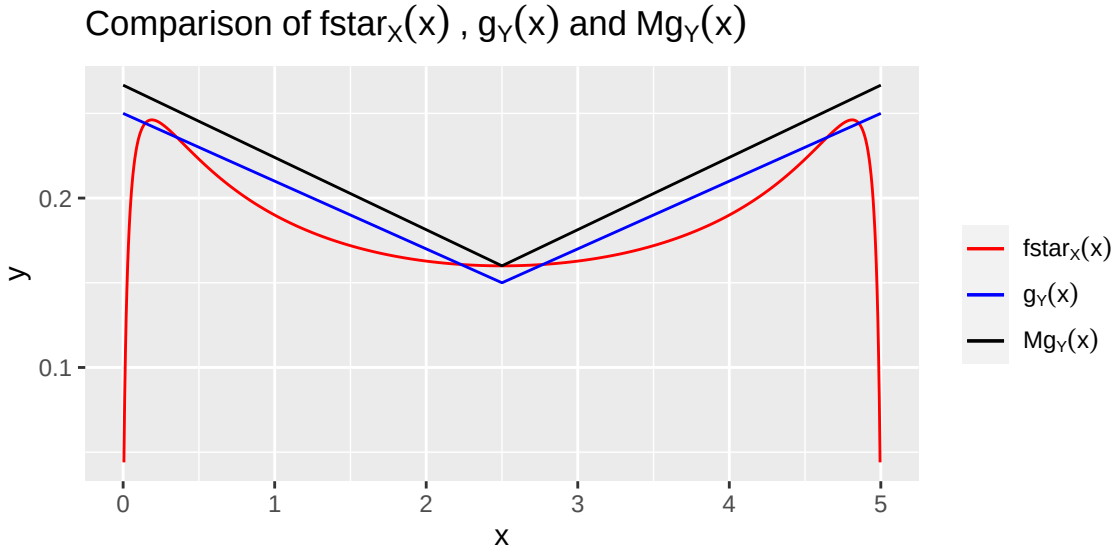


Figure 1: Graph comparing f*_X, g_Y and the envelope Mg_Y.

Step 2: Simulation from $g_Y(\cdot)$

To simulate from $g_Y(y)$, we will use inversion. Inversion is a simulation method that relies on the existence of $F_X(X)^{-1}$. If $X \backsim F_X(continuous) \implies U = F_X(X) \backsim U(0,1)$. I chose to simulate from inversion because g_Y is made up of linear functions which should be easy to integrate and find the inverses of.
Here is the general algorithm for inversion:

1. Simulate $U = u \sim U(0,1)$.
2. Set $X = F_X(U)^{-1}$.

Recall from above that we want to simulate from:

$$g_Y(x) = \begin{cases} -0.04x + 0.25 & \text{for } x \in (0, 2.5] \\ 0.04x + 0.05 & \text{for } x \in (2.5, 5) \end{cases}$$

I first look for the CDF:

$$\int_0^x (-0.04y + 0.25)dy = \left[\frac{-0.04y^2}{2} + 0.25y\right]_0^x = -0.02x^2 + 0.25x$$

$$\int_0^{2.5} (-0.04y + 0.25)dy + \int_{2.5}^x (0.04y + 0.05)dy = 0.5 + \left[\frac{0.04y^2}{2} + 0.05y\right]_{2.5}^x = 0.02x^2 + 0.05x + 0.25$$

Hence,

$$G_Y(x) = \begin{cases} -0.02x^2 + 0.25x & \text{for } x \in (0, 2.5] \\ 0.02x^2 + 0.05x + 0.25 & \text{for } x \in (2.5, 5) \end{cases}$$

And finally, I look for the inverse CDF by solving the two following equations for x using the quadratic formula $\frac{-b \pm \sqrt{(b^2 - 4ac)}}{2}$ to find the roots:

$$-0.02x^2 + 0.25x = y \implies 0.02x^2 - 0.25x + y = 0 0.02x^2 + 0.05x + 0.25 = y \implies 0.02x^2 + 0.05x + (0.25 - y) = 0$$

We get that: $x = 6.25 - 25\sqrt{0.0625 - 0.08y}$ and $x = -1.25 + 25\sqrt{0.0225 + 0.08y}$. I took the negative and positive square roots respectively because these roots will lie within $(0, 5)$.

Inversion Algorithm for $g_Y(\cdot)$

The final inversion algorithm is therefore:

1. Simulate $U = u \sim U(0,1)$.
2. If $u \leq 2.5$ then $x = 6.25 - 25\sqrt{0.0625 - 0.08u}$
   Else $x = -1.25 + 25\sqrt{0.0225 + 0.08u}$.

Step 3: Implementing the Rejection Algorithm

We can now finally get onto to rejection algorithm to generate from f_X given by:

1. Simulate $U = u_1 \sim U(0,1)$.

2. If $u_1 \leq 2.5$ then $y = 6.25 - 25\sqrt{0.0625 - 0.08u_1}$
   Else $y = -1.25 + 25\sqrt{0.0225 + 0.08u_1}$.

3. Simulate $U = u_2 \sim U(0,1)$

4. If $u_2 \leq \frac{1}{Mg_Y(y)y(5-y)} \exp\left\{\frac{-1}{7}\left(\log\left(\frac{y}{5-y}\right)\right)^2\right\}$, set $X = y$.

5. Otherwise GOTO 1.

Coding the Rejection Scheme

We are now ready to start the rejection algorithm which is what the following piece of code does. We get an empirical acceptance probability of 88.2% which is very close to our theoretical acceptance probability of 87.9%. This verifies that the choice of g_Y was valid for simulating from f_X^*. rds is what I call the function that performs rejection. I also use system.time to see how quick the algorithm is. I get a system.time of 0.011s when $n = 100000$. We will compare this time in the next section when adding a squeezing step to the algorithm.

```
trap <- function(u) # Function that generates from g using inversion
{
ifelse(u<=0.5, 6.25-25*sqrt(0.0625-0.08*u), -1.25+25*sqrt(-0.0175+0.08*u))
}

# Code for rejection sampling using Mg_Y as an envelope:
rds <- function(n){
  x <- vector()  # Simulated values in x
  p <- vector() # Estimated acceptance probabilities in p
  len_x = 0
  mult = 1/theta # 1/acceptance probability
  while(len_x <= n){
    n_to_gen = max((n-len_x)*mult,10) # Determine number to generate - not less than 10
    u1 = runif(n_to_gen)
    y=trap(u1)
    u2 = runif(n_to_gen)
    # Accept X=y if u <= f*(y)/Mg(y):
    cond <- u2 <= fstar(y)/(M*g(y))
    p <- c(p, sum(cond)/n_to_gen)
    x <- c(x, y[cond])
    len_x <- length(x)
  }
  cat("theoretical acceptance probability = ", theta, "\n")
  cat("acceptance probability = ", p, "\n")
  return(x[1:n]) # Return a data frame with column name "x"
}

x=rds(100000)
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8795047
```

```
x_rds <- data.frame(x=x)
system.time(rds(100000))
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8779044 0.834185 1
```

```
##    user  system elapsed
##   0.047   0.009   0.059
```

Pretesting-squeezing

In this section, we will be performing squeezing to improve the efficiency of our simulation above. The idea behind squeezing is to find two functions $W_L(x)$ and $W_U(x)$ which are inexpensive to calculate and that will act as a lower and upper bound for $\frac{f_X(x)}{g_Y(x)}$:

$$W_L(x) \leq \frac{f_X(x)}{g_Y(x)} \leq W_U(x)$$

The rejection algorithm will therefore be modified to:

1. Generate $Y = y \sim g_Y(y)$.
2. Generate $U = u \sim U(0, 1)$.
3. If $u \leq W_U(y)$, set $X = y$.
4. If $u \geq W_L(y)$, GOTO 1.
5. Otherwise $u \leq \frac{f_X(y)}{Mg_Y(y)}$, set $X = y$.
6. Otherwise GOTO 1.

In our case, I have picked $W_U(x) = 1.07$ and

$$W_L(x) = \begin{cases} 2.9x & \text{for } x \in (0, 0.3] \\ 0.87 & \text{for } x \in (0.3, 4.6] \\ -2.94x + 14.5 & \text{for } x \in (4.6, 5) \end{cases}$$

Hence, we will have:

1. Simulate $U = u_1 \sim U(0, 1)$.
2. If $u_1 \leq 2.5$ then $y = 6.25 - 25\sqrt{0.0625 - 0.08u_1}$
   Else $y = -1.25 + 25\sqrt{0.0225 + 0.08u_1}$.
3. Simulate $U = u_2 \sim U(0, 1)$
4. If $u_2 \leq 1.07$, set $X = y$.
5. If $u_2 \geq W_L(y)$, GOTO 1.
6. If $u_2 \leq \frac{1}{Mg_Y(y)y(5-y)} \exp\left\{\frac{-1}{7}\left(\log\left(\frac{y}{5-y}\right)\right)^2\right\}$, set $X = y$.
7. Otherwise GOTO 1.

The following code plots a graph to show that my chose $W_U(y)$ and $W_L(y)$ are valid (Figure 2).

```
W_U <- function(x) 1.07 # Upper squeeze is a uniform variable
W_L <- function(x) ifelse(x<=0.3, 2.9*x, ifelse(x<=4.7, 0.87, -2.9*x+14.5))
```

```r
x=seq(0,5,l=1000)
fgx = data.frame(x=x, fx=fstar_g(x), wu=W_U(x), wl=W_L(x))
mylabs=list(expression(fstar[X](x)),expression(W[U](x)),expression(W[L](x)))

# Plotting the upper and lower squeeze and f^*_X:
p <- ggplot(fgx)
p + geom_line(aes(x,fx,colour="fx"))+
  geom_line(aes(x,wu,colour="fy"))+
  geom_line(aes(x,wl,colour="fz"))+
  labs(y="y", title=expression("Comparison of"~fstar[X](x)~"," ~W[U](x)~"and"~W[L](x)))+
  scale_colour_manual("", values=c("fx"="red","fy"="blue","fz"="black"), labels=mylabs)
```

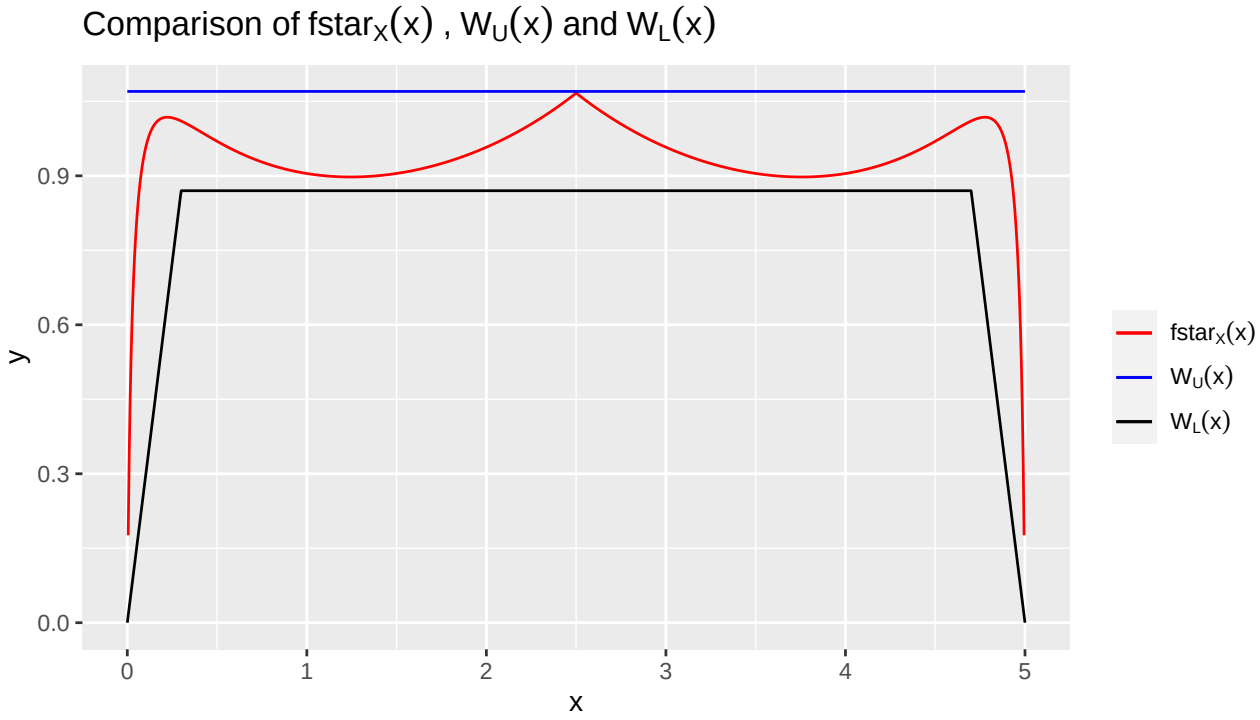


Figure 2: Adding squeezes to f*/g

Here is my rejection algorithm with the added squeezing conditions.
Although I am using easy-to-generate upper and lower squeezes (uniform and linear functions), my rejection algorithm with squeezing appears to be a slower. I obtain a system time of 0.024s. This could be because the R language is not optimised for ifelse statements so having too many if else statements would in fact take more time.

```r
trap <- function(u) # Function that generates from g using inversion
{
ifelse(u<=0.5, 6.25-25*sqrt(0.0625-0.08*u), -1.25+25*sqrt(-0.0175+0.08*u))
}

# Code for rejection sampling with squeezing:
rds_s <- function(n){
  x_s <- vector()  # Simulated values in x
  p_s <- vector() # Estimated acceptance probabilities in p
  len_x_s = 0
  mult_s = 1/theta # 1/acceptance probability
```

```
  while(len_x_s <= n){
    n_to_gen_s = max((n-len_x_s)*mult_s,10) # Determine number to generate
    u1_s = runif(n_to_gen_s)
    y_s=trap(u1_s)
    u2_s = runif(n_to_gen_s)
    # Adding the squeezing conditions:
    s_cond <- ifelse(u2_s<=W_L(y_s), TRUE,
                     ifelse(u2_s>W_U(y_s), FALSE,
                     ifelse(u2_s > (fstar(y_s))/(M*g(y_s)), FALSE, TRUE)))
    p_s <- c(p_s, sum(s_cond)/n_to_gen_s)
    x_s <- c(x_s, y_s[s_cond])
    len_x_s <- length(x_s)
  }
  cat("theoretical acceptance probability = ", theta, "\n")
  cat("acceptance probability = ", p_s, "\n")
  return(x[1:n])
}

x=rds_s(100000)
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8869082
```

```
system.time(rds_s(100000))
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8880952
```

```
##    user  system elapsed
##   0.069   0.025   0.098
```

Diagnostic Plots

We will now produce diagnostic plots that demonstrate the validity of my code for simulating from $f_X(x)$. The first diagnostic plot is a histogram. The purpose of a histogram is to display the distribution of a data set. I have chosen to use a histogram because they are preferable for numerical variables as opposed to bar-plots that are more relevant to categorical variables. I have also included a density plot of the true pdf to show how my simulated data behaves compared to the true pdf.
We see from the histogram (Figure 3) that the data closely follows the required theoretical distribution. There appears to be no outliers or gaps in the data with shows that our rejection algorithm (without squeezing) works well. The following code plots the histogram and a density plot of the true pdf.

```
library(ggplot2)

# Code for the histogram:
mylabs=list(expression(f[X](x)))
cols = c("fy"="blue")
x=x_rds$x
x_plot = cbind(x_rds, fy = nc*(1/(x*(5-x)))*exp((-1/7)*(log(x/(5-x)))^2))
p <- ggplot(x_plot)
 p+ labs(y="density",title="Histogram and true pdf") +
  geom_histogram(aes(x,y= ..density..),breaks=seq(0, 5, l=80))+
  geom_line(aes(x,fy,colour="fy")) +
   scale_colour_manual(name="",values=cols, labels=mylabs)
```

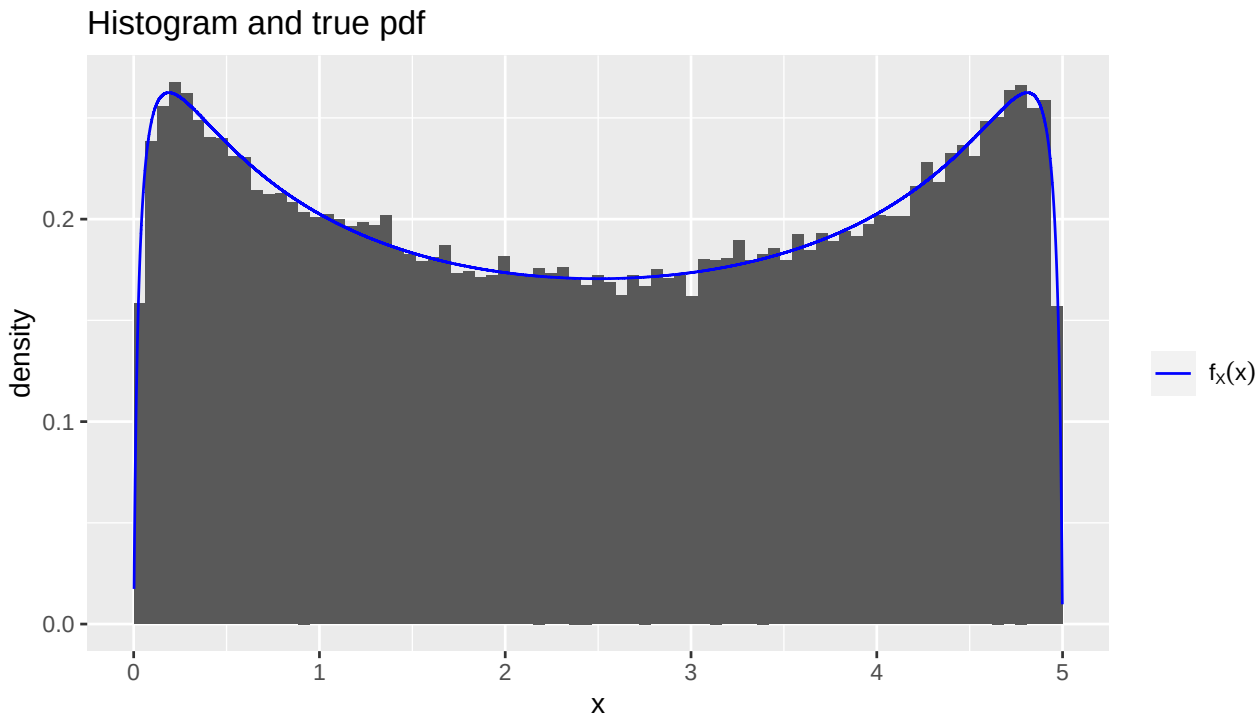


Figure 3: Histogram.

One draw back of the histogram is that it does not allow us to assess the independence of the data generated, hence we plot the auto-covariance sequence and the lag plot to verify independence. The auto-covariance sequence plot (Figure 4) shows no signs of significant correlation at any lag. The lag scatter plots (Figure 5) of $F_X(x_i)$ (the cdf evaluated at the data values) shows a random scatter, which further supports that the data are generated independently from $f_X(\cdot)$ (because if $X \sim f_X(x)$ then $F_X(X) \sim U(0,1)$). This piece of code plots two diagnostic plots: the autocovariance sequence and lag plots. For this, I needed to calculate the CDF which was done using the integrate and sapply function.

```
# Function to evaluate f_X(x)
f <- function (x) nc*(1/(x*(5-x)))*exp((-1/7)*(log(x/(5-x)))^2)

x = rds(500)
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8704833 0.8
```

```
x_rds = data.frame(x=x)

# Theoretical cdf
cdf_int <- function(x) (integrate(f,lower=0,upper=x)[1])$value
cdf <- function (x) sapply(x,cdf_int)

# Code to plot the auto-covariance sequence of generated data and the lag plots
ggAcf(x_rds)+
  labs(title="Autocovariance sequence of generated data")
```

```
gglagplot(cdf(x_rds$x), lags=4, do.lines=FALSE)+
  labs(title=expression("Lag Plots of " ~ F[X](X)))
```

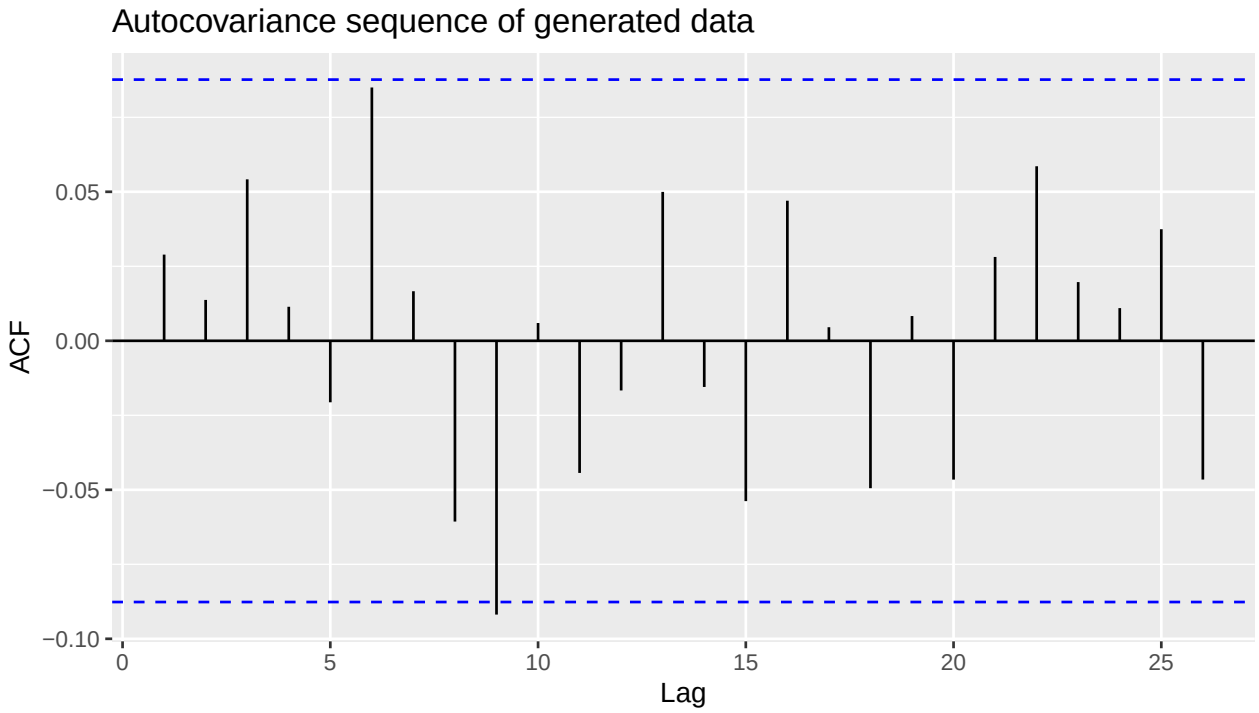
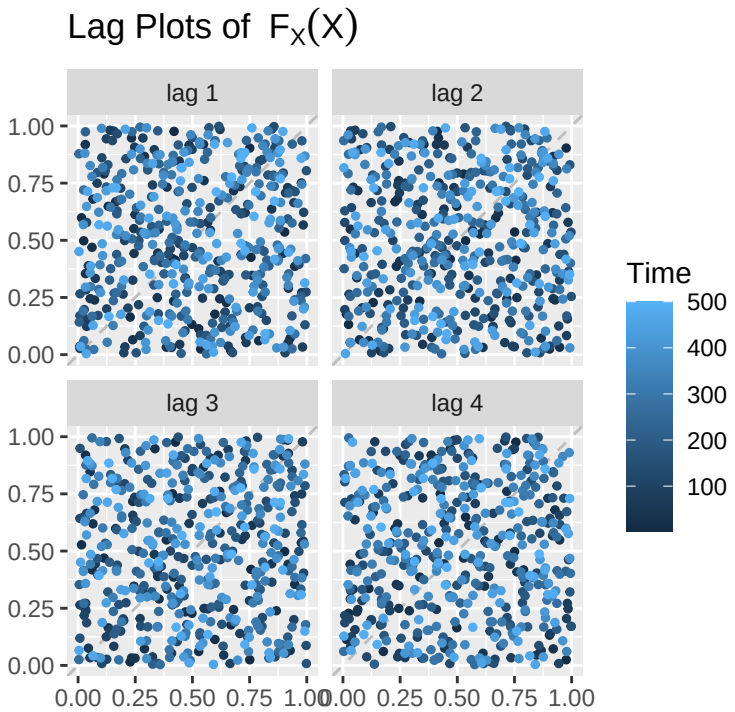


Figure 4: Autocovariance sequence of generated data and Lag plots.



The last diagnostic plot is a QQ-plot. For this we need the theoretical quantile function. The code below is to generate a QQ-plot when you can't calculate F(x) analytically. From Figure 6, there appears to be no outliers and the QQ-plot displays that the sample resembles the desired distribution very well.

```r
library(pracma,ggplot2)

# Generate some x from our probability density function
n=1000
x=rds(n)
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8757589 0.9
```

```r
x_plot = data.frame(x=x)

# Function which uses numerical integration to calculate F(p) - q
mycdf = function(p,q=0){
  integrate(f,0,p)$value-q
}

# Solving F(x) - q = 0 to calculate the quantiles
qfunc <- function(q){
  # Set lower as 0+10^(-5) and upper as 5-10^(-5) as it is close to where F_X(x) = 1
  bisect(mycdf, 0+10^(-5), 5-10^(-5),q=q)$root
}

# Making sure it works on a vector and returns a vector
qhn1 <- function(p) unlist(lapply(p, qfunc))

ggplot(x_plot, aes(sample=x))+
  labs(title="Empirical against theoretical quantiles")+
   stat_qq(distribution=qhn1,dparams=list()) +
   stat_qq_line(distribution=qhn1,dparams=list())
```

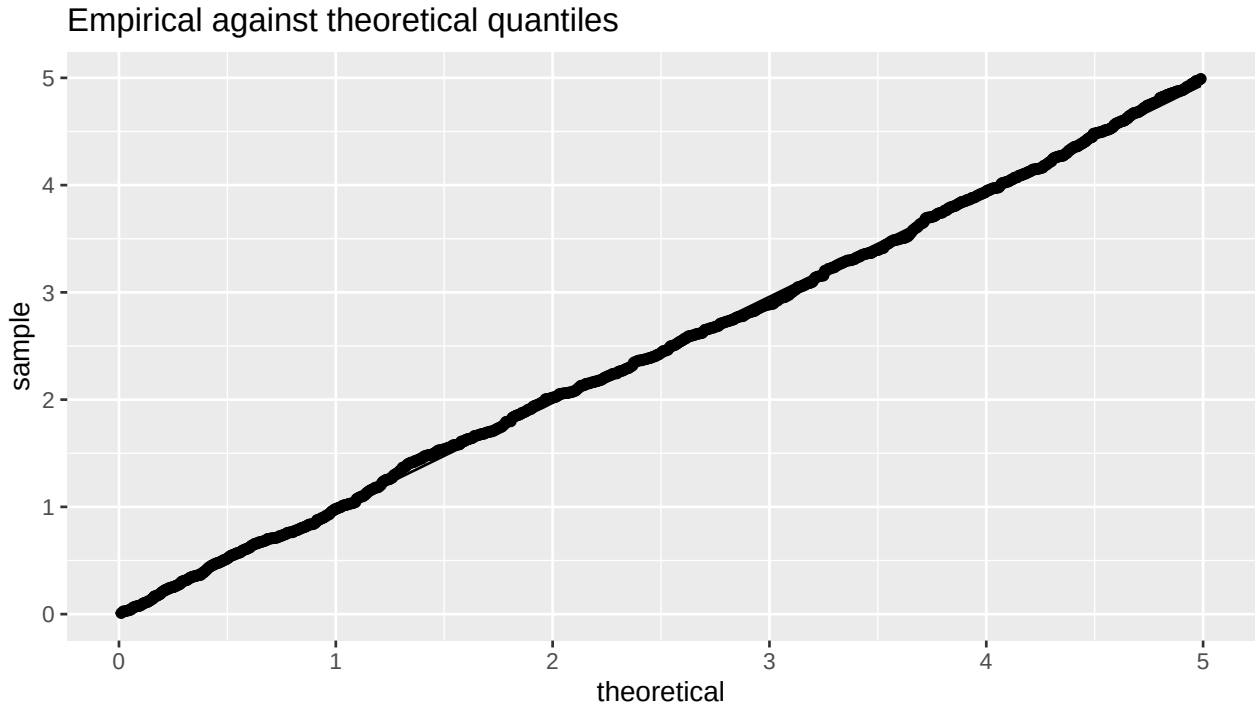


Figure 5: QQ-plot.

Goodness of Fit Tests

We will now be conducting two goodness of fit tests (statistical test): the K-S test and the Chi-squared test to evaluate how well our simulated data matched our distribution.
Our null hypothesis H_O is that the data are drawn from the distribution F_X. i.e. $F_{X_n}(x) = F_X(x)$ in distribution. The alternative hypothesis is H_a: $F_{X_n}(x)$ is different from $F_X(x)$. We test our null hypothesis against the alternative hypothesis.
The significance level for this test is $\alpha = \text{P(Type I error)} = \text{P(incorrectly rejecting } H_O)$. I chose $\alpha = 0.05$.
We use the p-value to choose whether we support or reject our null hypothesis: if $p < \alpha$ then I reject H_O and if $p > \alpha$ then I accept H_O.

Kolmogorov-Smirvov Test

We will perform a Kolmogorov-Smirnov test for some simulated data with varying values of n. The Kolmogorov-Smirnov (K-S) test is used to test the difference between the empirical cumulative distribution function (cdf), $F_{X_n}(x)$ evaluated from the data and the theoretical cumulative distribution function, $F_X(x)$. We performed the K-S test over 4 values of n: n=100, 1000, 5000 and 10000.

```
# Simulation study of KS test, using data set length stored in n_vals
n_vals = c(100,1000,5000,10000)
nn_vals = length(n_vals)
m = 100 # Taking m simulations of each length

ks.testrds <- function(x){
  kstestx = ks.test(x,cdf)  # cdf is the theoretical cdf from above
  return(c(kstestx$p.value, kstestx$statistic))
}
ks.results=data.frame()

for(n in 1:nn_vals){
  n1=n_vals[n]
  x <- matrix(0, nrow=n1, ncol=m)
  # One call to rds, then split into matrix in order to use the apply function
  x1 = rds(n1*m)
  for(i in 1:m)  x[,i] <- x1[((i-1)*n1+1):(i*n1)]
  ks.testx= apply(x,2,ks.testrds)
  ks.results = rbind(ks.results, data.frame(p.value=ks.testx[1,], D = ks.testx[2,],
                                            N=rep(n1,m)))
}
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8839362
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.878599 0.879276 0.8
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8794484
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.8792277 0.8952629
```

We expect to find uniformly distributed p-values as this will show that the data is generated from the theoretical distribution. We use box plots and violin plots to show the distribution is uniform.

The violin plot (Figure 6) is combination of a box plot and a density plot. Firstly, in a box-plot, we are able to see a lot of features: minimum and maximum value of the data, interquartile range (inc. 25th and 75th percentile), median, outliers. One of the disadvantages of the box plot however is that the underlying distribution is hidden - the shape of the distribution cannot be drawn immediately from the box-plot. The

violin plot is useful in that we also have the shape of the density through the density plot. We can see that as the sample size n increases, the D statistics decreases. This is expected as D is defined as $D = sup(F_n(x) - F(x))$.

The frequency polygons (Figure 8) enables us to plot several histogram on one chart without it begin overcrowded. The bars in the histogram are replaced by lines connecting the heights of the "bins" (classes). It is preferable to use frequency polygons in comparison to using transparency options for a histogram (it can look a bit messy). They are useful for comparing various distributions.

```
ggplot(ks.results, aes(factor(N), D))+
  geom_violin()+
  geom_boxplot(width=0.05)+
  labs(title="Distributions of the K-S test statistics")
```

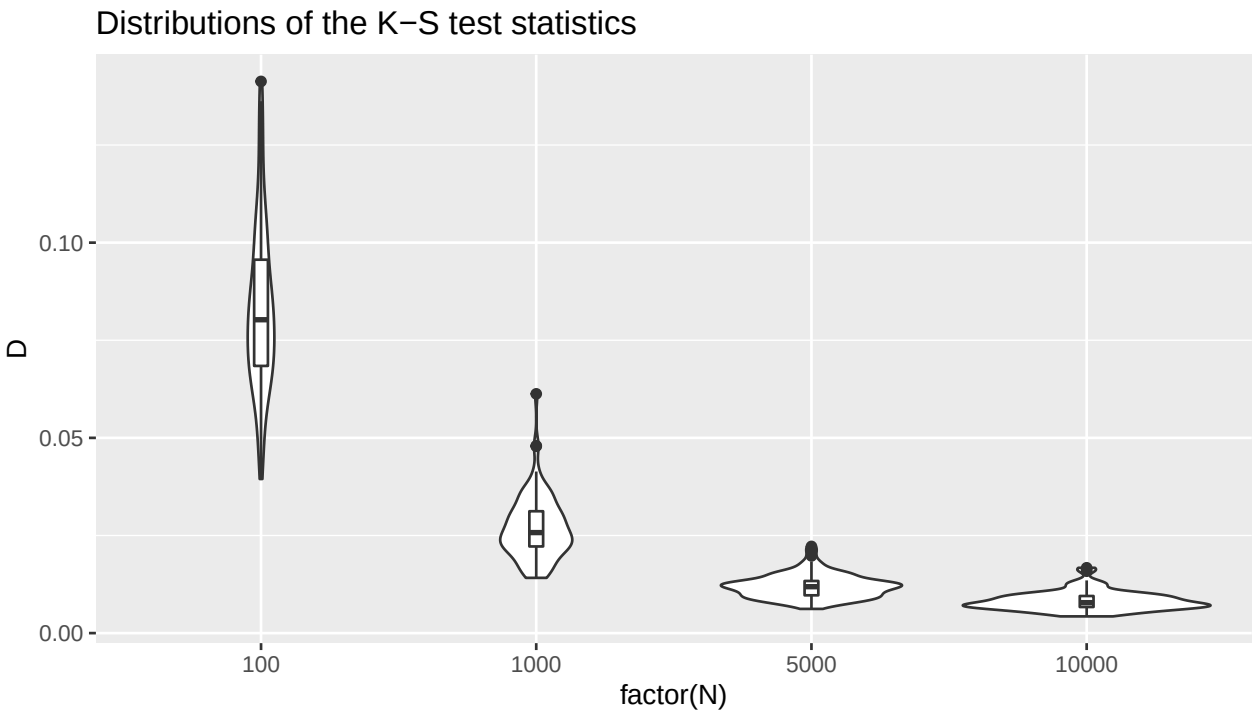


Figure 6: Violin plot, boxplot and frequency polygon from the K-S test.

```
ggplot(ks.results, aes(factor(N), p.value))+
  geom_violin()+
  geom_boxplot(width=0.1)+
  labs(title="Distributions of the K-S p-values")
```

```
ggplot(ks.results, aes(p.value,colour=factor(N)))+
  geom_freqpoly(breaks=seq(0,1,0.1))+
  labs(title="Frequency polygons of p-values")
```

```
library(knitr)
#options(dplyr.summarise.inform = FALSE)
ks.table <- ks.results %>% group_by(N) %>% summarise("Mean p-value" = round(mean(p.value),digits=3), "S
kable(ks.table)
```

N	Mean p-value	Std Dev (p)	Mean D	Std Dev (D)
100	0.529	0.259	0.083	0.021

N	Mean p-value	Std Dev (p)	Mean D	Std Dev (D)
1000	0.515	0.280	0.027	0.008
5000	0.509	0.262	0.012	0.003
10000	0.549	0.278	0.008	0.002

We can see from the results table above that as the values of N increases, the value of D decreases. Also, as we expect, the distribution of the p-values is roughly uniform as should be expected under the null hypothesis. This information displays no evidence to reject our null hypothesis. Our rejection algorithm works well to simulate from f_X^*.

Chi-Squared Test

Similarly, we can use the Chi-Squared Test to evaluate the fit between the simulated values and the theoretical ones. Below is the code for my test. I have first split my interval into 10 bins and stored the number observed in each bin in y. I then calculated the prob of being in each bin by integration of the pdf and stored the values in the vector p. I then used the inbuilt function chisq.test(x, p) in R.

```
f <- function (x) nc*(1/(x*(5-x)))*exp((-1/7)*(log(x/(5-x)))^2)

n=1000
x = rds(n)
```

```
## theoretical acceptance probability =  0.879276
## acceptance probability =  0.859932 0.879276 0.7
```

```
x = data.frame(x=x)
n_b = 10 # Number of bins
chi <- function(n_b) {
    bins = seq(0,5, l=n_b+1) # Creating N bins
    x$bins <- cut(x$x,breaks=bins) # Bin column
    y = count(x,bins)$n # Observed frequencies
    p <- vector() # Store probabilities
  for (i in 1:n_b) {
    # Prob of being in each bin:
    p <- c(p,integrate(f, bins[i], bins[i+1])$value)}
    results= chisq.test(x=y, p=p, rescale.p=TRUE)
}

print(chi(10))
```

```
##
##  Chi-squared test for given probabilities
##
## data:  y
## X-squared = 13.409, df = 9, p-value = 0.1449
```

The value for alpha I had set earlier was 0.05 and clearly here the p-value of 0.3445 is greater hence we accept our null hypothesis.

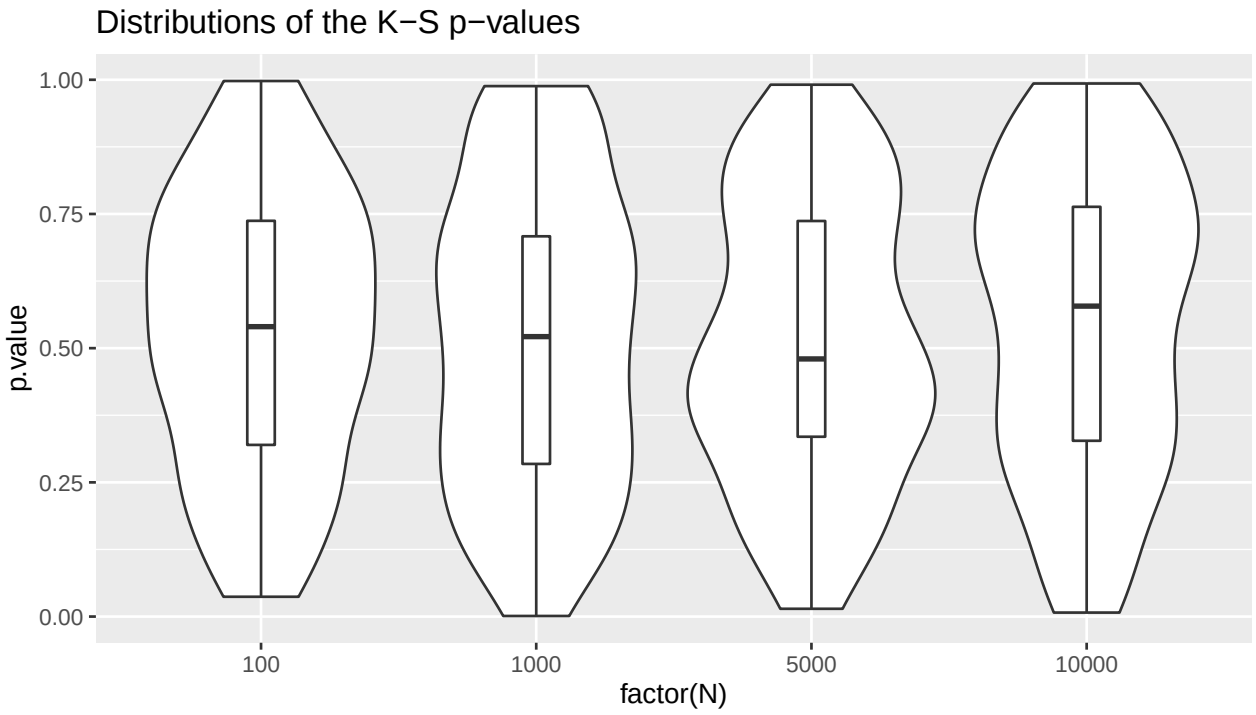


Figure 7: Violin plot, boxplot and frequency polygon from the K-S test.

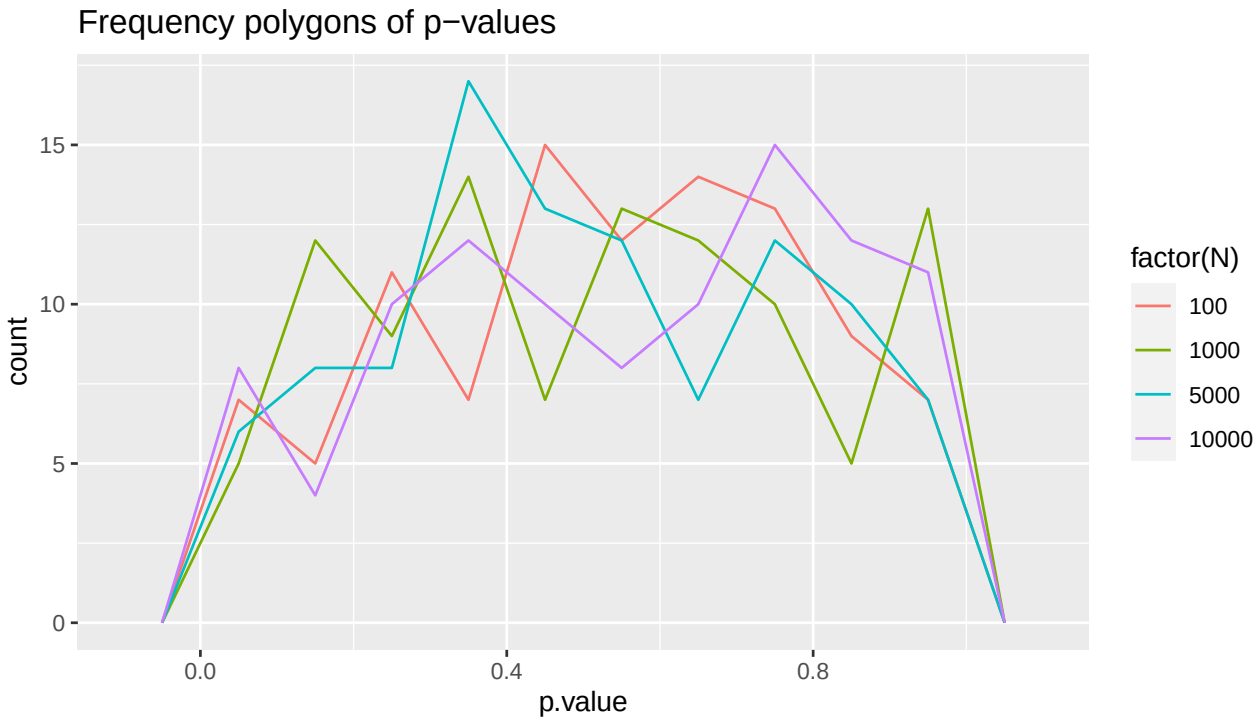


Figure 8: Violin plot, boxplot and frequency polygon from the K-S test.

Estimating the normalizing constant

Crude Monte Carlo with Importance Sampling

Crude Monte Carlo is a technique where we wish to estimate the value of an integral $\theta = \int h(x)dx$ because it is not analytically possible. We can rewrite our integral as $\theta = \int \phi(x) f_X(x)dx$ where $f_X(x)$ is a pdf and $\phi(x) = \frac{h(x)}{f_X(x)}$. We require the ability to sample from $f_X^*(x)$ which we know is possible here by the rejection scheme above and that that $X_1, X_2, ..., X_n \sim f_X(x)$ are iid. Effectively, what we will be doing is importance sampling.

Importance Sampling

We suppose that we can sample from another distribution $g_Y(x)$ and we can pointwise evaluate $f_X^*(x)$ and $g_Y(x)$ For importance sampling, we look for a $g_Y(x)$ that mimics $f_X^*(x)$.

$$\theta = \int \phi(x) f_X(x)dx = \int \phi(x)\frac{f_X(x)}{g_Y(x)} g_Y(x)dx = \int \phi(x)W(x)g_Y(x)dx = E_{g_Y}[\phi(X)W(x)]$$

I therefore pick my $g_Y(x)$ to be

$$g_Y(x) = \begin{cases} -0.04x + 0.25 & \text{for } x \in (0, 2.5] \\ 0.04x + 0.05 & \text{for } x \in (2.5, 5) \end{cases}$$

and set $\phi(X_i) = 1$
Given iid $X_1, X_2, ..., X_n \sim g_Y(\cdot)$, the estimator for θ will be:

$$\hat{\theta}_{IS} = \frac{1}{n}\sum_{i=1}^{n} W(X_i)\phi(X_i)$$

We can directly sample from g_Y using inversion as we have done earlier.
Now we show the variance of the importance sampling estimator $\hat{\theta}_{IS}$ is:

$$Var(\theta_{IS}) = \frac{1}{n^2} nVar[W(X)\phi(X)] = \frac{1}{n} Var[W(X)\phi(X)] = \frac{1}{n}[E_g[W^2(X)\phi^2(X)] - \theta^2]$$

The following code does importance sampling. When $n = 100000$, we find that $\theta = 0.9375831$ which is very close to the true integral calculated using integrate: 0.9378944.

```
fstar <- function(x) (1/(x*(5-x)))*exp((-1/7)*(log(x/(5-x)))^2)

# Function to evaluate (f*)^2/g:
fstar_g_squared <- function(x)
{ifelse(x<=5/2, ((((1/(x*(5-x)))*exp((-1/7)*(log(x/(5-x)))^2)))^2/(-0.04*x+0.25)), ((((1/(x*(5-x)))*exp

# Importance sampling:
n = 100000 # Number of iterations - create iid samples from our proposal distribution g_Y
u1 <- runif(n,0,1)
X <- trap(u1) # Generate from g_Y
thetahat_is = sum(fstar_g(X))/n
print(thetahat_is)
```

```
## [1] 0.9377383
```

```
theta = integrate(fstar,0,5)[1]$value

a=integrate(fstar_g_squared,0,5)[1]$value

var_thetahat_is = (a-theta^2)/n # Variance of the importance sampling estimate
print(var_thetahat_is)
```

```
## [1] 7.559997e-08
```

Hit-or-Miss Monte Carlo

Now we compare the Crude Monte Carlo method above to the Hit-or-Miss Monte Carlo method. Let $h(x)$ be a bounded function on (a, b) with $0 \leq h \leq c$. Similarly to Monte Carlo, we seek to evaluate the integral

$$\theta = \int_a^b h(x)dx = (b-a)\int_a^b h(x)f(x)dx$$

Where $f(x) = \frac{1}{b-a}$

Hit-or-Miss Monte Carlo algorithm

1. Simulate $U_i = u_i \sim U(a, b)$ for $i = 1, ..., n$
2. Simulate $V_i = v_i \sim U(0, c)$ for $i = 1, ..., n$
3. Define $\hat{\theta} = c(b-a)\sum_{i=1}^{n} I(v_i \leq h(u_i))$ to be the estimator of θ.

The properties of $\hat{\theta}$ are as follows:
Expectation: $E(\hat{\theta}) = \theta$ (it is an unbiased estimator)
Variance: $Var(\hat{\theta}) = \frac{\theta}{n}[c(b-a) - \theta]$

Hence we have:

$$\theta = \int_0^5 f_X^*(x)dx = 5\int_0^5 \frac{1}{x(5-x)} exp\left\{\frac{-1}{7}(\log(\frac{x}{5-x}))^2\right\}\frac{1}{5}dx$$

And the algorithm will be:

1. Simulate $U_i = u_i \sim U(0, 5)$ for $i = 1, ..., n$
2. Simulate $V_i = v_i \sim U(0, c)$ for $i = 1, ..., n$ where c is the maximum of f_X^*
3. Define $\hat{\theta} = 5c\sum_{i=1}^{n} I(v_i \leq f_X^*(u_i))$ to be the estimator of θ.

and the variance will be: $Var(\hat{\theta}) = \frac{\theta}{n}[5c - \theta]$

The code below is for the Hit or Miss method:

```
fstar <- function(x) (1/(x*(5-x)))*exp((-1/7)*(log(x/(5-x)))^2)

n = 10000
c = optimize(fstar, c(0, 5), maximum=TRUE)$objective
# Generate random numbers uniformly inside box c(b-a)
u = runif(n,0,5)
v = runif(n,0,c)
thetahat_hm = 5*c/n*sum(v<=fstar(u)) # Calculates theta hat for HoM
print(thetahat_hm)
```

```
## [1] 0.9400336
```

```
var_theta = theta/n*(5*c-theta) # Variance of theta hat
print(var_theta)
```

```
## [1] 2.748025e-05
```

With the Hit or Miss method, we approximate θ as 0.936341 when the true value is 0.9378944. This is slightly off compared to using importance sampling. When we compare the variance of Hit or Miss to the variance from importance sampling, we can clearly see that importance sampling yields a smaller variance. When $n = 10000$, we obtain a variance of $7.559997e-08$ with importance sampling and $2.748025e-05$ when performing Hit-or-Miss. We can clearly see that the variance is smaller when using importance sampling hence it is a better method to use.

Conclusion

To conclude, g_Y was a valid choice to construct the envelope and this was confirmed through a series of diagnostic plots and statistical tests. An interesting extension to this piece of work would be to code this script using another language to see whether the squeeze speeds up the rejection scheme.

References

E. McCoy. Stochastic Simulation, Lectures Notes, Imperial College London, 2020. 2020. R.C. Reimann, P. Filzmoser, R. G. Garrett, R. Dutter. Statistical Data Analysis Explained: Applied Environmental Statistics.John Wiley & Sons. 2008