Brief Stochastic Simulation Study

George Hilton

30/11/2020

Introduction

My CID is 01494579, written as a base 10 number, this is 1494579. $1494579 = 3.131.3803$ which gives my values for a, b, c, d as described in the provided document. I have $a = 0, b = 1, c = 9, d = 0.3$.

Thus, the aim of this Coursework is to consider the probability density function,

$$f_X(x) \propto \tilde{f_X}(x) = \frac{1}{x(1-x)} exp(\frac{-1}{9}(0.3 + log(\frac{x}{1-x}))^2) \quad x \in (0,1).$$

I will use the rejection algorithm to sample from $X \sim f_X$, performing multiple diagnostic tests to ensure the sample is valid. In addition, I will use Monte-Carlo integration to compute the normalising constant of the PDF.

1

To sample from $X \sim f_X$, using rejection, we must find a probability distribution $g_Y(y)$, such that the range of X is a subset of its range. In addition, for some $M \in \mathbb{R}$, $Mg_Y(x) \geq f_X(x)$, $\forall x \in (0,1)$. We simply define $M = \sup_x \frac{f_X(x)}{g_Y(x)}$. Given these functions we can now understand the General Rejection Algorithm.

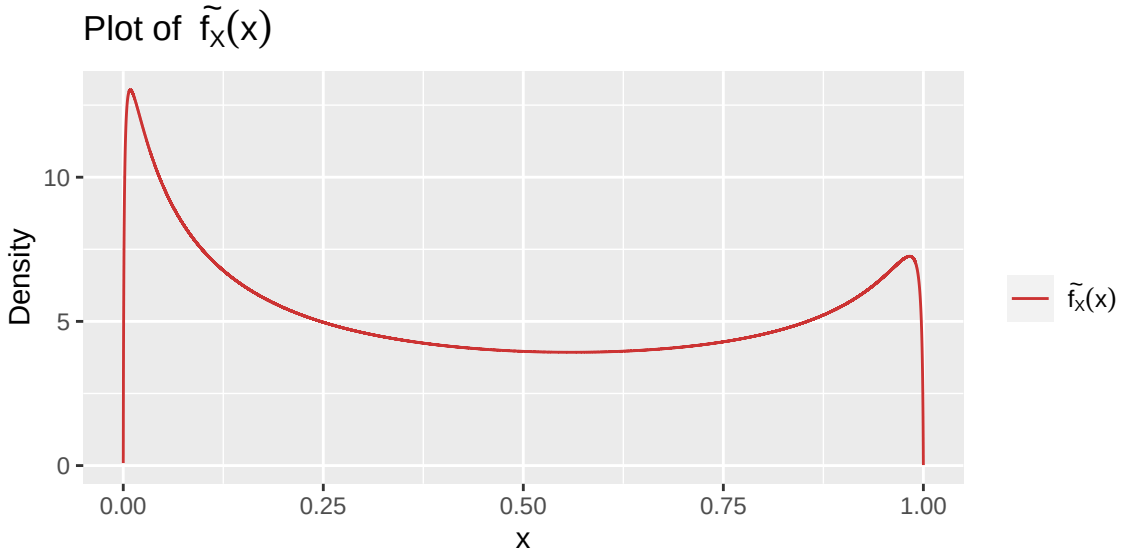
General Rejection Algorithm:

1. Generate $U = u \sim U(0,1)$ for the accept/rejection.
2. $Y = y \sim g_Y(y)$.
3. If $u \leq \frac{f_X(y)}{Mg_Y(y)}$, set $X = y \sim f_X(\cdot)$.
4. Otherwise, go to 1.

There is no requirement to compute the normalising constant of the PDF we want to sample from (or even the envelope function, but this is not relevant here), it suffices to sample from $\tilde{f_X}(x)$. Thus, we will be using $\tilde{f_X}(x) = \frac{1}{x(1-x)} exp(\frac{-1}{9}(0.3 + log(\frac{x}{1-x}))^2) \propto f_X(x)$ and $g_Y(y)$. But, the question of which distribution to choose for $g_Y(y)$ remains. To assist with selecting it, I plotted the curve of $\tilde{f_X}(x)$.

```
ftilde <- function(x){ #defining f~ as mentioned above
  (1/(x*(1-x)))*exp((-1/9)*(0.3+log(x/(1-x)))^2)
}

#create a function to perform the plotting as it is not computationally
```

```r
#expensive and saves storing many unnecessary global variables.
plotftilde <- function(){
x=seq(0,1,l=100000) #creating x values
df = data.frame(x=x, fn=ftilde(x)) #adding x values and f~(x) to the data frame
mylabels=list(expression(tilde(f[X])(x)))

#plotting the graph of f~
p <- ggplot(df)
p + geom_line(aes(x,fn,colour="fn")) +
  labs(y="Density", title=expression("Plot of "~tilde(f[X])(x)))+
  scale_colour_manual("", values=c("fn"="#CC3333"), labels=mylabels)
}
plotftilde()
```



We notice that this curve has very similar shape to a beta distribution with parameters $0 < \alpha, \beta < 1$ - so we want $Y \sim Beta(\alpha, \beta)$. Now, we are faced with the problem that we have three unknowns, M, α, β and want $Mg_Y(x) \geq \tilde{f}_X(x) \quad \forall x \in (0,1)$. As the form of $\tilde{f}_X(x)$ is complex, the analytical computation of $M = \sup_x \left[\frac{\tilde{f}_X(x)}{g_X(x)}\right]$ is troublesome. In addition, I also had to consider the restrictions of the task, i.e. the only permitted built in function (during simulation) is *runif*. Initially, I used *optimize* to construct M, α, β in the most efficient manner (I have excluded this calculation due to the page limit). This optimisation gave results of $M \approx 6, \alpha \approx 0.5, \beta \approx 0.7$. However, I could not find a method to generate from this beta distribution only using *runif*. As such, I felt a good compromise was to fix $\alpha, \beta = 0.5$ as I devised a method to generate from this distribution then, use *optimize* to find M.

The code chunk below calculates M and displays the results in a table.

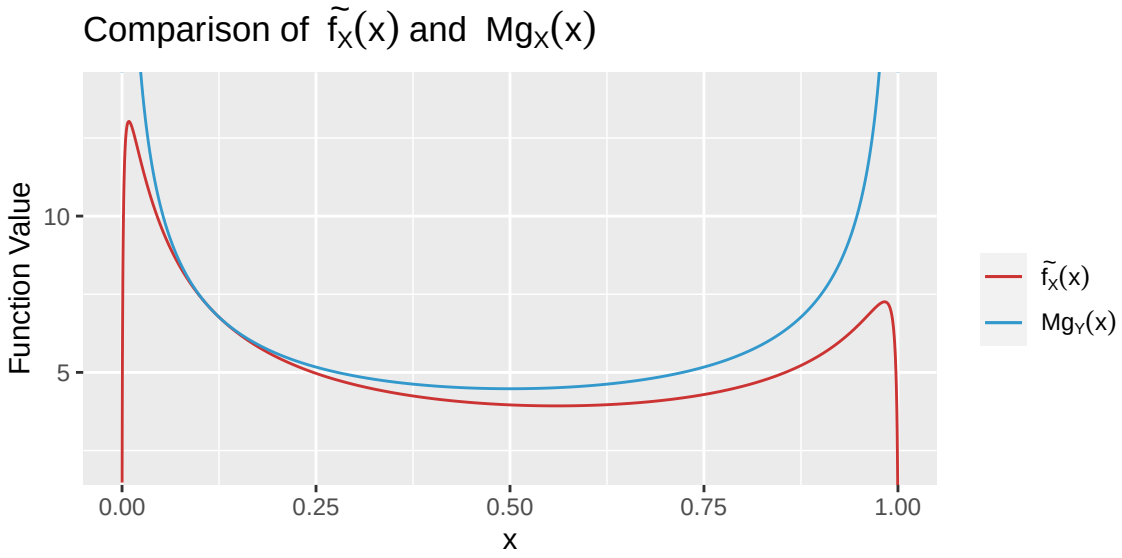
```r
#fixing a and b
a <- 0.5
b <- 0.5

foverg <- function(x){#f~/g which is used in the optimisation of M.
  ((1/(x*(1-x)))*exp((-1/9)*(0.3+log(x/(1-x)))^2))/((x^(a-1)*(1-x)^(b-1)*factorial(a+b-1))
    /(factorial(a-1)*factorial(b-1)))
```

```
}

optimiser <- function(){#function to output M
  optimize(foverg, lower=0, upper=1, maximum=TRUE)$objective}

#storing the results as global variables for later use.
M <- optimiser()

#displaying the results in a table
opresults <- function(){
  op.results <- cbind(M, a, b)
  colnames(op.results) <- c("M", "Alpha", "Beta")
  op.table <- op.results
  kable(op.table) %>% kable_styling(position="center")
}

opresults()
```

M	Alpha	Beta
7.032804	0.5	0.5

After this computation it is important to take a brief sanity check by plotting the curves of $Mg_Y(x)$ and $f_X(x)$ to ensure they are as we would expect. The code chunk below does this.

```
#construct g_y(x)
g <- function(x){
  (x^(a-1)*(1-x)^(b-1)*factorial(a+b-1))/(factorial(a-1)*factorial(b-1))
}
#function to perform the plots, it is computationally inexpensive and prevents
#unnecessary global variables
plot1 <- function(){
  x=seq(0,1,l=10000)
  df = data.frame(x=x, fn1=ftilde(x), fn2=M*g(x))
  mylabels=list(expression(tilde(f[X])(x)),expression(Mg[Y](x)))

  #plotting the graph of the pdf and its envelope function
  p <- ggplot(df)
  p + geom_line(aes(x,fn1,colour="fn1"))+
  geom_line(aes(x,fn2,colour="fn2"))+
  coord_cartesian(xlim = c(0, 1), ylim= c(2,14)) +
  labs(y="Function Value",
    title=expression("Comparison of "~tilde(f[X])(x)~"and "~Mg[X](x)))+
  scale_colour_manual("", values=c("fn1"="#CC3333","fn2"="#3399CC"), labels=mylabels)
}

plot1()
```



The graph plotted reveals that the 'envelope' function, $Mg_Y(x)$, is greater than $f_X(x)$ for $x \in (0,1)$ and a similar shape; this is what we want.

Before we define the final rejection algorithm there are several other things to consider: generating values from the beta distribution for the rejection scheme and squeezing methods.

To sample from the beta distribution I devised a method using a combination of Normal, Chi-Squared and Gamma distributions. Firstly note,

$$X_i \overset{iid}{\sim} N(0,1), \quad i = 1, ..., n \implies Z = \sum_{i=1}^{n} X_i^2 \sim \chi_n^2$$

also,

$$Z = \chi_n^2 \sim Gamma(\frac{n}{2}, 2)$$

and finally,

$$X \sim Gamma(a,c), \quad Y \sim Gamma(b,c) \implies \frac{X}{X+Y} \sim Beta(a,b)$$

As we want $Y \sim Beta(0.5, 0.5)$, we can reverse engineer the values for a, b, c, n above. Setting $a = b = 0.5$ gives $c = 2$ and $n = 1$. So we can generate randomly from a Beta distribution if, we can sample from a standard Normal only using random uniforms. The Box-Muller method does just that. So we have our algorithm to generate from $Beta(0.5, 0.5)$:

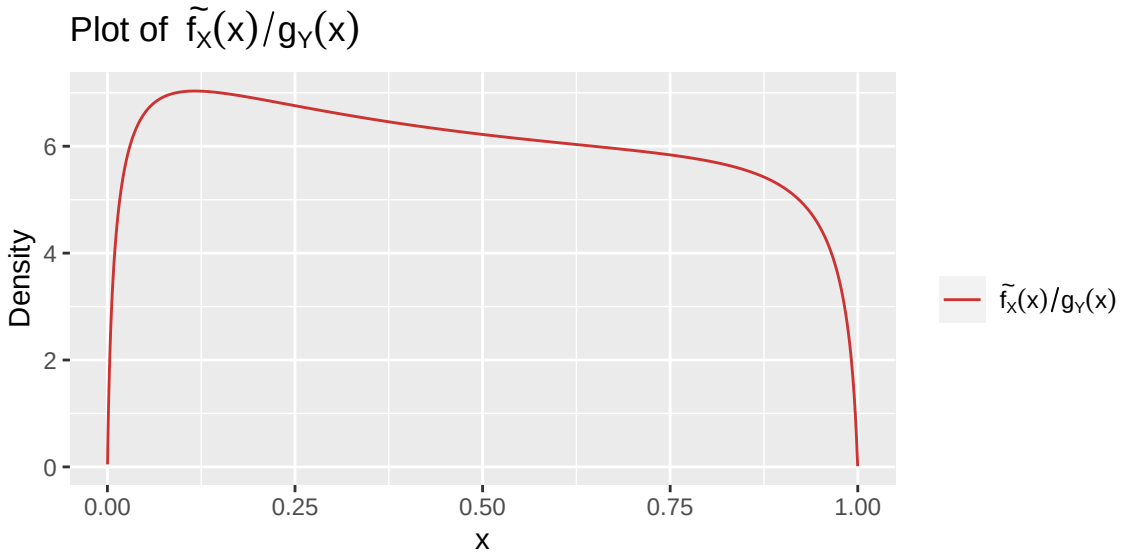
1. Using Box-Muller, generate two standard normal samples.
2. Square both the normal samples so they become Chi-Squared/Gamma variables.
3. Combine the variables as shown above to create Betas.

The code chunk below implements this algorithm.

```
boxmuller <- function(n){ #boxmuller function to generate normals
  m = ceiling(n/2)
  u = runif(m)
  v = runif(m)
  r = (-2*log(u))^(0.5)
  a = 2*pi*v
  x = r*cos(a)
  y = r*sin(a)
  return(c(x,y)[1:n])
}

beta05 <- function(n){
  n1 = boxmuller(n)
  n2 = boxmuller(n)
  x1 = n1^2
  x2 = n2^2
  y = x1/(x1+x2)
}
```

Next we consider the squeezing pre-test method. The fundamental aim of squeezing is to reduce the number of random points we test using the complex function $\frac{\tilde{f_X}(x)}{g_X(x)}$. To do this, we construct two functions: $W_L(x)$ and $W_U(x)$ such that $W_L(x) \leq \frac{\tilde{f_X}(x)}{g_X(x)} \leq W_U(x)$. From this, we can alter the final rejection algorithm to include this squeezing method; I will define this method later on. Firstly, let's plot $\frac{\tilde{f_X}(x)}{g_X(x)}$ to motivate some ideas for $W_L(x)$ and $W_U(x)$.

```
foverg_complete <- function(x){ #construction of f~/g
 ftilde(x)/g(x)
}
#create a function to perform the plotting as it is not computationally expensive
#and saves storing many unnecessary global variables.
plotfoverg <- function(){
  x=seq(0,1,l=10000) #creating x values
  df = data.frame(x=x, fn=foverg_complete(x)) #adding x values and f~(x) to the data frame
  mylabels=list(expression(tilde(f[X])(x)/g[Y](x)))
  #plotting the graph of f~
  p <- ggplot(df)
  p + geom_line(aes(x,fn,colour="fn")) +
  labs(y="Density", title=expression("Plot of "~tilde(f[X])(x)/g[Y](x)))+
  scale_colour_manual("", values=c("fn"="#CC3333"), labels=mylabels)
}

plotfoverg()
```

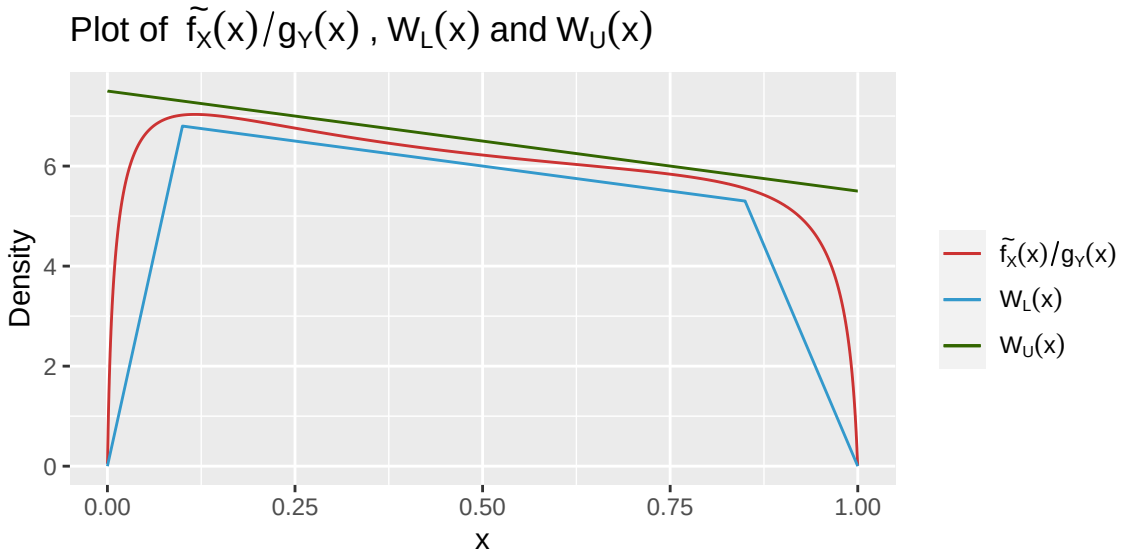


From the plot we can define fairly simple functions for $W_L(x)$ and $W_U(x)$ which satisfy our conditions. I used basic geometry to construct $W_L(x)$ as a combination of lines beneath the curve and $W_U(x)$ as a linear function the top of the curve. This is implemented below.

```
wl <- function(y){ifelse(0 <= y & y < 0.1, 68*y, ifelse(0.1 <= y & y < 0.85, -2*y + 7,
  ifelse(0.85 <= y & y <= 1, -35.3*y + 35.3, NA)))}

wu <- function(y){ifelse(0 <= y & y <= 1, -2*y + 7.5, NA)}
```

The code chunk below plots $W_L(x)$, $W_U(x)$ and $\frac{\tilde{f_X}(x)}{g_X(x)}$.

```
plot2 <- function(){#same motivation for plotting function again
  x=seq(0,1,l=10000)
  df2 = data.frame(x=x, fn1=foverg_complete(x), fn2=wl(x), fn3=wu(x))
  mylabels=list(expression(tilde(f[X])(x)/g[Y](x)),
    expression(W[L](x)), expression(W[U](x)))
  p <- ggplot(df2)
  p + geom_line(aes(x,fn1,colour="fn1"))+
  geom_line(aes(x,fn2,colour='fn2'))+
  geom_line(aes(x,fn3,colour='fn3'))+
  labs(y="Density",
    title=expression("Plot of "~tilde(f[X])(x)/g[Y](x)~","~W[L](x)~"and"~W[U](x)))+
  scale_colour_manual("", values=c("fn1"="#CC3333", "fn2"="#3399CC", 'fn3'='#336600'),
    labels=mylabels)
}

plot2()
```



The plot demonstrates we have constructed the functions exactly as we wanted.

At this point, we are able to define the final rejection algorithms to sample from $X \sim f_X$. There will be two cases, one with squeezing, and one without.

Rejection Algorithm with squeezing:

1. Generate $U = u \sim U(0, 1)$
2. Generate $Y = y \sim Beta(\alpha, \beta)$
3. If $Mu \leq W_L(y)$ go to 4, else if $Mu > W_U(y)$ go to 1, else if $u > \frac{\tilde{f_X}(x)}{g_Y(x)}$ go to 1, else go to 4
4. Accept and set $X = y$

Alternatively, we have the rejection algorithm without squeezing:

1. Generate $U = u \sim U(0, 1)$
2. Set $Y = y \sim Beta(\alpha, \beta)$
3. If $u \leq \frac{\tilde{f_X}(x)}{g_Y(x)}$ then accept and set $X = y$
4. Otherwise go to 1

Before the implementation of these algorithms let's calculate the theoretical acceptance probability, so we can compare it to the observed acceptance value. To do this, we compute the following integral numerically,

$$\theta = \frac{\int_0^1 \tilde{f_X}(x)dx}{M} \approx 0.76.$$

The code chunk below implements this.

```r
integrand1 <- function(x){ #integrand for theoretical acceptance probability
  ftilde(x)/M
}

theta = integrate(integrand1, lower=0, upper=1)
theta = theta$value#theoretical acceptance probability
#stored as global variable to maximize the efficiency of our sampling methods.
```

Below, are the algorithms for sampling from the distribution we were given: *rdsSQ* unsurprisingly is the method with squeezing and *rds* is the one without. They generate n samples, efficiently and return a list of samples, predicted acceptance probability and actual acceptance probability.

```r
rdsSQ <- function(n){ #the function which generates the sample
  x <- vector("numeric") #vector to keep track of simulated values
  count <- 0
  total <- 0
  while(count<n){
    ngen = ceiling(1/theta * (n-count)) #reduces the number of iterations
    u <- runif(ngen) #generate u, uniformly, for accept/reject
    y <- beta05(ngen)#generate v, uniformly, for use in the envelope function
     #generate y using beta inversion
    out <- ifelse(M*u <= wl(y), T, ifelse(M*u > wu(y), F,
      ifelse(u > foverg_complete(y)/M, F, T))) #conditioning case as mentioned above
    x <- c(x, y[out]) #accept X=y
    count <- length(x) #tracks acceptance number
    total <- total + ngen #tracks total number
 }
 predictedp = format(theta) #theoretical acceptance probability
 actualp = format(count/total) #actual acceptance probability
 return(list(x=x, 'Predicted Acceptance Probability'=predictedp,
  'Actual Acceptance Probability'=actualp))
}

rds <- function(n){ #the function which generates the sample
  x <- vector("numeric") #vector to keep track of simulated values
  count <- 0
  total <- 0
  while(count<n){
    ngen = ceiling(1/theta * (n-count)) #reduces the number of iterations
    u <- runif(ngen) #generate u, uniformly, for accept/reject
    y <- beta05(ngen) #generate y using beta inversion
    x <- c(x, y[u <= foverg_complete(y)/M]) #accept X=y if u <= f(y)/Mg(y)
    count <- length(x) #tracks acceptance number
    total <- total + ngen #tracks total number
 }
 predictedp = format(theta) #theoretical acceptance probability
 actualp = format(count/total) #actual acceptance probability
 return(list(x=x, 'Predicted Acceptance Probability'=predictedp,
  'Actual Acceptance Probability'=actualp))
}
```

Once again, this is a good stage to compute some form of sanity check, are the algorithms functioning as we would expect them to? To do this I have created histograms using samples from *rdsSQ* and *rds*, then

overlayed the PDF. We expect their shapes to be similar.

Before I can implement this function we need to know the normalising constant of the PDF. Now, although Question 3 centres around this, we need to perform some type of calculation now. Using inbuilt R functions we can compute it numerically but, using a substitution we can get to the answer analytically. Consider,

$$\int_0^1 \frac{1}{x(1-x)} exp(-\frac{1}{9}(0.3+log(\frac{x}{1-x}))^2)dx = \frac{1}{K}.$$

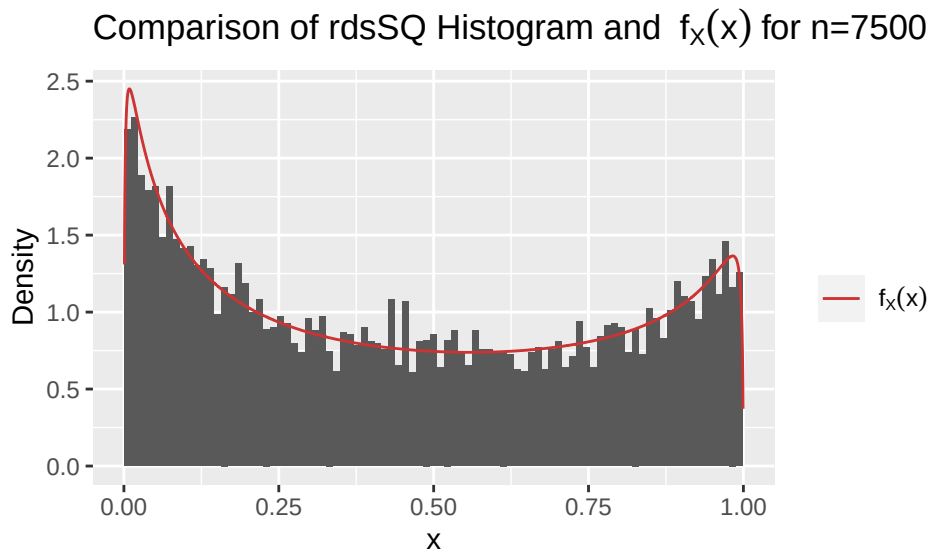
Letting $u = log(\frac{x}{1-x})$, transforms the integral to,

$$\int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{9}(0.3+u^2)}du = K\int_{-\infty}^{\infty} \mathrm{e}^{-\frac{1}{2}(\frac{u-(-0.3)}{\sqrt{4.5}})^2}du = K\int_{-\infty}^{\infty} \frac{1}{K}\mathrm{e}^{-\frac{1}{2}(\frac{u-(-0.3)}{\sqrt{4.5}})^2}du = 1.$$

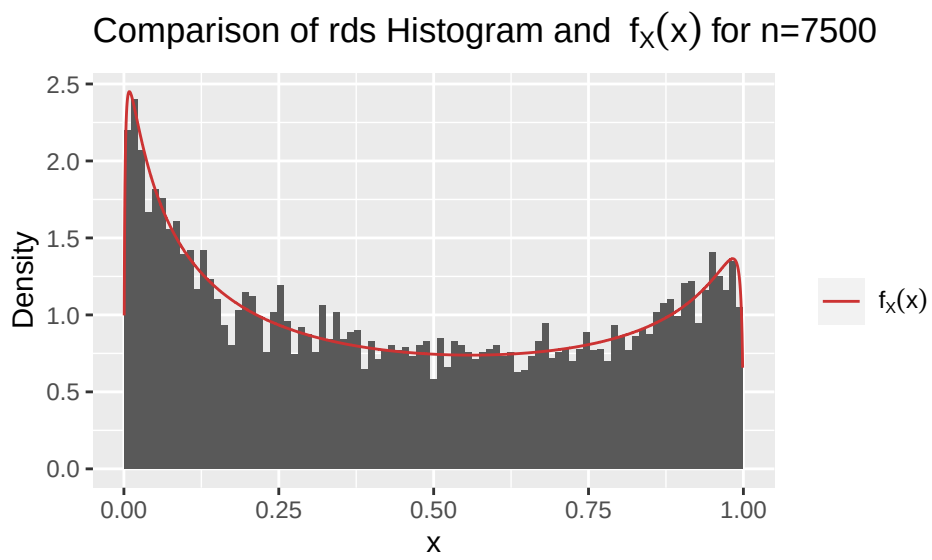
We recognise this as the integral of a normal PDF namely, $U \sim N(-0.3, 4.5)$, so we clearly have that $K = \sqrt{2\pi\sigma^2} = \sqrt{9\pi}$.

The code chunk below produces the plots described above.

```
K = sqrt(9*pi) #stored globally for later use

f <- function(x){#pdf
  1/K * (1/(x*(1-x)))*exp((-1/9)*(0.3+log(x/(1-x)))^2)
}

plotrdsSQ <-function(n){
  x = rdsSQ(n)$x #ensuring we'll have numeric data
  df1 = data.frame(x, pdf=f(x))
  df2 = data.frame(x)
  mylabels=list(expression(f[X](x)))
  #plotting the graph
  p <- ggplot(df1)
  p + geom_histogram(aes(x,y= ..density..), breaks=seq(0, 1,l=90))+
  geom_line(aes(x,pdf,colour="pdf"))+
  labs(y="Density",
    title=expression("Comparison of rdsSQ Histogram and "~f[X](x)~"for n=7500"))+
  scale_colour_manual("", values=c("pdf"="#CC3333"), labels=mylabels)
}

plotrds <-function(n){
  x = rds(n)$x #ensuring we'll have numeric data
  df1 = data.frame(x, pdf=f(x))
  df2 = data.frame(x)
  mylabels=list(expression(f[X](x)))
  #plotting the graph
  p <- ggplot(df1)
  p + geom_histogram(aes(x,y= ..density..), breaks=seq(0, 1,l=90))+
  geom_line(aes(x,pdf,colour="pdf"))+
  labs(y="Density",
    title=expression("Comparison of rds Histogram and "~f[X](x)~"for n=7500"))+
  scale_colour_manual("", values=c("pdf"="#CC3333"), labels=mylabels)
}

plotrdsSQ(7500)
```



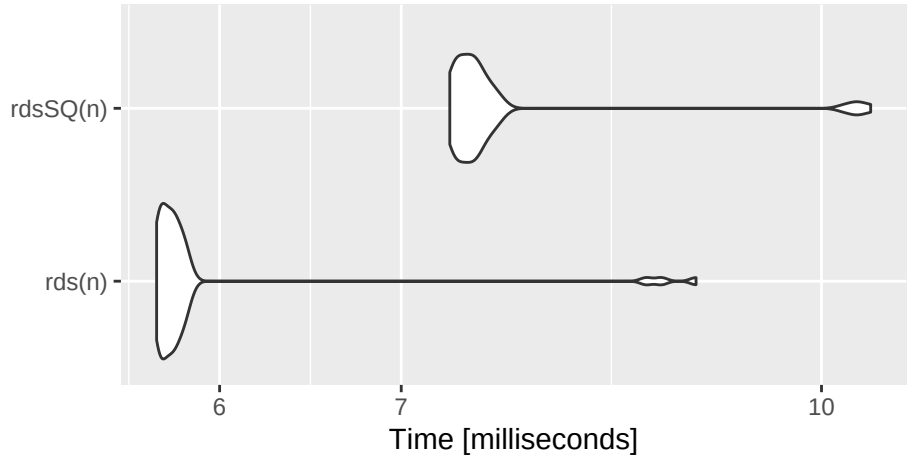

```
plotrds(7500)
```



Evidently, both functions are working as we want them to and sampling correctly. I chose $n = 7500$ so that we have a large sample size to see the trends clearly. Below are some timings using the 'microbenchmark' package, I chose a sample size of $10,000$ so that we can take many tests to calculate more accurate results, instead of far fewer repeats with a very large n.

```
benchmark <- function(n, t){
  results <- microbenchmark(rds(n), rdsSQ(n), times = t)
  a <- autoplot(results)
  return(list(results,a))
}

benchmark(10000,50)
```

```
## [[1]]
## Unit: milliseconds
##      expr      min       lq     mean   median       uq       max neval
##    rds(n) 5.687701 5.706302 5.934813 5.757901 5.806401  8.990101    50
##  rdsSQ(n) 7.294501 7.339702 7.653793 7.426901 7.509200 10.426301    50
##
## [[2]]
```



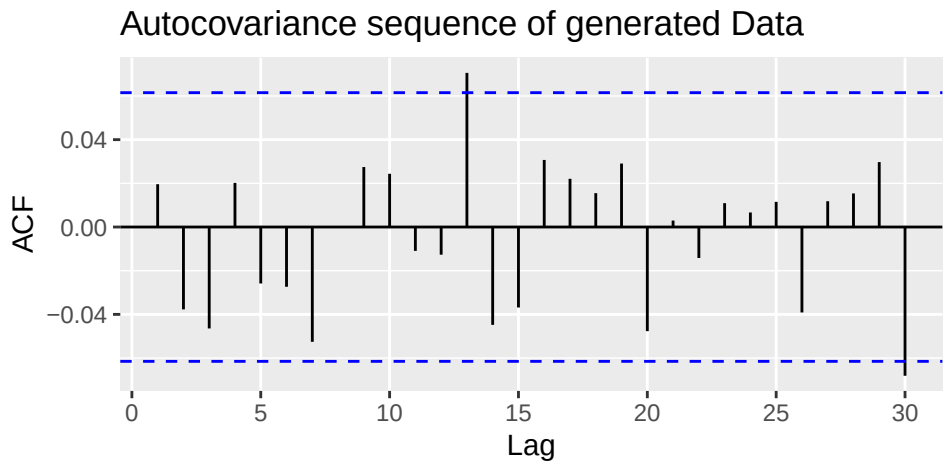
We see, using the data and violin plots, that on average the regular *rds* function is more efficient than *rdsSQ*; this is most likely due to the fact that $f_X(x)$ and $g_Y(x)$ are not too complex, and the extra conditioning within the squeezing functions and final algorithm slows *rdsSQ* down. Both functions are computing samples of $10,000$ in under 10 milliseconds which is very efficient.

2

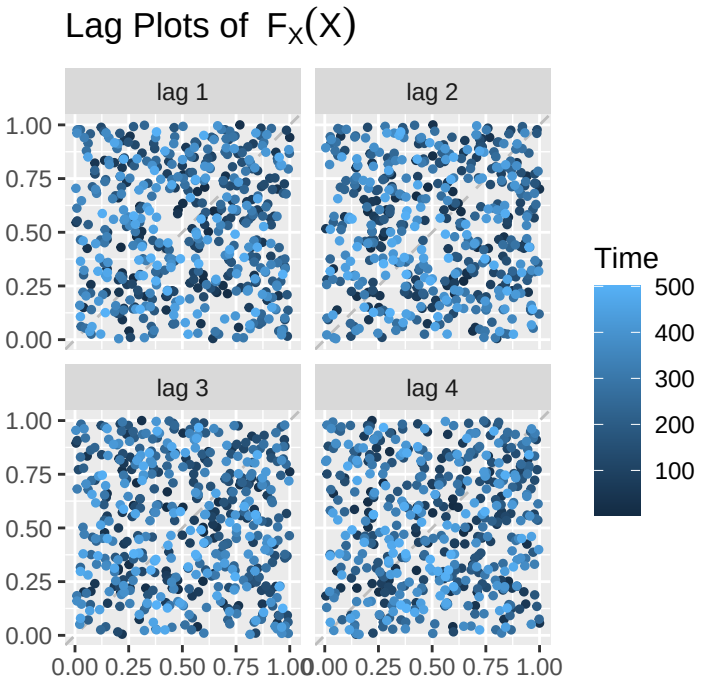
The next stage is to complete some diagnostic tests, we will only consider the *rds* function as it is more efficient and we are satisfied that both *rds* and *rdsSQ* are sampling very similarly from $X \sim f_X$.

To create auto-covariance and lag plots, we need F_X: this can only be computed numerically using integrals. However, the integral function in R does not operate element-wise for vectors. This creates issues as the diagnostics need the CDF to be able to take a vectorised argument. As such, I created a function 'cdfFx' which finds the CDF of each element of a vector. I compared *lapply* against *vectorize* and found the former to be more efficient (these tests have been excluded due to page count). Below is code implementing both plots.

```
cdfFx <- function(x){ #CDF function
  unlist(lapply(x, function(y) ifelse(y==0, 0,
    ifelse(y>0 & y<1, integrate(f, lower=0, upper=y)$value, ifelse(x==1, 1)))))
}
#function which produces an autocovariance plot
autocov <- function(n){
  x = rds(n)$x
  ggAcf(x)+
  labs(title="Autocovariance sequence of generated Data")
}
autocov(1000)
```



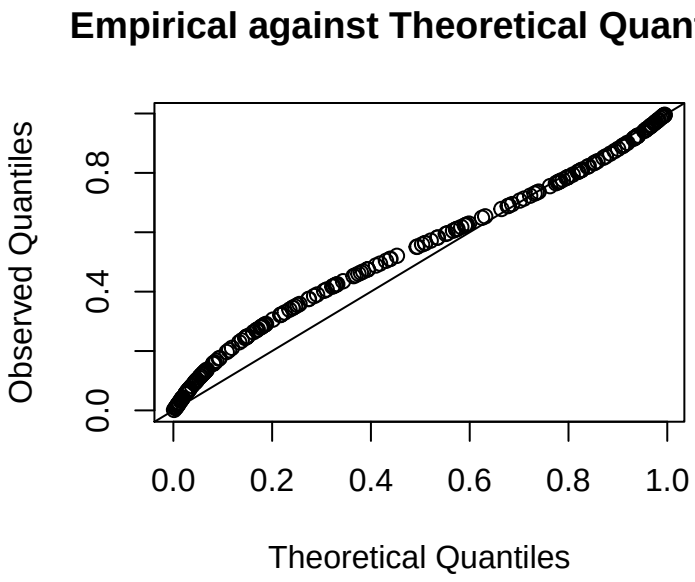

```r
#function which produces an lag plot
lagplt <- function(n){
  x = rds(n)$x
  gglagplot(cdfFx(x), lags=4, do.lines=FALSE)+
  labs(title=expression("Lag Plots of " ~ F[X](X)))
}

lagplt(500) #smaller sample size so the plots are clear and coherent
```



The auto-covariance shows no signs of significant correlation at any lag and, the lag scatter plots of $F_X(x_i)$ are apparently random which would suggest that points are being generated independently from f_X.

The final diagnostic plot I'll be utilising is a 'QQplot'. This plots the theoretical quantiles against the observed ones. The code chunk below implements this.

```
qqplot<- function(n){ #function to produce qqplots, saves using global variables
  x0 = rds(n)$x
  x0 <- sort(x0)

  cdfq <- function(p,q=0){ #function which computes F_X(x) - q
  cdfFx(p)-q
  }

  qfunc <- function(q){#finds the root between zero and one of cdfq and q
    bisect(cdfq, 0, 1,q=q)$root
  }
  qhn1 <- function(p) unlist(lapply(p, qfunc)) #ensures we return a vector

  plot(x = qhn1(x0), y = x0, main="Empirical against Theoretical Quantiles",
       xlab = "Theoretical Quantiles", ylab="Observed Quantiles")#plot
  abline(0,1)
}

qqplot(200)
```



The QQ plot provides a very interesting result. The shape of the data points is almost linear, with only slight deviations at the lower tail. From that we can confirm that the quantiles are very similar for both the observed and theoretical CDF. We are building a convincing argument, from the diagnostics, that the functions are sampling correctly. However, let's conduct some diagnostic tests to reinforce the predictions.

Firstly, we consider the Chi-Squared diagnostic test. This is a 'goodness of fit' test and will reveal whether it is likely that the samples generated by *rds* are from $X \sim f_X$. It performs a hypothesis test by comparing the probability of being in a numeric 'bin' against the number of generated values in said 'bin'. We have:

H_0 = *rds* generates samples from $X \sim f_X$.
H_a = *rds* does not generate samples from $X \sim f_X$.

Below is the code chunk implementing this test.

```
chitest <- function(n){
  x <- seq(0,1,length.out=21)#bin values
  x <- x[1:20] #removed last value as we are parametrising by 'lower' in the integral
  binwidth <- x[5]-x[4] #width of bins
  data = rds(n)$x #sample from rds
  expected <- unlist(lapply(x, function(y) integrate(f, lower=y, upper=y+binwidth)$value))
  observed <- unlist(lapply(x, function(y) length(data[y<data & data<y+binwidth])))
  res <- chisq.test(x=observed, p=expected, rescale.p=TRUE)
  return(res)
}

chitest(1000000)
```

```
##
##  Chi-squared test for given probabilities
##
## data:  observed
## X-squared = 21.496, df = 19, p-value = 0.31
```

Viewing the p-value, we see it is greater than 0.05, this mean we accept the null hypothesis (at a 95% significance) and can confirm that *rds* is generating from the correct distribution.

In addition to the Chi-Squared test, we can perform the Kolmogorov-Smirnoff test. This checks whether there is significant statistical evidence to suggest the sample was generated from $X \sim f_X$. It performs a Hypothesis test like before and also calculates $D = \sup_x |F_{X_n}(x) - F_X(x)|$. I have implemented the KS test as well as Chi-Squared test, as it uses a different method to evaluate the hypothesis and calculates a different statistic. Hence if our results match the Chi-Squared test we can be very confident that we are sampling correctly. Our hypotheses are:

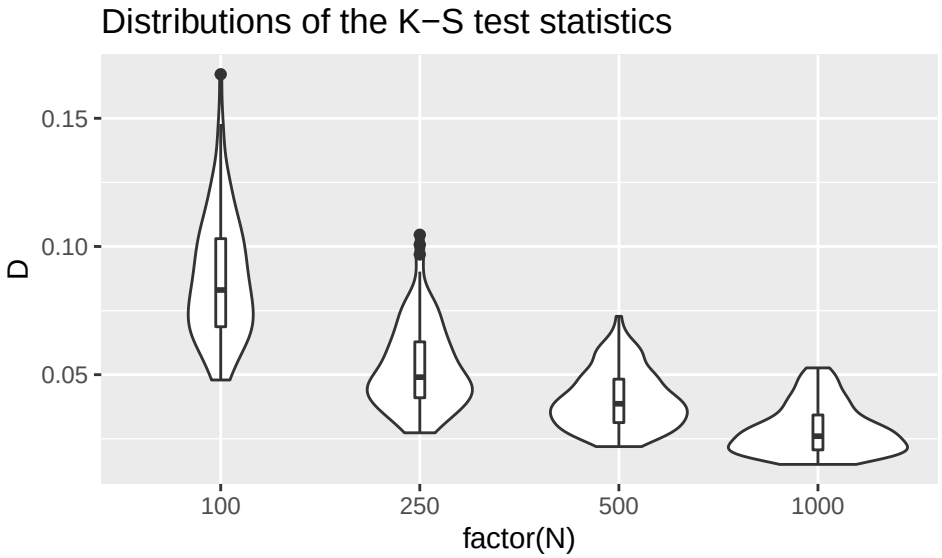
H_0: *rds* generates samples from $X \sim f_X$.
H_a: *rds* does not generate samples from $X \sim f_X$.

Beneath is the KS test.

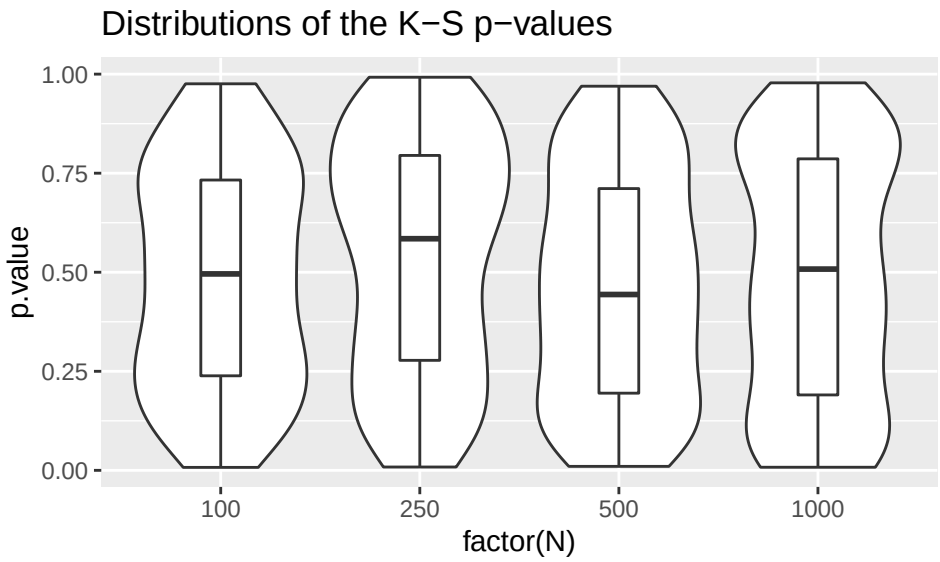
```
#A short function to perform the KS test and return the P and D value
ksfunction <- function(x){
  kstestx = ks.test(x,cdfFx)
  return(c(kstestx$p.value, kstestx$statistic))
}

kstest <- function(){
  N = c(100,250,500,1000) #sample sizes to test
  Nlen = length(N)
  p=100 #number of repeats for means
  ks.results=data.frame()

  for(n in 1:Nlen){
    n1=N[n]
    x <- matrix(0, nrow=n1, ncol=p)
```

```
    # one call to rds, then split into matrix in order to use the apply function
    x1 = rds(n1*p)$x
    for(i in 1:p)  x[,i] <- x1[((i-1)*n1+1):(i*n1)]
    ks.testx= apply(x,2,ksfunction)
    ks.results = rbind(ks.results, data.frame(p.value=ks.testx[1,], D = ks.testx[2,],
      N=rep(n1,p)))
  }
  #plotting results
  p1 <- ggplot(ks.results, aes(factor(N), D))+
  geom_violin()+
  geom_boxplot(width=0.05)+
  labs(title="Distributions of the K-S test statistics")
  p2 <- ggplot(ks.results, aes(factor(N), p.value))+
  geom_violin()+
  geom_boxplot(width=0.2)+
  labs(title="Distributions of the K-S p-values")
  p3 <- ggplot(ks.results, aes(p.value,colour=factor(N)))+
  #geom_histogram(breaks=seq(0,1,by=0.05))+
  geom_freqpoly(breaks=seq(0,1,0.1))+
  labs(title="Frequency polygons of p-values")
  #summarising the results in a table
  ks.table <- ks.results %>% group_by(N) %>%
    summarise("Mean p-value" = round(mean(p.value),digits=3),
    "Std Dev (p)" = round(sqrt(var(p.value)), digits=3),
    "Mean D"=round(mean(D), digits=3), "Std Dev (D)"= round(sqrt(var(D)), digits=3))
  t <- kable(ks.table) %>% kable_styling(position="center")
  return(list(p1,p2,p3,t))
}

kstest()
```

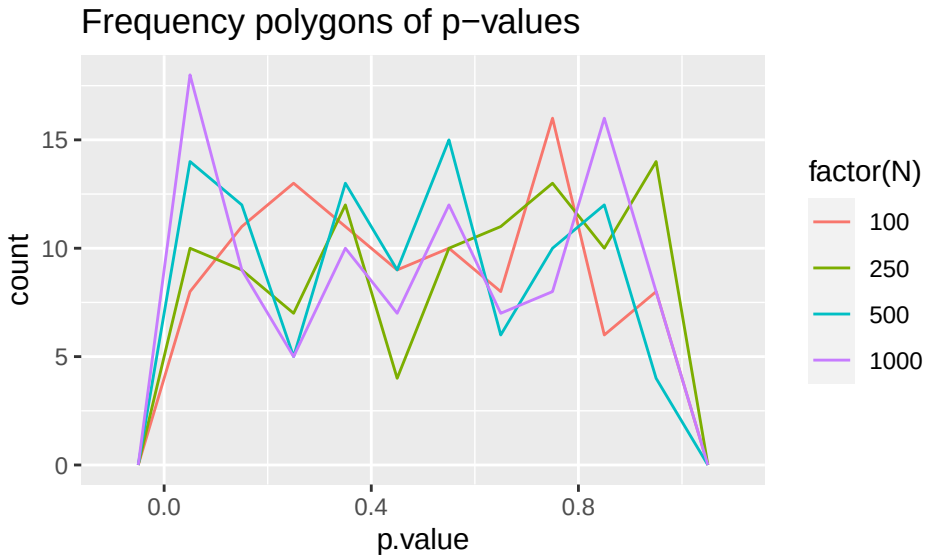
[[1]]



[[2]]



[[3]]



[[4]]

N	Mean p-value	Std Dev (p)	Mean D	Std Dev (D)
100	0.485	0.278	0.088	0.024
250	0.540	0.297	0.053	0.016
500	0.465	0.285	0.040	0.011
1000	0.483	0.313	0.029	0.010

For small N, we see the results are unsurprisingly inaccurate due to the sample size. However as N gets larger the mean D-value approaches zero which informs us that the samples are very likely to be from the same distribution (this is validated by the standard deviation values). In addition, the p-value is greater than 0.05 in all cases; this means that we accept the null hypothesis (*rds* generates samples from $X \sim f_X$)

and as such, we can be statistically confident that the two samples are from the same distribution.

The violin plots of the p-value and D-statistic give a great visual interpretation of the results, demonstrating that the algorithm is working correctly. The frequency polygon with a high-mid central region reveals that most of the calculated p-values were between 0.1 and 0.9 which is once again, what we want.

From the many diagnostic we have completed, it is clear that we have sampled very effectively from $X \sim f_X$.

3

We are tasked with computing the normalising constant of $\tilde{f_X}(x)$ using Crude Monte Carlo integration with minimal variance. The general method for computing the constant is as follows:

We want to estimate

$$\theta = \int_0^1 \tilde{f_X}(x)dx = \sqrt{9\pi}.$$

To do so, we continue as follows,

1. Write $\theta = \int_0^1 \tilde{f_X}(x)dx = \int_0^1 \phi(x)h_Z(x)$, where $h_Z(x)$ is a PDF over the domain of the integral (transformations can be made).
2. We now approximate θ by $\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n} \phi(X_i)$, where $X_i \sim Z$.
3. $Var(\hat{\theta}) = \frac{Var(\phi(X))}{n}$ and $E(\hat{\theta}) = \theta$

The code chunk below implements some rudimentary procedures to calculate estimates for the normalising constant and $nVar(\theta)$. The first method sets $h_Z(x) = 1, \quad Z \sim U(0,1)$. The second sets $h_Z(x)$ to be a beta PDF; using $\alpha, \beta = 0.5$ so we can generate the sample using the same method as before.

```
ftsq <- function(x){#f~ squared
  ((1/(x*(1-x)))*exp((-1/9)*(0.3+log(x/(1-x)))^2))^2
}

uniformmc <- function(n){mean(ftilde(runif(n)))} #estimate from uniform

uniformvar <- function(){#uniformvariance calculation
  integrate(ftsq, lower=0, upper=1)$value - (integrate(ftilde, lower=0, upper=1)$value)^2
}

phi_beta <- function(x){ftilde(x)/g(x)} #phi for the beta case

betamc <- function(n){mean(phi_beta(beta05(n)))} #beta estimation

betavar <- function(){
  phi2 <- function(x){ftsq(x)/g(x)}
  integrate(phi2, lower=0, upper=1)$value - (integrate(ftilde, lower=0, upper=1)$value)^2
}

estimates <- function(){
  c1 <- c(uniformmc(100000), betamc(10000))
  c2 <- c(uniformvar(), betavar())
```

```
  est.results = cbind(c1, c2)
  colnames(est.results) <- c("Estimate", "nVariance")
  rownames(est.results) <- c("Uniform", "Beta")
  kable(est.results) %>% kable_styling(position="center")
}

estimates()
```

	Estimate	nVariance
Uniform	5.311917	3.458656
Beta	5.302968	3.375741

From the data we see both estimates are near $\sqrt{9\pi} \approx 5.317$ which is the correct answer (we know from the calculation in Question 2). Although the estimates are accurate, we want to reduce the variance. I could not create a variance reduction method by transforming the integral, largely due to the complex form of $\tilde{f_X}(x)$. So, I had to consider either antithetic or control variates to reduce the variance. Recalling the plot of $\tilde{f_X}(x)$, it is not monotonic, so I chose control variates. The main idea of control variates is as follows:

Say we have a situation of estimating, $\theta = E_{f_X}[\phi(X)] = E_{f_X}[Z]$ and, for $j = 1, ...p$, $\exists W^{(j)} = \psi(X)$ such that $E_{f_X}[W^{(j)}]$ is known and W is correlated with Z.

Then, given a sample $X_1, ..., X_n \overset{iid}{\sim} f_X$, we construct the estimator:

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n}\left[Z_i - \sum_{j=1}^{p}\beta_j(W_i^{(j)} - E(W^{(j)}))\right].$$

$E(\hat{\theta}) = \theta$ as required and β_j are weights which we determine. Recall that our aim is to reduce the variance, consider

$$Var(\hat{\theta}) = \frac{1}{n}Var\left(Z - \sum_{j=1}^{p}\beta_j(W^{(j)} - E(W^{(j)}))\right)$$

$$Var(\hat{\theta}) = \frac{1}{n}\left(E\left[Z - \sum_{j=1}^{p}\beta_j(W^{(j)} - E(W^{(j)}))\right] - \theta^2\right).$$

Hence, we can minimise variance by minimising the above expectation. We will use regression to determine the various beta values that fit our needs.

I implemented two methods using control variates, in both, x_i are generated from $U(0,1)$. The first,

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n}\left[\tilde{f_X}(x_i) - \beta_1(W_1(x_i) - E_{f_X}(W_1)) - \beta_2(W_2(x_i) - E_{f_X}(W_2))\right]$$

and the second,

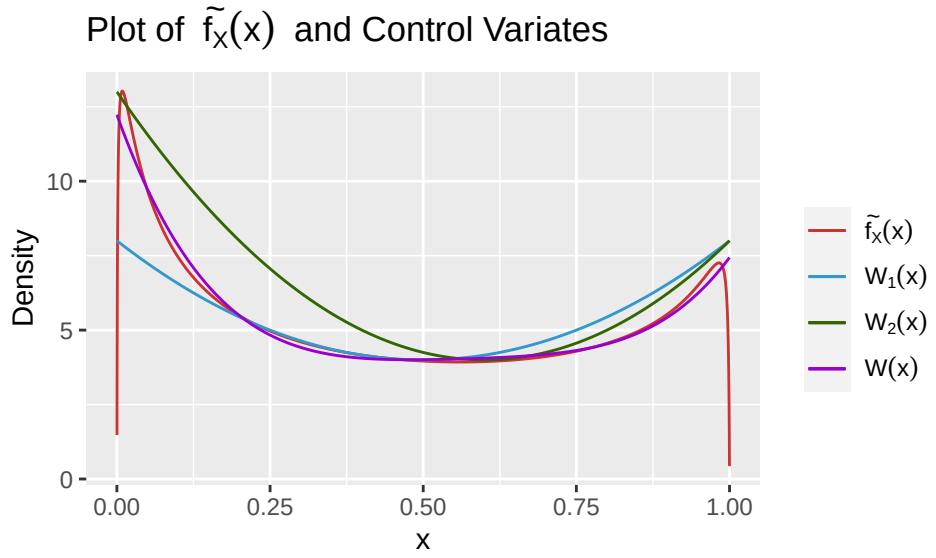
$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n}\left[\tilde{f_X}(x_i) - \beta(W(x_i) - E_{f_X}(W))\right].$$

To calculate W_1, W_2, I used two quadratics which were similar to $\tilde{f_X}(x)$, calculated using the appearance of $\tilde{f_X}(x)$. For W, I used a least squares algorithm to determine a quartic polynomial which was the closest fit

to $\tilde{f_X}(x)$, to increase the correlation and thus, reduce variance.

The code chunks below implement this, the first constructs and plots our variates.

```
w1 <- function(x){#a quadratic similar to f~ to reduce cor
  16*x^2 - 16*x + 8
}

w2 <- function(x){#a quadratic similar to f~ to reduce cor
  25*x^2 - 30*x + 13
}

wcoeff <- function(n){#least squares to calculate coefficients of W
  x=runif(n)
  y=ftilde(x)
  x0 = rep(1, n)
  x1 = x
  x2 = x^2
  x3 = x^3
  x4 = x^4
  lsfitw=lm(y~x0 + x1 + x2 + x3 + x4-1)
  return(lsfitw)
}

coeffs <- wcoeff(10000000) #storing the coefficients globally for efficiency

w <- function(x){
  coeffs$coefficients[1] + coeffs$coefficients[2]*x + coeffs$coefficients[3]*x^2 +
    coeffs$coefficients[4]*x^3 + coeffs$coefficients[5]*x^4
}

EW1 <- integrate(w1, lower=0, upper=1)$value #E(W_1)
EW2 <- integrate(w2, lower=0, upper=1)$value #E(W_2)
EW <- integrate(w, lower=0, upper=1)$value #E(W)

plotcontrol <- function(){#function to produce plots
x=seq(0, 1 ,l=10000)
n=10000
df2 = data.frame(x=x, fn1=ftilde(x), fn2=w1(x), fn3=w2(x), fn4=w(x))
  mylabels=list(expression(tilde(f[X])(x)), expression(W[1](x)), expression(W[2](x)),
    expression(W(x)))
  p <- ggplot(df2)
  p + geom_line(aes(x,fn1,colour="fn1"))+
  geom_line(aes(x,fn2,colour='fn2'),)+
  geom_line(aes(x,fn3,colour='fn3'))+
  geom_line(aes(x, fn4, color='fn4'))+
  labs(y="Density", title=expression("Plot of "~tilde(f[X])(x)~" and Control Variates"))+
  scale_colour_manual("", values=c("fn1"="#CC3333", "fn2"="#3399CC", 'fn3'='#336600',
    'fn4'='#9900CC'), labels=mylabels)
}

plotcontrol()
```



We see all the curves have a fairly simple shape to $\tilde{f_X}(x)$ (ignoring the downward tails) which is what we want.

The next chunk performs a correlation check.

```
correlation_check <- function(n){
  x=runif(n)
  w1s= w1(x) - EW1 #we shift the function by their expectation
  w2s= w2(x) - EW2
  ws= w(x) - EW
  cat("cor(f(x),w)=",cor(ftilde(x),ws),"\n")
  cat("cor(f(x),w1)=",cor(ftilde(x),w1s),"\n")
  cat("cor(f(x),w2)=",cor(ftilde(x),w2s),"\n")
}

correlation_check(1000000)
```

```
## cor(f(x),w)= 0.9737952
## cor(f(x),w1)= 0.8255172
## cor(f(x),w2)= 0.9177668
```

Here, we see there is quite high correlation between W_1, W_2 and $\tilde{f_X}(x)$ which is important for our algorithm. Unsurprisingly, W has much higher correlation, this is to be expected based on the way it was constructed.

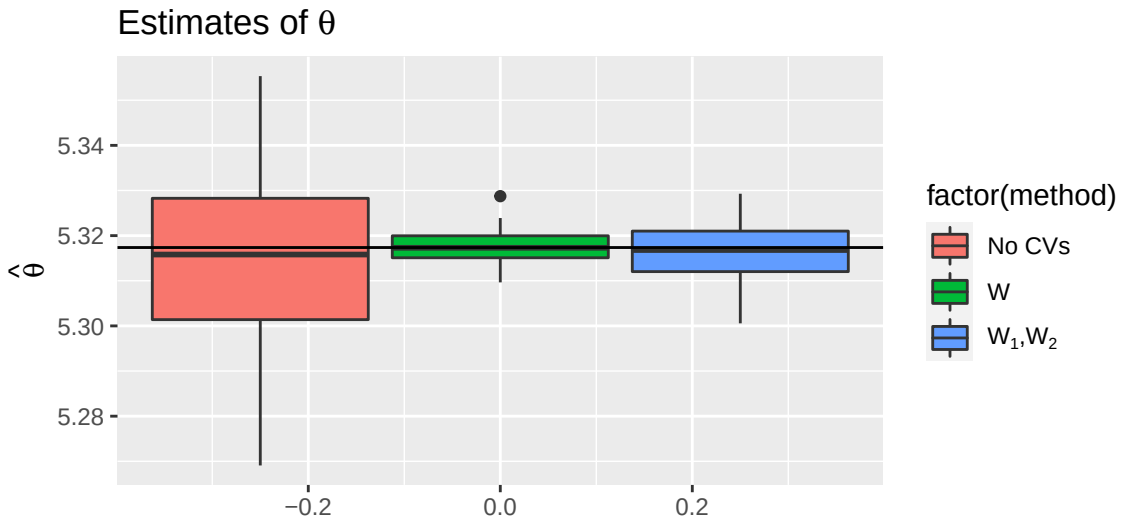
The next code chunk calculates our weights and the variance of both estimates.

```
bsim <- function(n){
  x=runif(n) #random uniform sample
  y=ftilde(x) #generate ftilde values for regression
  w1v= w1(x) - EW1 #shift functions by their expectation
  w2v= w2(x) - EW2
  wv= w(x) - EW
  lsfitcv1=lm(y~w1v+w2v-1) #least squares
  lsfitcv2=lm(y~wv-1)
```

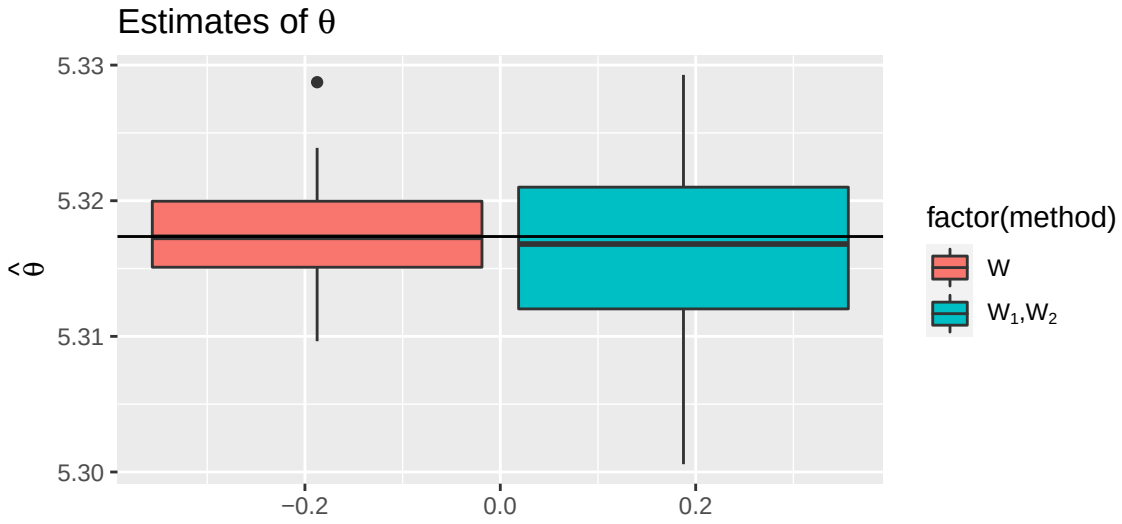
```
  b1 = lsfitcv1$coefficients[1] #coefficients
  b2 = lsfitcv1$coefficients[2]
  b = lsfitcv2$coefficients[1]
  theta2 = K^2
  varest1 = sum(lsfitcv1$residuals^2)/n-theta2 #variance
  varest2 = sum(lsfitcv2$residuals^2)/n-theta2
  return(list(b1, b2, b, varest1, varest2))
}

bsimv <- bsim(1000000) #stores variables globally
```

The next chunk runs another simulation study to view the performance of our new estimators and compare to the Monte-Carlo estimator without control variates.

```
n = 10000 #sample size
nrep = 100 #number of repeats for mean

thetahat1=thetahat2=thetahat3=vector()
for(i in 1:nrep){
  x=runif(n)
  w1vals = w1(x)
  w2vals = w2(x)
  wvals = w(x)
  thetahat1[i]=mean(ftilde(x))
  thetahat2[i]=mean(ftilde(x)-bsimv[[3]]*(wvals - EW))
  thetahat3[i]=mean(ftilde(x)-bsimv[[1]]*(w1vals - EW1)-bsimv[[2]]*(w2vals- EW2))
}

cv1 = data.frame(thetahat=thetahat1,method=rep("f(x)",nrep),vart=rep(uniformvar(),nrep))
cv2 = data.frame(thetahat=thetahat2,method=rep("w",nrep),vart=rep(bsimv[[5]],nrep))
cv3 = data.frame(thetahat=thetahat3,method=rep("w1 and   w2",nrep),vart=rep(bsimv[[4]],
                                                                            nrep))
cv=rbind(cv1,cv2,cv3)
cv23=rbind(cv2, cv3)
mylabs=c("No CVs","$W$", "$W_1, W_2$")
ggplot(data=cv, aes(fill=factor(method),y=thetahat))+geom_boxplot()+
labs(title=TeX("Estimates of $\\theta$"), x="", y=TeX("$\\hat{\\theta}$"))+
geom_hline(yintercept=K)+
scale_fill_discrete(labels=lapply(mylabs,TeX))
```




```
ggplot(data=cv23, aes(fill=factor(method),y=thetahat))+ geom_boxplot()+
  labs(title=TeX("Estimates of $\\theta$"), x="", y=TeX("$\\hat{\\theta}$"))+
  geom_hline(yintercept=K)+ scale_fill_discrete(labels=lapply(mylabs[2:3],TeX))
```




```
cv_summary <- cv %>%
group_by(method) %>%
summarise(frequency = n(), mean_thetahat=mean(thetahat), nmeanvar=mean(as.double(vart)))
cv_summary$nmeanvar =format(cv_summary$nmeanvar,digits=4)
cv_summary$mean_thetahat =format(cv_summary$mean_thetahat,digits=5)

cv_summary <- cv_summary[,-1]
row.names(cv_summary) <- c("No CVs","$W$","$W_1, W_2$")
cv_summary %>% kable(align=rep("l",3),col.names=c("$n$","$\\hat{\\theta}$",
  "$n \\mbox{var}(\\hat{\\theta})$"),
```

```
escape=F) %>% kable_styling(position="center",
bootstrap_options = c("bordered"),full_width = FALSE)
```

	n	$\hat{\theta}$	$n\text{var}(\hat{\theta})$
No CVs	100	5.3148	3.4587
W	100	5.3173	0.1850
W_1, W_2	100	5.3165	0.4584

I have included two box plots, the second shows a higher resolution for the smaller variances generated using control variates. On the box plots the horizontal line passing through the whole diagram is the actual value of the normalising constant, the line within each box represents the estimate. The results are as we would expect, there is significant reduction in variance between the uniform estimator and the weighted version. In addition, our variance is reduced further by the weight with the highest correlation to $f_X(x)$. But, all estimators are very close to the actual value.

I would expect the best way to reduce the variance further would be some form of manipulation to the original integral, potentially changing the domain or, the function itself by a substitution; however I could not devise a way to do this.

There is an alternative method to Crude Monte Carlo namely, Hit-or-Miss Monte Carlo. However, there is little point implementing the procedure, as we can analytically show that variance is always greater with the Hit-or-Miss method than with the Crude method.

Let's outline the Hit-or-Miss method then discuss the issues with its variance.

Let h be a bounded function on (a, b) with $0 \leq h \leq c$. As before, we aim to estimate

$$\theta = \int_a^b h(x)dx = \text{area under curve.}$$

We take a uniform sample from a rectangle: $U = u_i \sim U(0,1), V = v_i \sim U(0,1), \quad i = 1, ..., n$ and define

$$\tilde{\theta} = \frac{c(b-a)}{n} \sum_{i=1}^{n} \text{I}(v_i \leq h(u_i)).$$

It can be shown using definitions that $E(\tilde{\theta}) = \theta$ and $Var(\tilde{\theta}) = \frac{\theta}{n}[c(b-a) - \theta]$.

Now, let's compare the variance calculations to Crude Monte Carlo. Recall,

$$\theta = \int_a^b h(x)dx = \int_a^b \phi(x) f_X(x)dx, \quad f_X(x) = \frac{1}{b-a}, \quad x \in (a, b).$$

So, $\phi(X) = (b-a)h(X)$ and the Crude Monte Carlo estimator is:

$$\hat{\theta} = \frac{1}{n} \sum_{i=1}^{n} \phi(X_i), \quad X_i \sim U(a, b)$$

$$\hat{\theta} = \frac{(b-a)}{n} \sum_{i=1}^{n} h(X_i), \quad X_i \sim U(a, b).$$

From this, we can derive,

$$E(\hat{\theta}) = \theta$$

$$Var(\hat{\theta}) = \frac{(b-a)^2}{n} \int_a^b \left[h(x) - \frac{\theta}{b-a} \right]^2 f_X(x) dx.$$

Now, we can manipulate the variance to demonstrate that $Var(\hat{\theta}) \leq Var(\tilde{\theta})$:

$$Var(\hat{\theta}) = \frac{b-a}{n} \left[\int_a^b h^2(x) dx - \frac{2\theta}{b-a} \int_a^b h(x) dx + \frac{\theta^2}{(b-a)^2} \int_a^b dx \right]$$

$$Var(\hat{\theta}) = \frac{b-a}{n} \left[\int_a^b h^2(x) dx - \frac{\theta^2}{b-a} \right], \quad \text{let } c = \sup_{y \in (a,b)} [h(y)]$$

$$Var(\hat{\theta}) \leq \frac{b-a}{n} \left[c \int_a^b h(x) dx = \frac{\theta^2}{b-a} \right] = Var(\tilde{\theta})$$

As such, there is no reason to implement the Hit-or-Miss procedure.

Conclusion

Over the course of the three questions, we have successfully taken the complex function $\tilde{f_X}(x)$ and generated a rejection algorithm able to sample very efficently and accurately from $X \sim f_X$, while only requiring *runif*. Following this, we convinced ourselves that the algorithm was working as we expect it to with diagnostic plots. Finally, we computed the normalising constant of $\tilde{f_X}(x)$ using Crude Monte-Carlo integration and discussed its superiority to Hit-or-Miss Monte-Carlo.

References

E. McCoy. Stochastic simulation, Lectures Notes, Imperial College London, 2020. 2020
E. McCoy. Stochastic simulation, RMarkdowns, Imperial College London, 2020. 2020
Shravan Vasishth abd Michael Broe, The Foundations of Statistics: A Simulation-based Approach, Springer