ST407 Monte Carlo Methods Assignment 1

Question 1

The density of interest with parameters is as follows;



*Part a Evaluate density*

The function f.density was used to evaluate in R as follows;

f.density <- function(x, a, b){

y = (x^(a - 1)\*(1-(x^a))^(b - 1))

y

}

Note that the normalizing constant of in 1 is given by however the function was evaluated only up to its normalizing constant. To implement the function f.density and produce the required plot for say , the following code was executed;

x = seq(0,1,length=1000)

a = 1/5

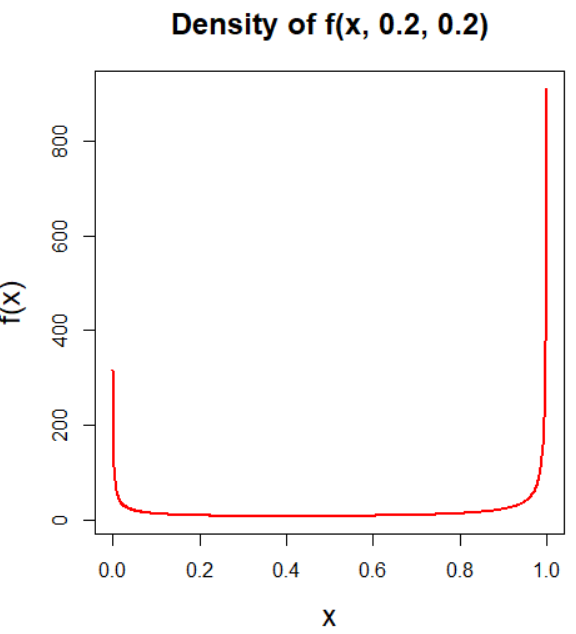
b = 1/5

fx1 = f.density(x, a, b)

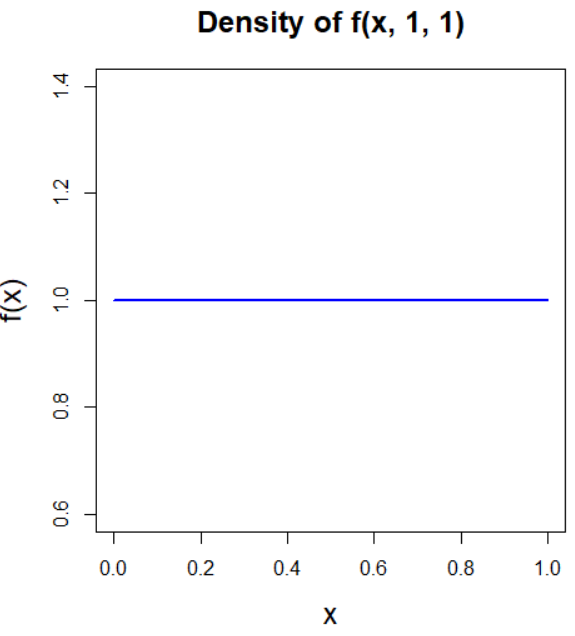
plot(x, fx1)

The plots for all parameterisations are displayed below;

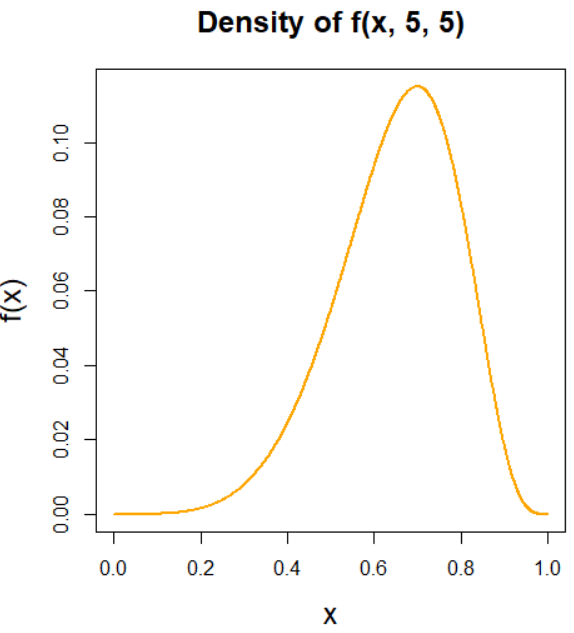
Plot I:



Plot II:



Plot III:



*Part b Inversion Sampling*

Inversion sampling was used to return samples from . Corresponding to the algorithm for inversion sampling, this was achieved by firstly determining the CDF of , namely (See 2), subsequently finding the inverse of the CDF (See 3), generating uniform samples U[0, 1] and obtaining the desired density by means of

CDF

(2)

Inverse CDF

(3)

The function inversion\_sampler was written in R to execute this as below, i.e to generate *n* samples from the target distribution for any given .

inversion\_sampler = function(alpha, beta, n){

#Setup

u = runif(n) #uniform samples [0,1] of sample size n

fu = (1 - (1-u)^(1/b))^(1/a) #Inverse cdf

fu #Return fu

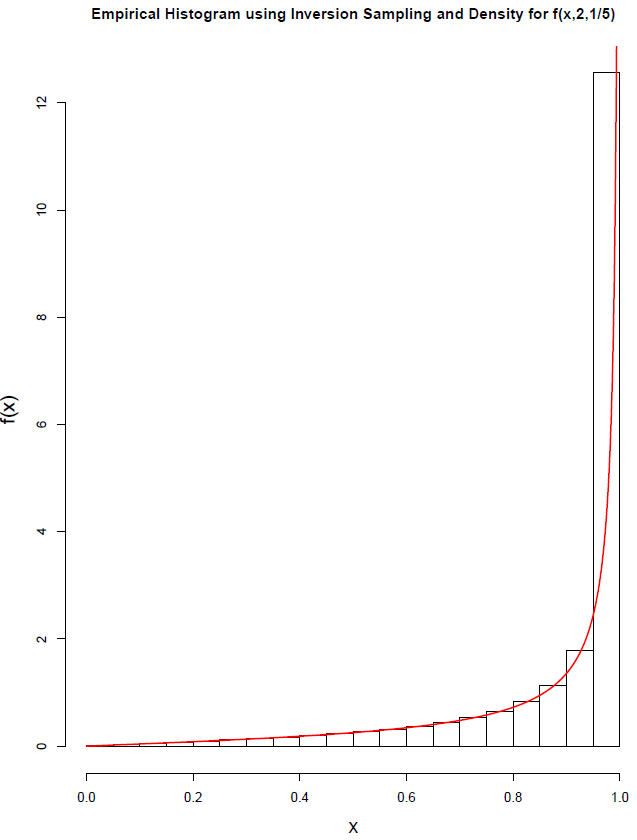
}

A possible assumption in using inversion sampling is that we can get an analytic form of the inverse of the cdf, which is not always feasible, but is in this case.

(ii) To generate 1,000,000 samples from the density involves simply executing the following;

samples = inversion\_sampler(2, 1/5, 1000000)

The required density can be visualised using the hist command. The empirical histogram of the density and the true target density are plotted together in figure 1b illustrating the application of inversion sampling.

 ­­­­

*Figure 1b. Inversion sampling for (n samples = 1,000,000)*

*Part c Rejection Sampling*

Rejection sampling was also used to return samples from using the function rejection\_sampler as below. A uniform distribution was chosen as the proposal distribution. In rejection sampling, a sample is accepted with probability given by;

(4)

Whereby is the target distribution in question (i.e (1)), ) is the proposal distribution (uniform) and is some constant bound given by;

Whereby is the normalizing constant found to be , therefore;

This is maximised when is equal to zero, i.e;

This derivative can be evaluated for any given using the function evaluate\_fx\_derivative as below. To get the maximum of the curve, the roots of this equation were found using the R function uniroot.all (x = 0 was disregarded in this case). The maximum was then found by evaluating the density at the root, i.e;

bound = f\_density(root\_fx, a, b)

Whereby;

f\_density <- function(x, a, b){

norm\_constant = a\*b

y = norm\_constant\*(x^(a - 1)\*(1-(x^a))^(b - 1))

y

}

As previously stated for n given proposal samples, samples corresponding to the target distribution were accepted with probability given by (4) as shown below in the rejection\_sampler function

rejection\_sampler = function(a, b, num\_req\_samps){

#Step 1: Get bound

evaluate\_fx\_derivative = function(a, b){

function (x) (a-1)\*x^(a-2)\*(1-x^a)^(b-1) -a\*(b-1)\*(x^(2\*a-2)\*(1-x^a)^(b-2))

}

fx\_deriv = evaluate\_fx\_deriv(a,b)

#Roots

roots\_all <- uniroot.all(fx\_deriv, c(0, 1))

root\_fx = roots\_all[-1]

bound = f\_density(root\_fx, a, b)

print(bound)

#Num proposals

num\_proposals = num\_req\_samps + round(num\_req\_samps\*(bound))

#Rejection algorithm

x = runif(num\_proposals)

u = runif(num\_proposals)

accept <- u <= (f\_density(x, a, b)/bound)

x\_accept = x[accept]

x\_accept = x\_accept[1:num\_req\_samps]

x\_accept

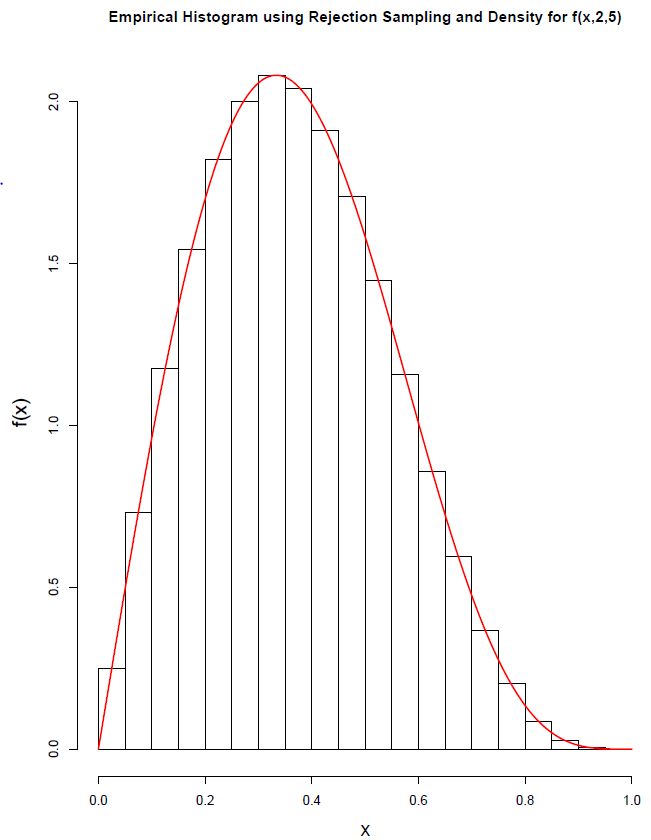
}

Note that an assumption made in this implementation of rejection sampling is that we can analytically compute the bound by finding the maximum of the curve which may not always be feasible.

ii. To generate 1,000,000 samples from the density involves simply executing the following;

fx\_is = rejction\_sampler(2, 5, 1000000)

The required density can be visualised using the hist command. The empirical histogram of the density and the true target density are plotted together in figure 1c illustrating the application of rejection sampling.



*Figure 1c. Rejection sampling for (n samples = 1,000,000)*

*Part d Comment*

It could be argued that both the inversion\_sampler function and the rejection\_sampler function are relatively efficient given that they generate a sample size to the order of 6 (1 million samples) in under 3 seconds as displayed in Table 1d. The functions were created without the use of any explicit loops in a effort to reduce the overall run time. The elapsed times were found in R using the the sys.time() function. Each time corresponds to the average elapsed time of three runs of the function. As an example, the time required to execute one run of the inversion\_sampler would be;

start\_time = Sys.time()

fu = inversion\_sampler(a\_is, 5, 1000000)

end\_time = Sys.time()

time\_elapsed = end\_time - start\_time

In comparing the two functions, it seems the inversion\_sampler function is more efficient than the rejection\_sampler function. The latter function contains significantly more computations including determining the roots of the function, which could explain this.

*Table 1d. Elapsed time required to run the sampling functions*

|  |  |  |
| --- | --- | --- |
| Function | Inversion Sampling timing  (s) | Rejection Sampling timing  (s) |
|  | 0.25 | 0.29 |
|  | 0.15 | - |

Due to time constraints no other combinations of alpha and beta were trialled however both the inversion\_sampler function and rejection\_sampler function allow the user to input any possible values of alpha and beta, however it cannot be said without further testing whether a correct solution could be obtained for all parameterisations.

*Part e.*

To get a sample from as below, the function h\_sampler was written.

 (5)

Let

Given that it is a sample from the max of two normalised densities the following must hold;

Whereby the lower bound is given when there is no overlap between the two functions and upper bound is given where there is complete overlap.

To sample from the principles of a ‘coin toss’ will be used, i.e that of a Bernoulli distribution whereby the probability that we will draw a sample from is ½ and the is also ½. The pseudo code of the h\_sampler function is given below;

Pseudocode

***Repeat;***

1. Generate variables according to following distributions;

1. Implement ‘Coin Toss’ to take a samples from either or ;

***Until;*** U is less than whichever function is a maximum at w, normalized by the total mass at w

Evaluate functions at w and accept sample if above holds

***Then***

z = w

The function in R was implemented as follows;

h\_sampler2 = function(g, num\_samps){

#1. Get two target densities

h1 = inversion\_sampler(2, 1/g, num\_samps)

h2 = inversion\_sampler(2, g, num\_samps)

#Coin toss between two distributions (Bernoulli)

i = runif(num\_samps)

w = h1\*(i>1/2) + h2\*(i<1/2)

# Threshold

u = runif(num\_samps)

#If U is less than whichever function is a maximum at w, normalized by the total mass at w

threshold=(max(f\_density(w, 2, 1/g), f\_density(w, 2, g)))/(f\_density(w, 2, 1/g) +f\_density(w, 2, g))

accept\_w <- u <= threshold

#Accepted samples

x\_samp = w[accept\_w]

x\_samp

}

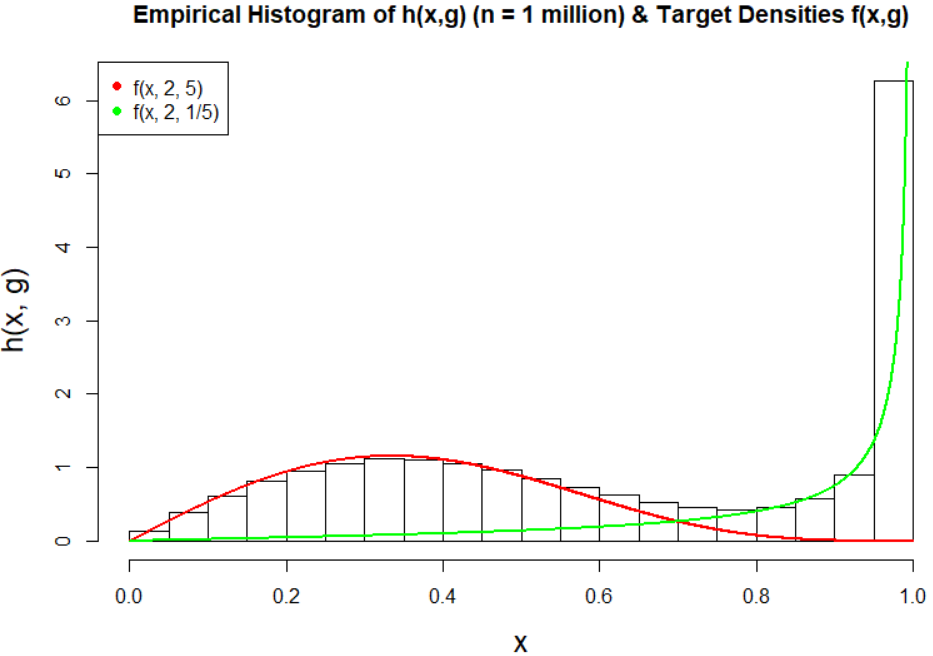
(ii).The empirical histogram of the density and the true target densities are plotted together in figure 1e. The density plots are scaled according to the total mass of the two plots given by;

Normalizing factor = = 1.83950

The CDF is given by (2) above while is the point of intersection of the two functions found analytically, by equating *f1* and *f2* and solving for x, to be;

In this case for the point of intersection was found to be 0.699 for which the normalizing factor above was found.

As shown in figure 1e the h\_sampler effectively samples from ,



*Figure 1e. Empirical histogram of and target densities*

Part e (iii)

It seems that the function h\_sampler effectively samples from as shown in figure 1e. The function was firstly written in a loop and then recreated without in order to reduce the run time. The total run time is shown in table 1e. Note that as we increasing gamma the timing increases. This is perhaps unsurprising, as if we liken the function to a gamma distribution, the second parameter ( is the scale parameter since most of it’s influence is on the spread of the data, therefore you would suspect it would take more time to sample from a distrubtion of greater spread (larger gamma) perhaps.

*Table 1e*

|  |  |
| --- | --- |
|  | h\_sampler timing  (s) |
|  | 1.00 |
|  | 1.04 |
|  | 1.16 |

*Question 2*

The region of interest in question 2 is defined as follows;

 (I)

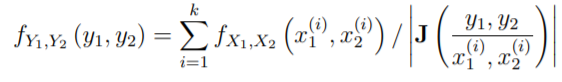
*Part a Proof*

We need to show that;

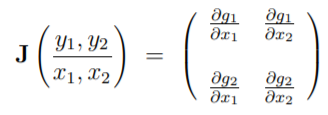
If

Then has density where

That is we need the pdf of a transformed random vector Y in given that we know the pdf, fX(x), of the original random vector X.1 This can be achieved using the following relation;

 (II)

And the  **J** is as follows**;**



In our case we have;

and

Therefore expression II becomes;

/ (III)

Since we are showing this for (U,V) ~ Unif(A) then we know that for a uniform distribution on a two dimenstional set2;

Therefore expression III becomes;

/ =

=

Since is a constant and then;

*So*

or where

Q.E.D

*Part b Boundary Conditions*

To provide conditions on f such that A can be enclosed in a rectangle of finite area B requires us to find the limits of this rectangle B3. That is we need to find bounds for the function and where we will let;

and

Since we only need the upper bounds and the lower bound for given by

To find these boundaries requires that we maximise these functions, that is the boundaries of the region B can be written as follows;

To find the maximum of these functions is not trivial so the boundaries were defined as above.

*Part c*

(i) The density of interest is as follows;

 (II)

The function fy\_eval was written in R to evaluate given as follows;

fy\_eval = function(y){

fy = exp(-(y-2)^2) + exp(-(y+2)^2)

fy

}

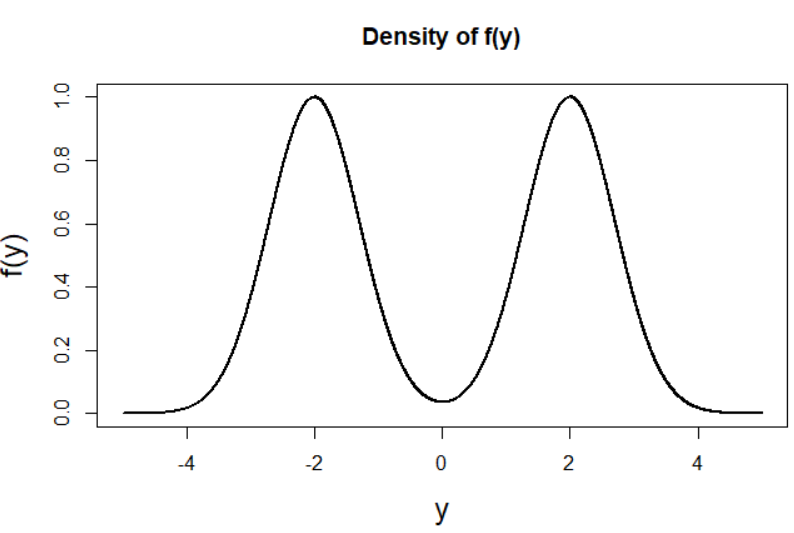
To evaluate and plot up to it’s normalizing constant the following was executed

y = seq(0,1, length = 5000)

fy = fy\_eval(y)

plot(y, fy)

The plot is shown in figure 2c below



*Figure 2.c Density plot of f(y)*

Part c (ii)

The function A\_eval was written to evaluate and visualise the region (given by I). The following pseudocode explains the

*Pseudocode of function A\_eval which is used to evaluate and visualise the region A*

*A\_eval(u.bounds, v.bounds, dimension\_of\_variables)*

1. Initialise the variables u and v
   * u and v were defined using the seq function. The bounds of both u and v are left as user-defined inputs to the function to allow for different bound variations to be easily trialled.
   * Initially no constraints were put on v (for e.g a sequence from -10 to 10 of length 1000 was created) while in the case of u the only constraint on that the initial/minimum value was 0 (for e.g a sequence from 0 to 10 of length 1000 was created)
2. Evaluate *f(v/u)* for every possible combination of u and v
   * A data frame with columns u and v was created using the expand.grid function in R to encompass all possible combinations of u and v in the rows.
   * The function (given by fy\_eval as defined in c(i)) was evaluated for all possible combinations of u and v, specifically and the result was appended as a column to the dataframe.
3. Check which values of u meet the constraint
   * A logical variable called ‘check’ was found and appended to the data frame whereby;
4. Plot region A (u, v)

* Plot u against v and colour the plot according to whether the boundary check is satisfied or not

The function A\_eval below is the realisation of the pseudocode as explained above

**A\_eval** <- function(ubnds, vbnds, dimension1){

#1. Setup variables

v = seq(ubnds[1],ubnds[2], length = dimension1)

u = seq(vbnds[1], vbnds[2], length = dimension1)

#2. Create dataframe

data <- expand.grid(v=v, u=u)

#3. Evaluate function at y = v/u

data <- within(data, f <- sqrt(fy\_eval(v / u)))

#4. Check where boundary constraints met i.e 0 <= u <= fy(v/u)

data$check <- (data$u <= data$f) & (data$u >= 0)

#5. Plot v vs u and colour according to whether the boundary check satisfied or not

plot(v ~ u, data = data, col = data$check + 1,

pch = 16, cex.lab=1.5, main = 'Visualisation of region A')

legend('topleft', legend = c('Not A', 'A'), col = 1:2, pch = 16)

}

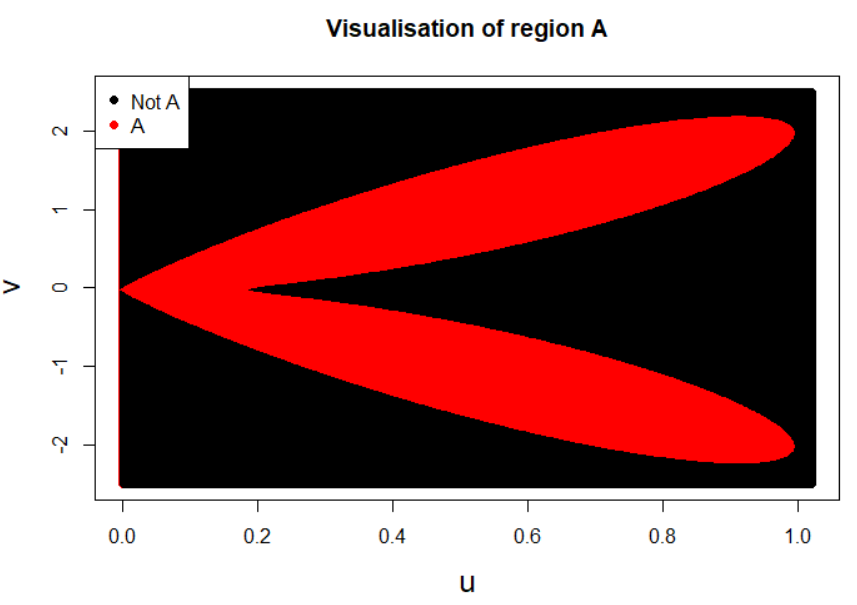
To utiltise the function and produce the plot as shown in Figure 2c(ii) the code below was executed. As explained in the pseudocode, different values for the ranges of u and v were firstly trialled, however upon visualisation of the region, the bounds as set below were deemed sufficient.

#Evaluate function

ubnds = c(-2.5, 2.5)

vbnds = c(0, 1.02)

A\_eval(ubnds, vbnds, 500)



*Figure 2cii. Visualisation of region A*

*Appropriate bounds*

Upon evaluation and visualisation of region A, the following bounds were deemed appropriate;

(III)

*Part d. Samples from*

To generate samples from the density using only randomness from the runif command, the function A\_sampler was created as below. A uniform sample is taken over the area of the rectangle given by bounds as defined in c (given by III). Samples are accepted if they meet the constraint on the region of boundary A, i.e . Note that the lower bound on u is set to be 0 so that constraint is always met. It was found that to generate the required amount of samples the number of proposals had to be increased by a factor of 3.9. The function A\_sampler is as follows;

A\_sampler = function(num\_samps, ubnds, vbnds){

#1. Variables

us = c() #Initialise empty vector

vs = c() #Initialise empty vector

num\_samps = num\_proposals\*3.9 #Increase num of samples by a factor of 3.9

#2. Uniformly sample over Rectangle

u = runif(num\_samps, ubnds[1], ubnds[2]) #uniform [0, 1]

v = runif(num\_samps, vbnds[1], vbnds[2]) #uniform [-2.5, 2.5]

#3. Accept samples based on constraint on the boundary of region A

accept <- u <= sqrt(fy\_eval(v/u))

us = u[accept]

vs = v[accept]

#Return region A i.e (u,v)

return(list(us = us,vs = vs))

}

*Part e Execution & Visualisation*

To obtain 1,000,000 samples from the density the following code was executed;

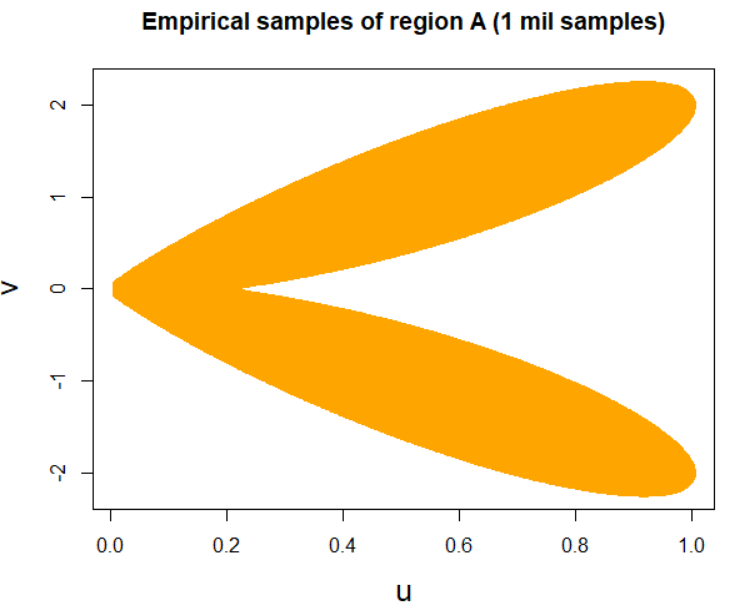
ubnds = c(0, 1.02) #Bounds for u

vbnds = c(-2.5, 2.5) #Bounds for v

num\_proposals = 1000000

As = A\_sampler(num\_proposals, ubnds, vbnds)

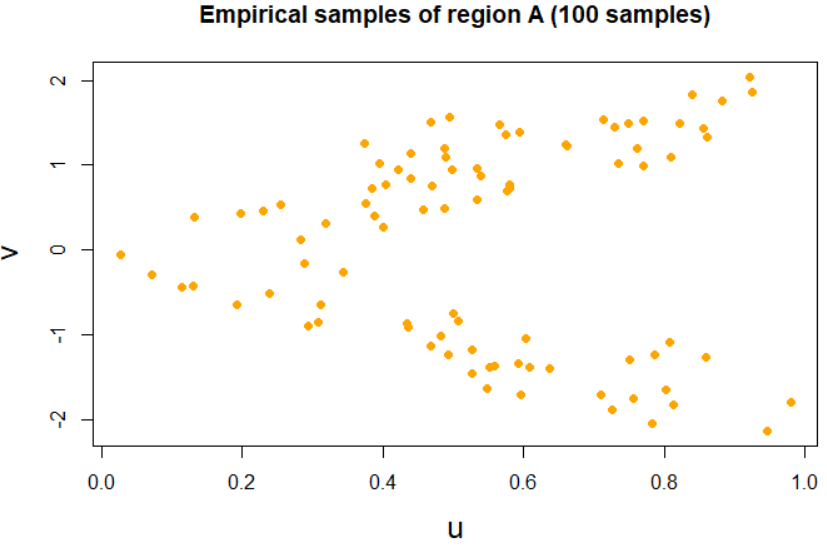
The resultant samples are visualised in Figure 2ei.



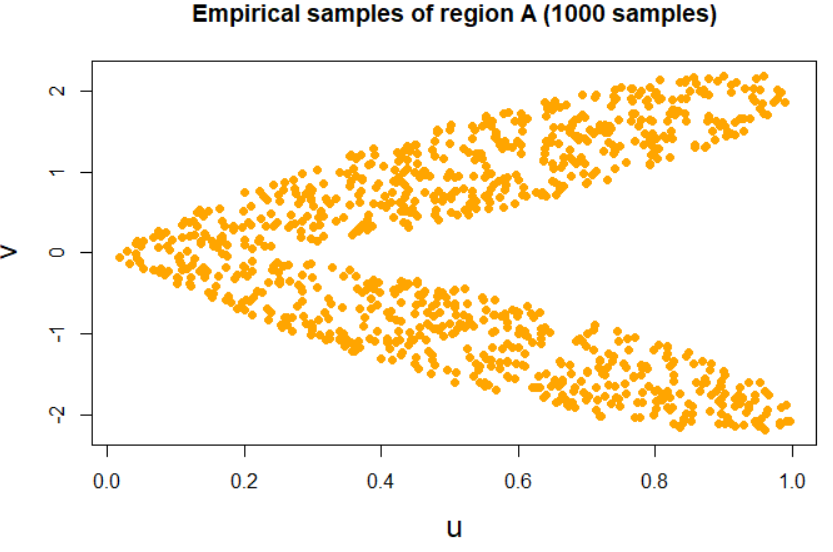
*Figure 2ei. 1,000,000 samples from the density generated using the function A\_sampler*

*Sampling function illustration*

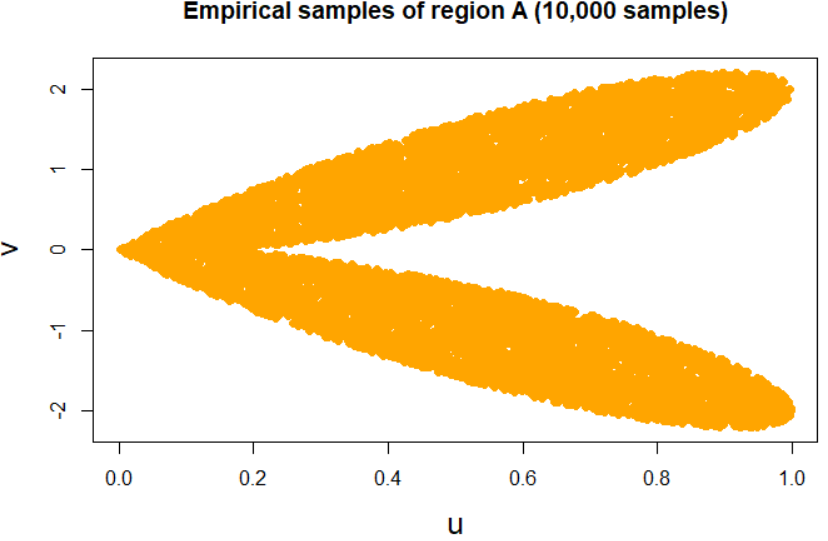
To illustrate the workings of the sampling function A\_sampler and to emphasise the difference between the empirical samples and analytic solution, the function was implemented for sample sizes of increasing orders of magnitude starting at an order of 2 (Figure 2eii), an order of 3 (Figure 2eii), an order or 4, (Figure 2eiii), an order of 5 (Figure 2eiv), while an order of 6 was previously implemented as shown in Figure 2ei.



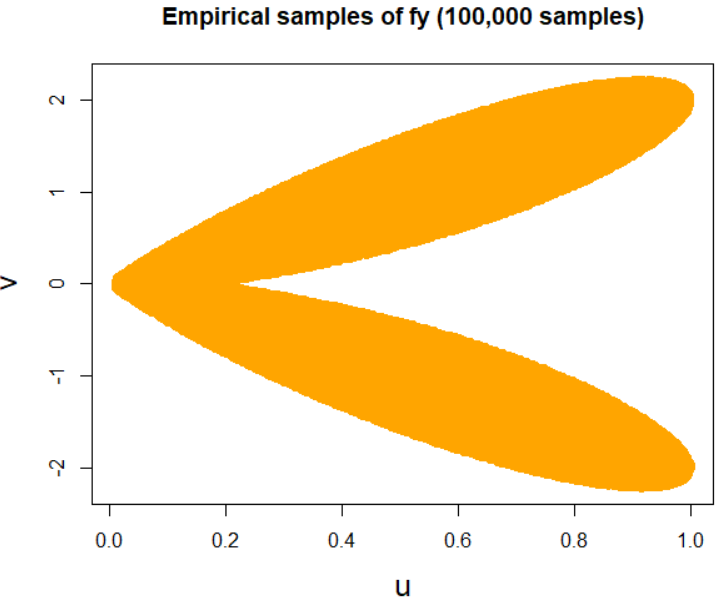
*Figure 2eii. Order of magnitude 2 (n = 100)*



*Figure 2eiii Order of magnitude 3 (n = 1000)*



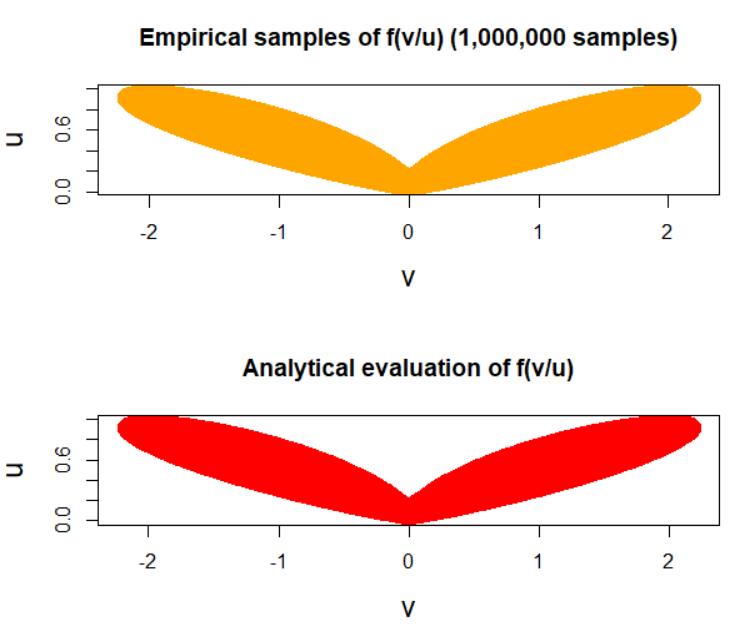
*Figure 2eiv Order of magnitude 4 (n = 10000)*



*Figure 2ev. Order of magnitude 5 (n = 100,000)*

*Comparison*

When the empirical samples generated using the A\_sampler function are compared with the analytic solution found in part c by plotting the results on matching scales, the result is identical, is shown in Figure 2eii.



*Figure 2evi. Comparison of Empirical samples and Analytic evalutation*

*Efficiency & Practicality*

If the solution to the boundaries in part b had been found, this would have enabled us to draw uniformly from the exact rectangle B and therefore from the target density A. This would likely have improved the efficiency of the algorithm as it would reduce the amount of required proposal samples. As for a given sample size requirement, the above algorithm requires 4 times the amount of proposals which is a significant factor so therefore not the most practical.

However it could be argued that the function A\_sampler is relatively efficient given it generates a sample size to the order of 6 (1 million samples) in under 3 seconds as displayed in Table 2e. The function was created without the use of any explicit loop in an effort to reduce the overall run time. The elapsed times were found in R using the sys.time() function. The time corresponds to the average elapsed time of three runs of the function.

Table 2e

|  |  |  |
| --- | --- | --- |
| Function | Number of Samples | Sampling function timing (s) |
|  | 1,000,000 | 1.09 sec |

*References*

1. ‘*Transformation of Random Vectors’* (2019), at <http://ece-research.unm.edu/bsanthan/ece340/note1.pdf>
2. ‘*Probability Theory with Simulations - Two-dimensional continuous distributions’* (2019) at <http://math.bme.hu/~vetier/df/Part-IV.pdf>
3. Martino. L, Miguez. J (2010) ‘*A rejection sampling scheme for posterior probability distributions via the ratio-of-uniforms method’* Universidad Carlos III de Madrid.