Stochastic Simulation Coursework

Nikolai Krokhin

19/11/2021

```
knitr::opts_chunk$set(warning=FALSE, message=FALSE)
suppressMessages(library(tidyverse,verbose=FALSE,warn.conflicts=FALSE,quietly=TRUE))
suppressMessages(library(forecast,verbose=FALSE,warn.conflicts =FALSE,quietly=TRUE))
library(knitr)
library(ggplot2)
library(forecast)
library(goftest)
library(BoutrosLab.plotting.general)
```

```
## Loading required package: lattice

## Loading required package: latticeExtra

##
## Attaching package: 'latticeExtra'

## The following object is masked from 'package:ggplot2':
##
##     layer

## Loading required package: cluster

## Loading required package: hexbin

## Loading required package: grid

##
## Attaching package: 'BoutrosLab.plotting.general'

## The following object is masked from 'package:stats':
##
##     dist
```

(largest prime factor 1453 params 0,2,5,-0.6) CID:01724711 1. We provide a method of generating from a distribution with pdf

$$f_X(x) \propto \frac{1}{x(2-x)} \mathrm{e}^{-0.2(-0.6+\log(\frac{x}{2-x}))^2}, \quad 0 < x < 2,$$

using ratio of uniform sampling. i.e. $h(x) = \frac{1}{x(2-x)} \mathrm{e}^{-0.2(-0.6+\log(\frac{x}{2-x}))^2}$. where $h(\cdot) \geq 0$ and $\int h < \infty$

Generating from $f_X(x)$ using Ratio of Uniform

For the general ratio of uniform sampling algorithm to simulate a random variable $X \sim f_X(x)$, we check that $h(x)$ and $x^2 h(x)$ are bounded in the domain of x so that the bounding rectangle exists. #insert proof of boundedness

General Ratio of Uniform Algorithm:

1. Generate bounding rectangle $C_h = \{(u,v) : 0 \leq u \leq \sqrt{h(\frac{v}{u})}\} = [0,a] \times [b,c]$ where $a = \sup_x \sqrt{h(x)}, b = \inf_{x \leq 0} x\sqrt{h(x)}, c = \sup_{x \geq 0} x\sqrt{h(x)}$

2. Generate $(U_1, U_2) \sim U(0,1)$.

3. Let $U = aU_1$, $V = b + (c-b)U_2$.

4. If $U \leq \sqrt{h(\frac{V}{U})}$, set $X = \frac{V}{U}$, otherwise GOTO 2

If,

$$h(x) = \frac{1}{x(2-x)} \mathrm{e}^{-0.2(-0.6+\log(\frac{x}{2-x}))^2}, \quad 0 < x < 2$$

We find that $a = 1.32566$, $b = 0$, $c = 2.521868$ below.

```
afunc <- function(x){sqrt(1/(x*(2-x))*exp(-0.2*(-0.6+log(x/(2-x)))^2))}
bfunc <- function(x){x*sqrt(1/(x*(2-x))*exp(-0.2*(-0.6+log(x/(2-x)))^2))}

a = optimise(afunc, c(0,2), maximum = TRUE)[[2]]
b <- bfunc(1.0e-52)
c = optimise(bfunc, c(0,2), maximum = TRUE)[[2]]


cat("a:",a, "b:",b,"c:",c)
```

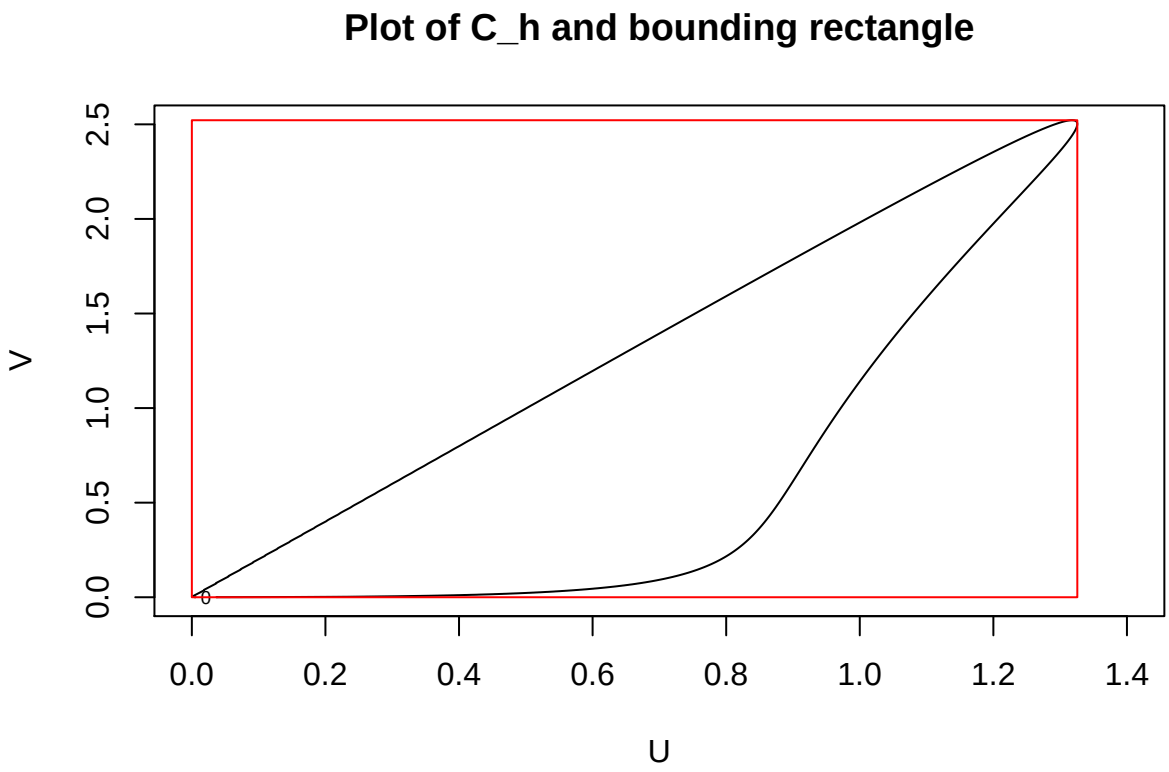
```
## a: 1.32566 b: 0 c: 2.521868
```

Here we define $h(x)$ over its whole domain and vectorize it in order to have an efficient algorithm for generating based on vectors and not loops.

```
h1<-function(x){
  if (x<=0 | x>=2){0}

  else{1/(x*(2-x))*exp(-0.2*(-0.6+log(x/(2-x)))^2)}
}
h <- Vectorize(h1, vectorize.args = "x")
```

The next step we take is crucial for when we decide on how to improve our generation technique and it will become apparent why later. We visualize C_h and the bounding rectangle to verify that we have calculated a, b, c correctly.

```
my.fun1 <- function(x,y){sqrt(h(y/x)) - x} #creating our border of C_h function
x<-seq(1.0e-6,1.4,length=1000)
y<-seq(1.0e-6,2.6,length=1000)
z<-outer(x,y,my.fun1)  #calculating surface based on outer product of x, y using border func
contour(x,y,z,level=0, main="Plot of C_h and bounding rectangle", xlab="U", ylab="V", xlim = c(0,1.4),
ylim = c(0, 2.5))#we plot just the 0th layer of our surface to get desired contour
rect(0, b, a, c, border = "red")
```




```
par(new=TRUE) #plot of rectangle and C_h on same pair of axes
```

Implementing Ratio of Uniform Algorithm:

The tricky part of generating n values of X from an algorithm that rejects values that were generated is that we are required to use a while loop until we have the desired number of generates. The issue is that looping, generating a point (U, V) and checking if it belongs to C_h until we have n successful attempts, is incredibly inefficient. So we seek to utilize vectorized manipulation as much as possible, which is very efficient in R. We may begin by building a function that creates two vectors of independent random uniforms of length k and from this we create our points (U, V) in our bounding rectangle, as outlined in the general algorithm. We then slice the potential X values where $X = \frac{V}{U}$ based on the condition that $U \leq \sqrt{h(\frac{V}{U})}$. However, this does not return k X values but only a fraction of k. The number of X values it returns on average is kp where p is our acceptance probability. The trick we can use is that if initially the two vectors of independent random uniforms are generated of length $\frac{k}{p}$ then we will on average generate $\frac{k}{p}p = k$ X values. To do this we require to find p which is given by $p = \frac{0.5\int h(x)dx}{a(c-b)}$, derived from area of C_h divided by area of rectangle.

```
print(0.5 * integrate(h,0,2)[[1]] / (a * (c-b)))
```

```
## [1] 0.2963777
```

We numerically find $p = 0.296377715034856$. So when we call the function we may in reality get more or less or exactly the right number of generates. If we get the exact right number of generates we are done. If we make too many generates we just slice off the end of not needed generates and we are done. If we want n generates and the first iteration generates less than n, say $n - i$, then we still need i generates. We can just call the generation function with uniform random vectors of length $\frac{i}{p}$ to on average get these i remaining generates. We continue in similar fashion until we have n generates.

```r
gen_1 <- function(n){ #input how many generates we want and the acceptance probability
  inside_genvec <- function(n){
    u1 <- runif(n)
    u2 <- runif(n)
    u = a * u1 #creating (u,v) points in rectangle
    v = b + (c - b) * u2
    pot_x <- v/u
    pot_x <- pot_x[u < sqrt(h(pot_x))] #slice and keep only those inside C_h
    return(pot_x)
  }
    vals<-c()
  p <- c()
  while (length(vals) < n) {
    n_to_gen <- ceiling((n - length(vals))/0.296377715034856)
    x <- inside_genvec(n_to_gen)
    vals <- c(vals, x)
    p <- c(p, length(x)/n_to_gen) # keep track of estimated acceptance prob
  }
  return(list(x=vals[1:n], p=p))
}
```

Here we illustrate the efficiency of our vectorized approach by looking at the number of loops necessary to make the n generates by creating a counter and also looking at how many excess generates we made that we do not want. We repeat 10 times for each n to see the consistency. Observe that rarely ever is the number of loops needed over 4 and at least half of the time we only need one loop to generate our sample. Further, the number of extras generated is small compared to the sample size and so is basically adding on no computational time. Overall, the small number of loops needed and small number of extras generated means that this algorithm is much more efficient that looping n number of times and having 0 extras.

```r
gen_1_count_test <- function(n){ #counting number of loops in while
  inside_genvec <- function(n){
    u1 <- runif(n)
    u2 <- runif(n)
    u = a * u1
    v = b + (c - b) * u2
    pot_x <- v/u
    pot_x <- pot_x[u < sqrt(h(pot_x))]
    return(pot_x)
  }
  vals <- c()
  count<-0  #initialize count
  while (length(vals) < n) {
    vals <- c(vals, inside_genvec(ceiling((n - length(vals))/0.296377715034856)))
    count<-count+1 #increment
  }
  return(count)
}
for (j in 1:5){
  counts<-c()
  for (i in 1:10){
    counts <- c(counts,gen_1_count_test(10**j))}
  cat("Counts to generate", 10**j, "samples (on 10 occassions of generating):",counts,"\n")
}
```

```
## Counts to generate 10 samples (on 10 occassions of generating): 2 1 1 1 1 1 2 1 1 1
```

```
## Counts to generate 100 samples (on 10 occassions of generating): 3 1 2 2 2 2 1 1 1 1
## Counts to generate 1000 samples (on 10 occassions of generating): 2 3 2 2 4 1 1 1 1 1
## Counts to generate 10000 samples (on 10 occassions of generating): 2 1 1 1 1 2 5 1 1 2
## Counts to generate 1e+05 samples (on 10 occassions of generating): 4 2 1 2 1 2 3 1 4 1
```

```
gen_1_extras_test <- function(n){ #counting extra X generates
  inside_genvec <- function(n){
    u1 <- runif(n)
    u2 <- runif(n)
    u = a * u1
    v = b + (c - b) * u2
    pot_x <- v/u
    pot_x <- pot_x[u < sqrt(h(pot_x))]
    return(pot_x)
  }
  vals <- c()
  while (length(vals) < n) {
    vals <- c(vals, inside_genvec(ceiling((n - length(vals))/0.296377715034856)))
  }
  return(length(vals)-n) #number of extras
}
for (j in 1:5){
  counts<-c()
  for (i in 1:10){
    counts <- c(counts,gen_1_extras_test(10**j))}
  cat("Extras in generating", 10**j, "samples (on 10 occassions):",counts,"\n")
}
```

```
## Extras in generating 10 samples (on 10 occassions): 5 1 2 2 1 1 2 0 0 0
## Extras in generating 100 samples (on 10 occassions): 1 1 1 0 1 1 1 0 4 4
## Extras in generating 1000 samples (on 10 occassions): 1 2 22 12 9 40 0 9 2 10
## Extras in generating 10000 samples (on 10 occassions): 2 68 33 2 5 6 2 57 0 0
## Extras in generating 1e+05 samples (on 10 occassions): 9 5 8 15 115 198 1 4 144 0
```

Implementing Pre-test Squeezing for Ratio of Uniform Algorithm:

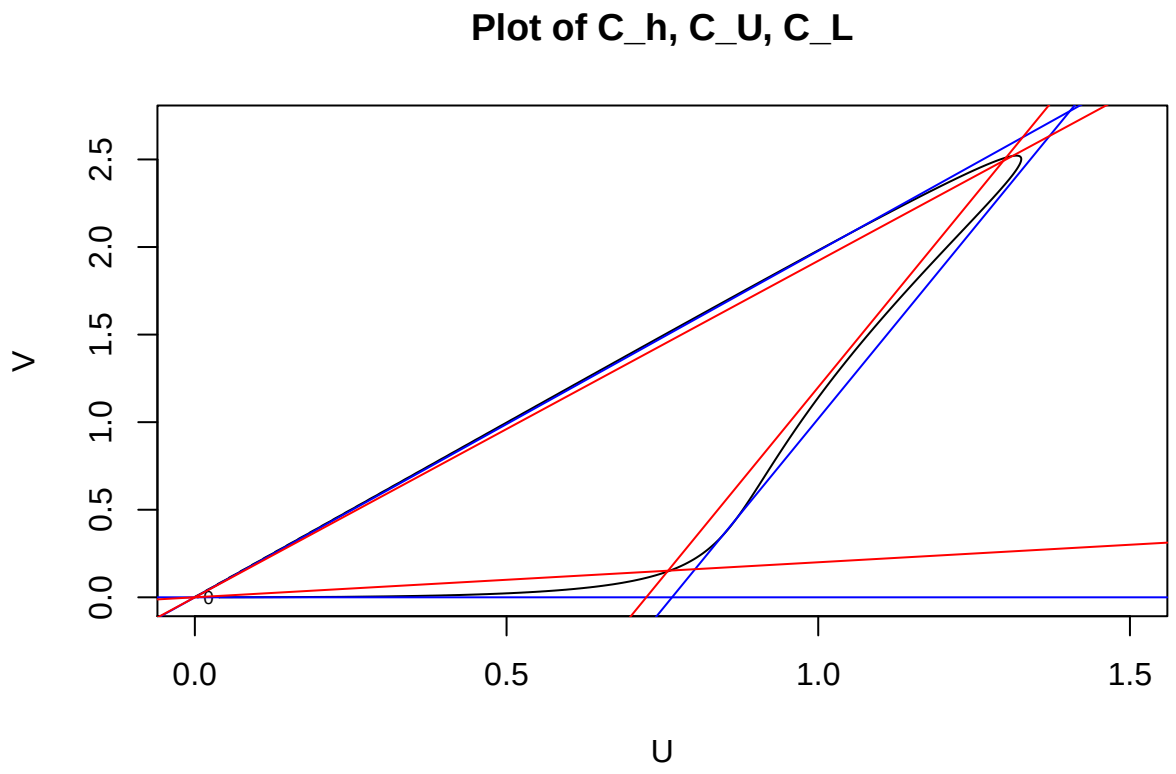
Our $h(x)$ is computationally very demanding and so we seek to implement pre-test squeezing on the above algorithm. Although this does not change the acceptance probability the computational cost of the algorithm will decrease and so we hope to see a shorter time for this algorithm to run. Now is when come back to the plot of C_h and exploit the triangular shape that we notice. Via inspection we can define an upper bounding triangle and a lower bounding triangle.

$$C_U = \{(x, y) : y > 0, y < 1.977784x, y > 4.35x - 3.33\}$$

$$C_L = \{(x, y) : y > 0.2x, y < 1.921208, y > 4.35x - 3.15\}$$

```
my.fun1 <- function(x,y){sqrt(h(y/x)) - x}
x<-seq(1.0e-6,1.4,length=1000)
y<-seq(1.0e-6,2.6,length=1000)
z<-outer(x,y,my.fun1)
contour(x,y,z,level=0, main="Plot of C_h, C_U, C_L", xlab="U",
        ylab="V", xlim = c(0,1.5), ylim = c(0,2.7))
abline(h=0,col='blue')  #upper bound triangle
abline(0, (c+0.1)/a, col='blue')
abline(-3.33, 4.35, col = "blue")
```

```r
abline(0, 0.2, col = "red") #lower bounding triangle
abline(-3.15,4.35, col = "red")
abline(0,(c+0.025)/a, col = "red")
```



The algorithm becomes: 1. Generate bounding rectangle $C_h = \{(u,v) : 0 \leq u \leq \sqrt{h(\frac{v}{u})}\} = [0,a] \times [b,c]$ where $a = \sup_x \sqrt{h(x)}, b = \inf_{x \leq 0} x\sqrt{h(x)}, c = \sup_{x \geq 0} x\sqrt{h(x)}$
2. Generate $(U_1, U_2) \sim U(0,1)$.
3. Let $U = aU_1$, $V = b + (c-b)U_2$.
4. If $(U,V) \notin C_U$ GOTO 2
5. If $(U,V) \in C_L$, set $X = \frac{V}{U}$ END
6. Elif $U \leq \sqrt{h(\frac{V}{U})}$, set $X = \frac{V}{U}$, otherwise GOTO 2

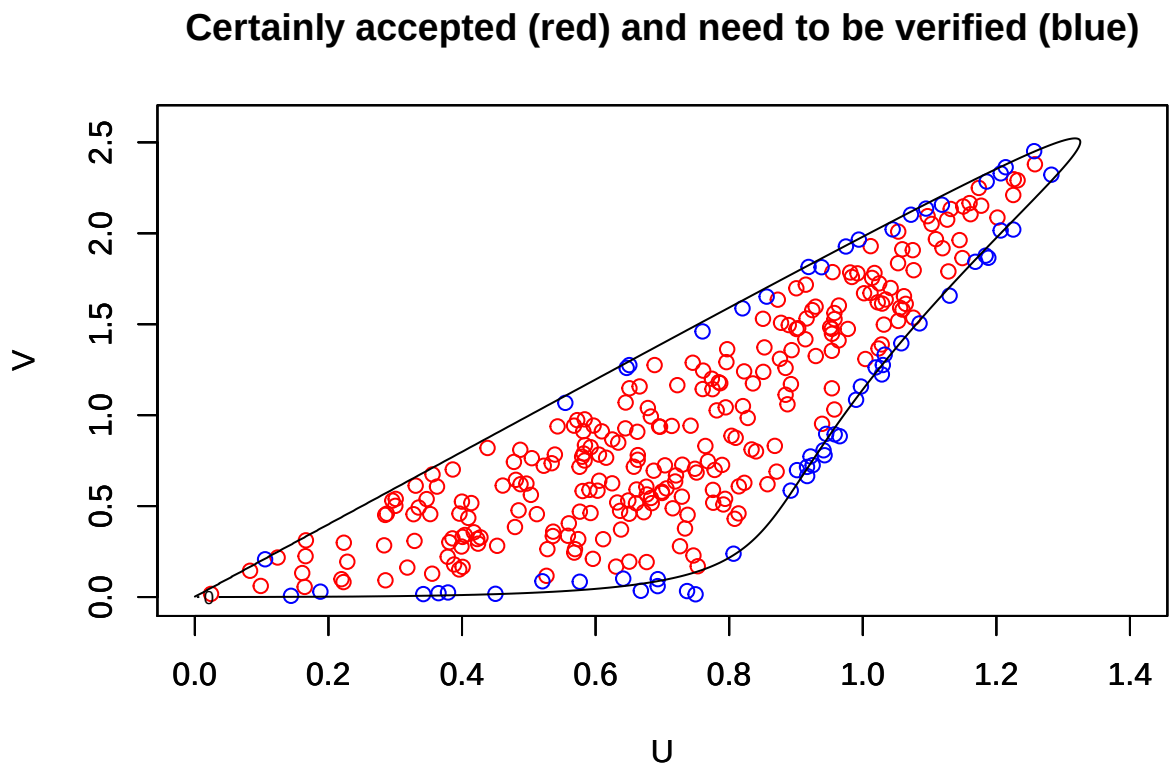
```r
gen_2 <- function(n){
  inside_genvec <- function(n){
    u1 <- runif(n)
    u2 <- runif(n)
    u = a * u1
    v = b + (c - b) * u2
    booloutside <- v > 0 & v < u* (c+0.1)/a & v > 4.35 * u - 3.33 #define upper triangle criterion
    u <- u[booloutside] #remove everything outside upper triangle
    v <- v[booloutside]
    boolinside <- v > 0.2 * u & v < u * (c+0.025)/a & v > 4.35 * u - 3.15
    #define lower triangle criterion
    certainu <- u[boolinside] #definitely acceptable X
    certainv <- v[boolinside]
```

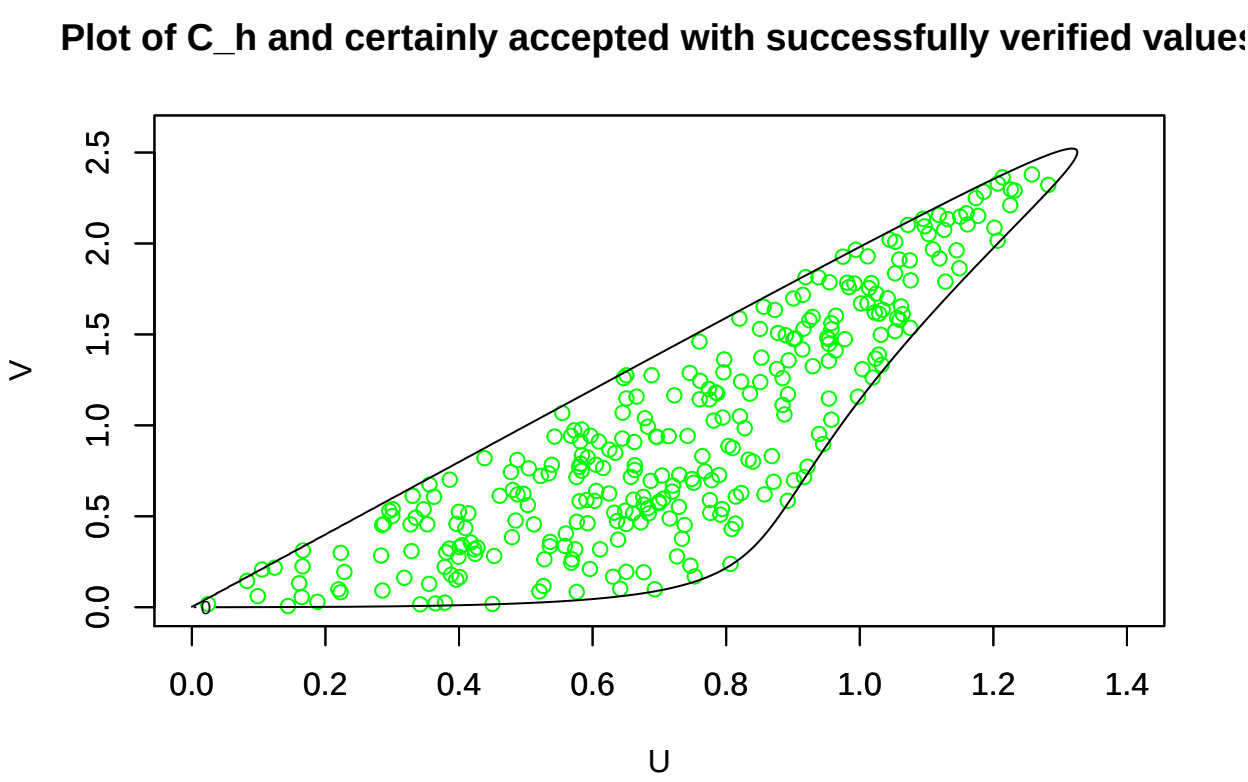
```
    uncertainu <- u[!boolinside] #possibly acceptable X
    uncertainv <- v[!boolinside]
    if (length(uncertainu) == 0){return(certainv/certainu)} #catches a bug on next line with h
    boolfinal <- uncertainu < sqrt(h(uncertainv/uncertainu)) #check possibly acceptable X
    certainu <- c(certainu, uncertainu[boolfinal])
    #add the successful ones from the possibly acceptable X
    certainv <- c(certainv, uncertainv[boolfinal])
    xvals <- certainv/certainu
    return(xvals)
  }
  vals<-c()
  p <- c()
  while (length(vals) < n) {
    n_to_gen <- ceiling((n - length(vals))/0.2963777715034856)
    x <- inside_genvec(n_to_gen)
    vals <- c(vals, x)
    p <- c(p, length(x)/n_to_gen) # keep track of estimated acceptance prob
  }
  return(list(x=vals[1:n], p=p))
}
```

We visually illustrate the process of pre-test squeezing via the two plots below. The first plot shows the initial phase of the squeeze where the only points left are inside the outer triangle. The plot shows red points that are definitely accepted since they are in the inside triangle and the blue points will have to be verified using h. The next plot confirms that the blue points that successfully passed the h test are added to the cohort of the red points.

```
plotsqueeze <- function(n){
    u1 <- runif(n)
    u2 <- runif(n)
    u = a * u1
    v = b + (c - b) * u2
    booloutside <- v > 0 & v < u* (c+0.1)/a & v > 4.35 * u - 3.33 #define upper triangle criterion
    u <- u[booloutside] #remove everything outside upper triangle
    v <- v[booloutside]
    boolinside <- v > 0.2 * u & v < u * (c+0.025)/a & v > 4.35 * u - 3.15
    #define lower triangle criterion
    certainu <- u[boolinside] #definitely acceptable X
    certainv <- v[boolinside]
    plot(certainu, certainv, col ="red", xlim = c(0,1.4), ylim = c(0,2.6), xlab="", ylab ="")
    uncertainu <- u[!boolinside] #possibly acceptable X
    uncertainv <- v[!boolinside]
    par(new=TRUE)
    plot(uncertainu, uncertainv, col ="blue", xlim = c(0,1.4), ylim = c(0,2.6),xlab="U", ylab="V")
    if (length(uncertainu) == 0){return(certainv/certainu)} #catches a bug on next line with h
    boolfinal <- uncertainu < sqrt(h(uncertainv/uncertainu)) #check possibly acceptable X
    certainu <- c(certainu, uncertainu[boolfinal]) #add the successful ones from the
    # possibly acceptable X
    certainv <- c(certainv, uncertainv[boolfinal])
    xvals <- certainv/certainu
    par(new=TRUE)
    my.fun1 <- function(x,y){sqrt(h(y/x)) - x} #creating our border of C_h function
    x<-seq(1.0e-6,1.4,length=1000)
    y<-seq(1.0e-6,2.6,length=1000)
```

```
    z<-outer(x,y,my.fun1)  #calculating surface based on outer product of x, y using border func
    contour(x,y,z,level=0, main="Certainly accepted (red) and need to be verified (blue)",
            xlab="U", ylab="V", xlim =         c(0,1.4), ylim = c(0,2.6))
    #we plot just the 0th layer of our surface to get desired contour

    plot(certainu, certainv, col ="green", xlim = c(0,1.4), ylim = c(0,2.6), xlab="", ylab="")
    par(new=TRUE)
    contour(x,y,z,level=0, main="Plot of C_h and certainly accepted with successfully verified values",
            xlab="U", ylab="V", xlim = c(0,1.4), ylim = c(0,2.6))
    #we plot just the 0th layer of our surface to get desired contour


}
plotsqueeze(1000)
```





Implementing traingle based Ratio of Uniform Algorithm: Another idea to increase the speed of the algorithm is instead of generating the initial (U, V) in the bounding rectangle we can generate (U, V) uniformly inside C_U. Since this will greatly increase the acceptance probability while the computational cost of generating inside the triangle is essentially equivalent to the rectangle. We find the vertices to be $(0, 0), (1.403, 2.776), (0, 0.765)$ and so $p = \frac{0.5 \int h(x)dx}{0.5 \times 0.7655172 \times 2.776}$ using the area of the triangle. This gives $p = 0.9324083$ meaning nearly every point generated inside the triangle will be accepted.

```
print(integrate(h,0,2)[[1]] / (2.7763157823739490838945526039787270636400732479673014599007847514
                               *0.7655172))
```

```
## [1] 0.9324083
```

We have the algorithm (Devroye 1986, p570) as follows to generate a uniformly random # point inside the triangle $((\mathbf{v_0}), (\mathbf{v_1}), (\mathbf{v_2}))$.

1.Generate $R_1, R_2 \sim U(0, 1)$
2. If $R_1 < R_2$ then swap R_1 and R_2.
3. return $(1 - R_1)\mathbf{v}_0 + (R_1 - R_1)\mathbf{v}_1 + R_2\mathbf{v}_2$

Using this our Ratio of Uniform algorithm is now:
Initial: Check if this algorithm is applicable by observing if shape of C_h can be bounded by triangle.
#Setup: identify bounding triangle $((\mathbf{v_0}), (\mathbf{v_1}), (\mathbf{v_2}))$ by inspection of C_h plot. Bounding triangle should look to have as small an area as possible by having tangential sides to C_h. 1. Generate $R_1, R_2 \sim U(0, 1)$
2. If $R_1 < R_2$ then swap R_1 and R_2.
3. $(U, V) = (1 - R_1)\mathbf{v}_0 + (R_1 - R_1)\mathbf{v}_1 + R_2\mathbf{v}_2$
4. If $U \leq \sqrt{h(\frac{V}{U})}$, set $X = \frac{V}{U}$, otherwise GOTO 2

```
gen_3 <- function(n){
  inside_gen_vec <- function(n){
    r1 <- runif(n)
    r2 <- runif(n)
    df <- data.frame(V1=r1, V2=r2) #allows vectorized swapping in this format
    df[df$V1 < df$V2, c("V1", "V2")] <- df[df$V1 < df$V2, c("V2", "V1")]
    #swapping r1 and r2 if r1 < r2
    u1 <- as.vector(df$V1)
    u2 <- as.vector(df$V2)
    u <- (u1 - u2) * 0.7655172 + u2 * 1.404 #generating (U,V) in triangle
    v <- u2 * 2.776 #simplified due to 0s in our vertexes
    pot_x <- v/u
    pot_x <- pot_x[u < sqrt(h(pot_x))]
    return(pot_x)
  }
  vals <- c()
  p <- c()
  while (length(vals) < n) {
    n_to_gen <- ceiling((n - length(vals))/0.932109)
    x <- inside_gen_vec(n_to_gen)
    vals <- c(vals, x)
    p <- c(p, length(x)/n_to_gen) # keep track of estimated acceptance prob
  }
  return(list(x=vals[1:n], p=p))
}
```

Implementing Pre-test Squeezing for triangle based Ratio of Uniform Algorithm:

The final idea is to combine the two previous algorithms to maximise computational speed; Generate the points inside C_U and accept them straight away if they are in C_L. This means we have a high acceptance probability and a lower computational cost. The algorithm is as follows:
1. Generate $R_1, R_2 \sim U(0,1)$
2 .If $R_1 < R_2$ then swap R_1 and R_2.
3. $(U,V) = (1-R_1)\mathbf{v}_0 + (R_1 - R_1)\mathbf{v}_1 + R_2\mathbf{v}_2$
5. If $(U,V) \in C_L$, set $X = \frac{V}{U}$ END
6. Elif $U \leq \sqrt{h(\frac{V}{U})}$, set $X = \frac{V}{U}$, otherwise GOTO 2

```
gen_4 <- function(n){
  inside_gen_vec <- function(n){ #combining the two above algorithms
    r1 <- runif(n)
    r2 <- runif(n)
    df <- data.frame(V1=r1, V2=r2)
    df[df$V1 < df$V2, c("V1", "V2")] <- df[df$V1 < df$V2, c("V2", "V1")] #triangle based algo part
    u1 <- as.vector(df$V1)
    u2 <- as.vector(df$V2)
    u <- (u1 - u2) * 0.7655172 + u2 * 1.404
    v <- u2 * 2.776
    boolinside <- v > 0.2 * u & v < u * (c+0.025)/a & v > 4.35 * u - 3.15  #squeezing algo part
    certainu <- u[boolinside]
    certainv <- v[boolinside]
    uncertainu <- u[!boolinside]
    uncertainv <- v[!boolinside]
```

```
    if (length(uncertainu) == 0){return(certainv/certainu)} #catches a bug on next line with h
    boolfinal <- uncertainu < sqrt(h(uncertainv/uncertainu))
    certainu <- c(certainu, uncertainu[boolfinal])
    certainv <- c(certainv, uncertainv[boolfinal])
    xvals <- certainv/certainu
    return(xvals)
  }
  vals <- c()
  p <- c()
  while (length(vals) < n) {
    n_to_gen <- ceiling((n - length(vals))/0.932109)
    x <- inside_gen_vec(n_to_gen)
    vals <- c(vals, x)
    p <- c(p, length(x)/n_to_gen) # keep track of estimated acceptance prob
  }
  return(list(x=vals[1:n], p=p))
}
```

Speed Comparisons:

We time the 4 algorithms for sample sizes 10, 1000 and 1000000. We observe that for small sample sizes (<1000) we see no real difference in the timings. When we start to generate a large number of samples, for example 1 million, we notice a significant difference between the 4 algorithms. We denote the algorithm by its number by introduction in this paper, order 1 to 4. We see that 1 is is roughly 3X less efficient than 3 which makes sense since the acceptance probability of 1 was 3X less than 3, meaning 3 times as many points need to be initially generated on average using 1. However, the difference between decreasing the bounding region area is not as dramatic as the pre-test squeeze. 2 is 7X more efficient by introduction of the upper and lower bounding triangles removing the computational cost of h. Similarly, 4 is 3X more fast with the pre-test squeeze on the bounding triangle algorithm. As expected we see that maximizing acceptance probability and using pre-test squeezing is the optimum strategy here, only taking 1.0596 seconds on average to generate 1 million X values.

```
for (i in 1:5){
  v1<-c()
  v2<-c()
  v3<-c()
  v4<-c()
  for (j in 1:5){
    v1<-c(v1,system.time({ gen_1(10**i) })[[1]])
    v2<-c(v2,system.time({ gen_2(10**i) })[[1]])
    v3<-c(v3,system.time({ gen_3(10**i) })[[1]])
    v4<-c(v4,system.time({ gen_4(10**i) })[[1]])
  }
  cat("Average time for algorithms for sample size", 10**i,":",
      mean(v1),mean(v2),mean(v3),mean(v4),"\n")
}
```

```
## Average time for algorithms for sample size 10 : 0.006 0.008 0.004 0.006
## Average time for algorithms for sample size 100 : 0 0 0 0.004
## Average time for algorithms for sample size 1000 : 0.008 0 0.004 0.002
## Average time for algorithms for sample size 10000 : 0.038 0.004 0.014 0.006
## Average time for algorithms for sample size 1e+05 : 0.434 0.066 0.166 0.05
```

Acceptance Probability

We also separately test the inside function for generation so see if the true acceptance rate is near to the theoretical acceptance probability for both the bounding rectangle and triangle. First we check the former. We know the theoretical value $p = 0.296377715034856$. For input sizes 10, 100, 1000, 10000, 100000 we take the average acceptance rate from 100 samples. We see that all are correct up to 2 s.f when the input size is >1000.

```
probtest <- function(n){
  u1 <- runif(n)
  u2 <- runif(n)
  u = a * u1
  v = b + (c - b) * u2
  booloutside <- v > 0 & v < u* (c+0.1)/a & v > 4.35 * u - 3.33
  #define upper triangle criterion
  u <- u[booloutside] #remove everything outside upper triangle
  v <- v[booloutside]
  boolinside <- v > 0.2 * u & v < u * (c+0.025)/a & v > 4.35 * u - 3.15
  #define lower triangle criterion
  certainu <- u[boolinside] #definitely acceptable X
  certainv <- v[boolinside]
  uncertainu <- u[!boolinside] #possibly acceptable X
  uncertainv <- v[!boolinside]
  if (length(uncertainu) == 0){return(length(certainv/certainu)/n)} #catches a bug on next line with h
  boolfinal <- uncertainu < sqrt(h(uncertainv/uncertainu)) #check possibly acceptable X
  certainu <- c(certainu, uncertainu[boolfinal]) #add the successful ones from the possibly
  # acceptable X
  certainv <- c(certainv, uncertainv[boolfinal])
  xvals <- certainv/certainu
  return(length(xvals)/n)
}
for (i in 1:5){
  v<-c()
  for (j in 1:100){
    v<-c(v,probtest((10**i)))
  }
  cat("Average acceptance probability of input size", 10**i,":",mean(v),"\n")
}
```

```
## Average acceptance probability of input size 10 : 0.286
## Average acceptance probability of input size 100 : 0.286
## Average acceptance probability of input size 1000 : 0.29303
## Average acceptance probability of input size 10000 : 0.294532
## Average acceptance probability of input size 1e+05 : 0.2942881
```

We repeat for the triangular method. We know the theoretical value $p = 0.932109$ Here we see that on average the true acceptance is just a little lower than anticipated perhaps due to the slight inaccuracy when finding the triangle area carried from defining the triangle by inspection. This means we could actually improve our generation algorithm by using the true probability instead of the theoretical one.

```
probtest2 <- function(n){ #combining the two above algorithms
  r1 <- runif(n)
  r2 <- runif(n)
  df <- data.frame(V1=r1, V2=r2)
  df[df$V1 < df$V2, c("V1", "V2")] <- df[df$V1 < df$V2, c("V2", "V1")] #triangle based algo part
  u1 <- as.vector(df$V1)
```

```
  u2 <- as.vector(df$V2)
  u <- (u1 - u2) * 0.7655172 + u2 * 1.404
  v <- u2 * 2.776
  boolinside <- v > 0.2 * u & v < u * (c+0.025)/a & v > 4.35 * u - 3.15  #squeezing algo part
  certainu <- u[boolinside]
  certainv <- v[boolinside]
  uncertainu <- u[!boolinside]
  uncertainv <- v[!boolinside]
  if (length(uncertainu) == 0){return(length(certainv/certainu)/n)} #catches a bug on next line with h
  boolfinal <- uncertainu < sqrt(h(uncertainv/uncertainu))
  certainu <- c(certainu, uncertainu[boolfinal])
  certainv <- c(certainv, uncertainv[boolfinal])
  xvals <- certainv/certainu
  return(length(xvals)/n)
}
for (i in 1:5){
  v<-c()
  for (j in 1:100){
    v<-c(v,probtest2((10**i)))
  }
  cat("Average acceptance probability of input size", 10**i,":",mean(v),"\n")
}
```

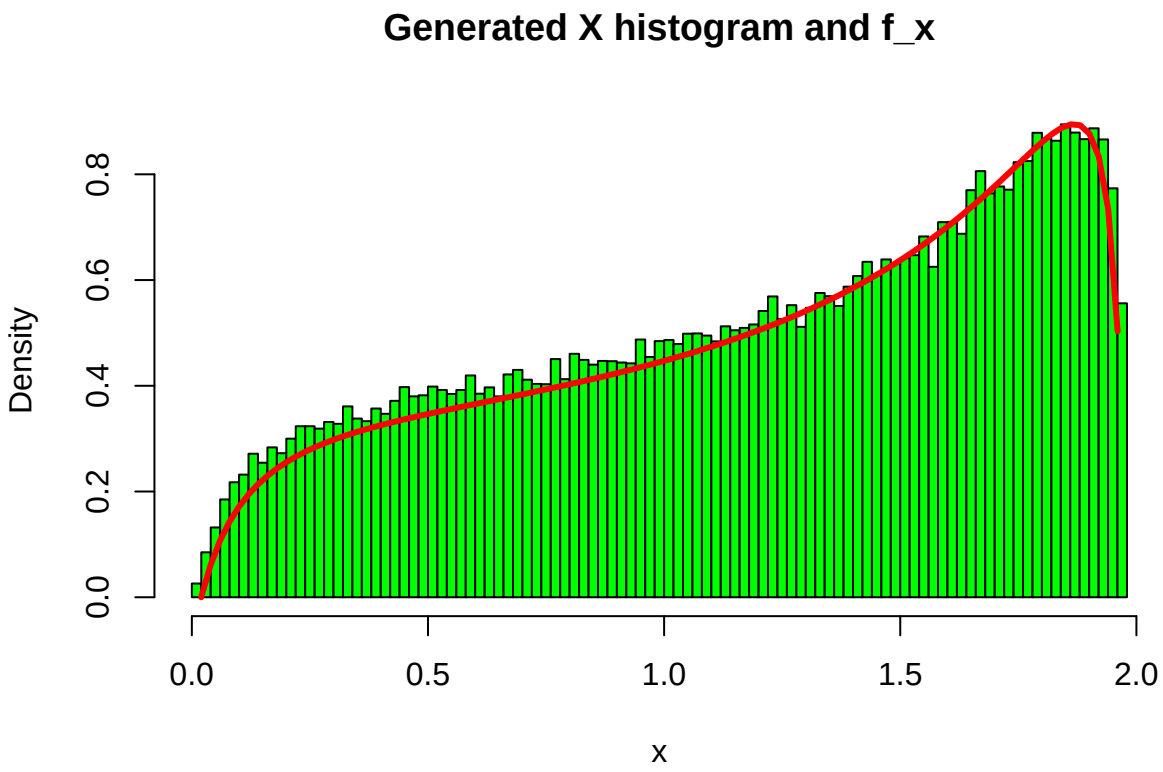
```
## Average acceptance probability of input size 10 : 0.922
## Average acceptance probability of input size 100 : 0.9253
## Average acceptance probability of input size 1000 : 0.92702
## Average acceptance probability of input size 10000 : 0.925627
## Average acceptance probability of input size 1e+05 : 0.925706
```

2.

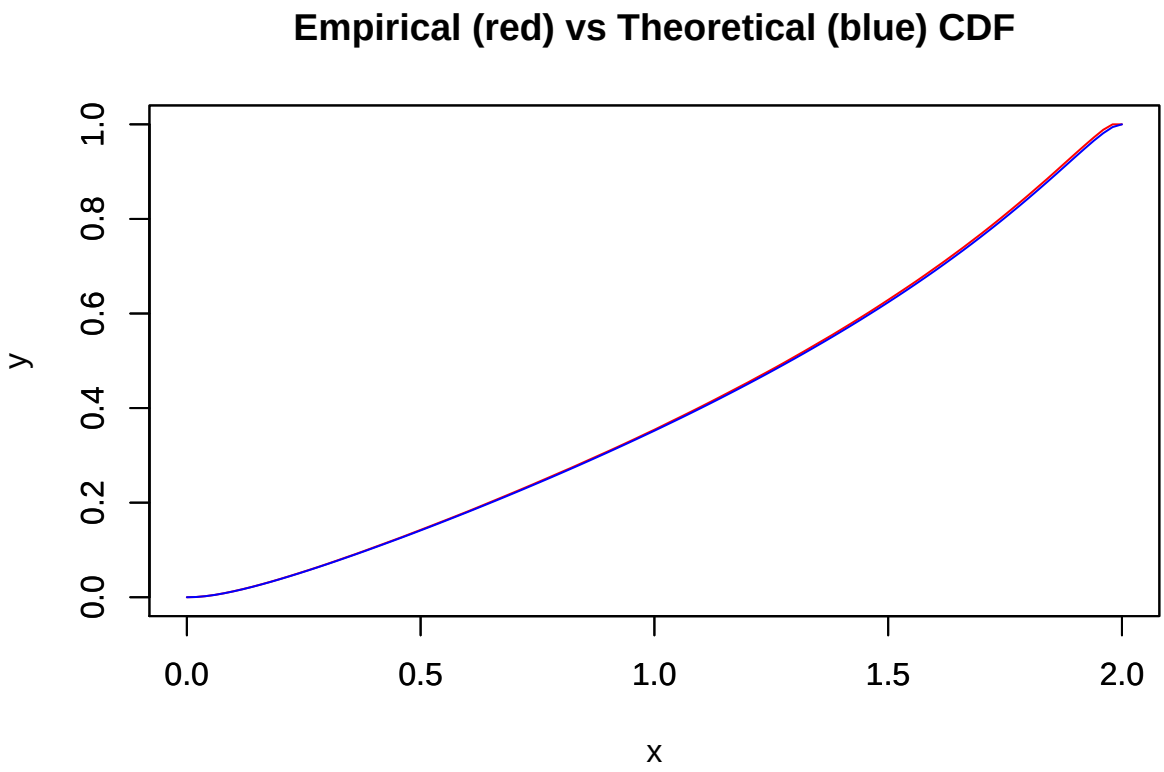
Diagnostic Plots

We produce below some diagnostics plots to demonstrate that the data generated are distributed as independent with pdf $f_X(x)$.

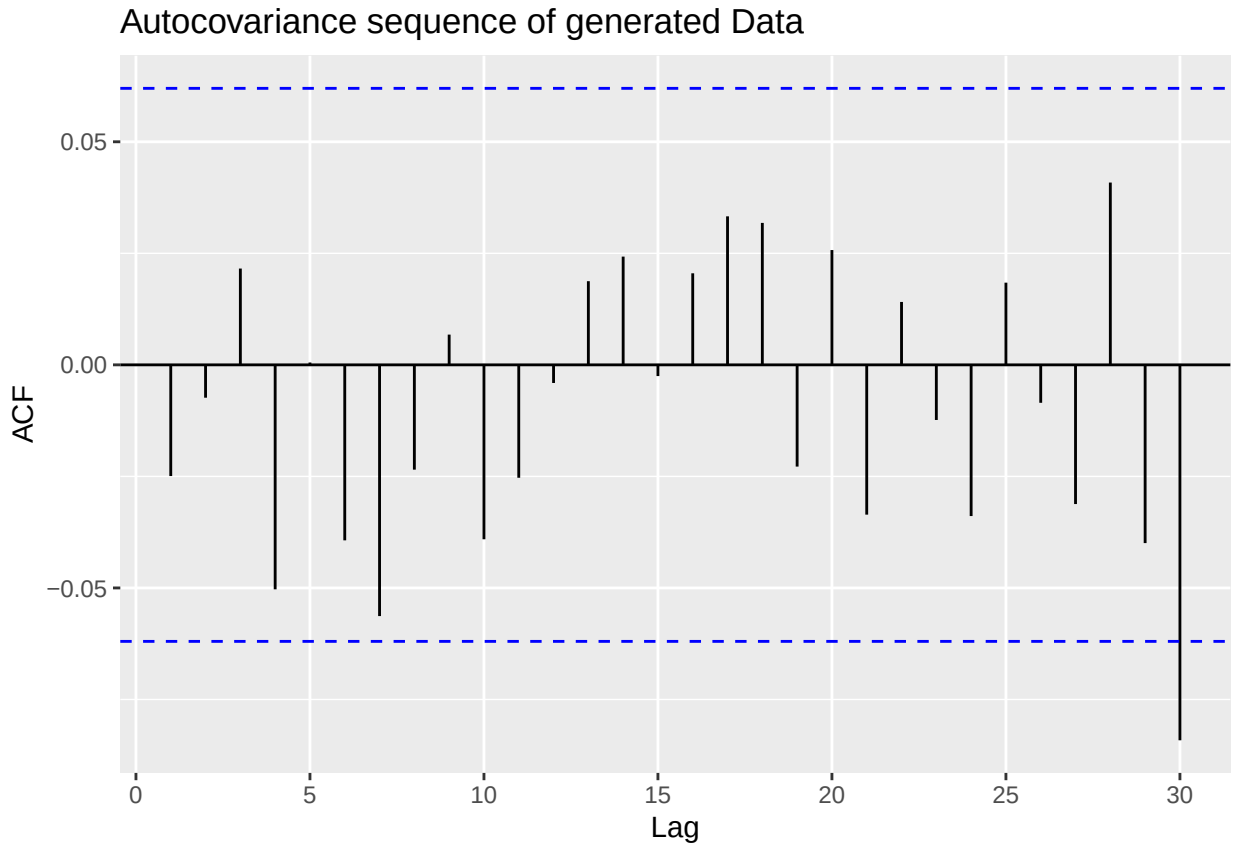
```
plt <- gen_4(100000)
hist(plt$x, breaks=100, col="green", xlab="x",
   main="Generated X histogram and f_x", freq = FALSE)
par(new=TRUE)
h2 <- function(x) {1/(x*(2-x))*exp(-0.2*(-0.6+log(x/(2-x)))^2)}
hint <- integrate(h2,0,2)[[1]]
f <- function(x) {1/(hint*x*(2-x))*exp(-0.2*(-0.6+log(x/(2-x)))^2)} #defining f

curve(f, from=0, to=2, axes = FALSE, xlab = "", ylab = "", col = "red", lwd = 3)
```



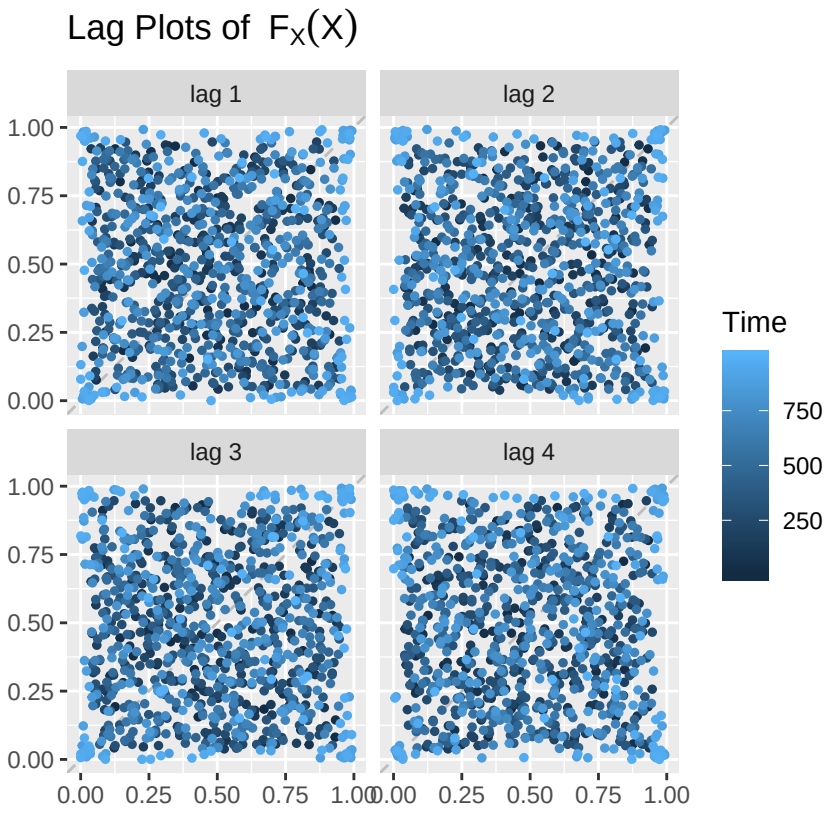

```
n = 1000000
sample1000 <- gen_4(n)
emp_cdf <- function(t){#given a sample of x return number of values less than t
  return(sum(sample1000$x<t,na.rm = TRUE)/n)
}
emp_cdf <- Vectorize(emp_cdf, vectorize.args = "t")

curve(emp_cdf, 1.0e-6,2-1.0e-6, col="red", ylab="y")
f_int <- function(x){integrate(f, 0, x)[[1]]}
f_int <- Vectorize(f_int)
par(new=TRUE)
appCDf <- approxfun(seq(1.0e-6,2-1.0e-6,length.out = 1000),
f_int(seq(1.0e-6,2-1.0e-6,length.out = 1000)))
curve(appCDf, 1.0e-6,2-1.0e-6,col="blue", ylab="", xlab ="",
main = "Empirical (red) vs Theoretical (blue) CDF")
```




```
xvals <- gen_4(1000)
ggAcf(xvals$x)+
  labs(title="Autocovariance sequence of generated Data")
```




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



Via the histogram and the empirical vs theoretical cdf plot, we see that the data closely follows the required theoretical distribution. Furthermore, the auto-covariance plot does not show any signs of significant correlations at any non-zero lag. The lag plots of $F_X(x_i)$, calculated from the cdf, display a random scatter, which additionally supports the data being generated independently from $f_X(\cdot)$ (since if $X \sim f_X(x)$ we have $F_X(X) \sim U(0,1)$).

Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (K-S) test quantifies the difference between the empirical cumulative distribution function (ecdf), $F_{X_n}(x)$ and the theoretical cumulative distribution function, $F_X(x)$. For our density above, we have,

$$F_X(x) = \int_0^x f(x)dx = \int_0^x \frac{1}{x(2-x)\int_0^2 h(x)dx} \mathrm{e}^{-0.2(-0.6+\log(\frac{x}{2-x}))^2}, \quad 0 < x < 2,$$

and the ecdf is defined:

$$F_{X_n}(t) = \frac{\text{number of elements in the sample} \leq t}{n} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{X_i \leq t},$$

The K-S statistics is given by

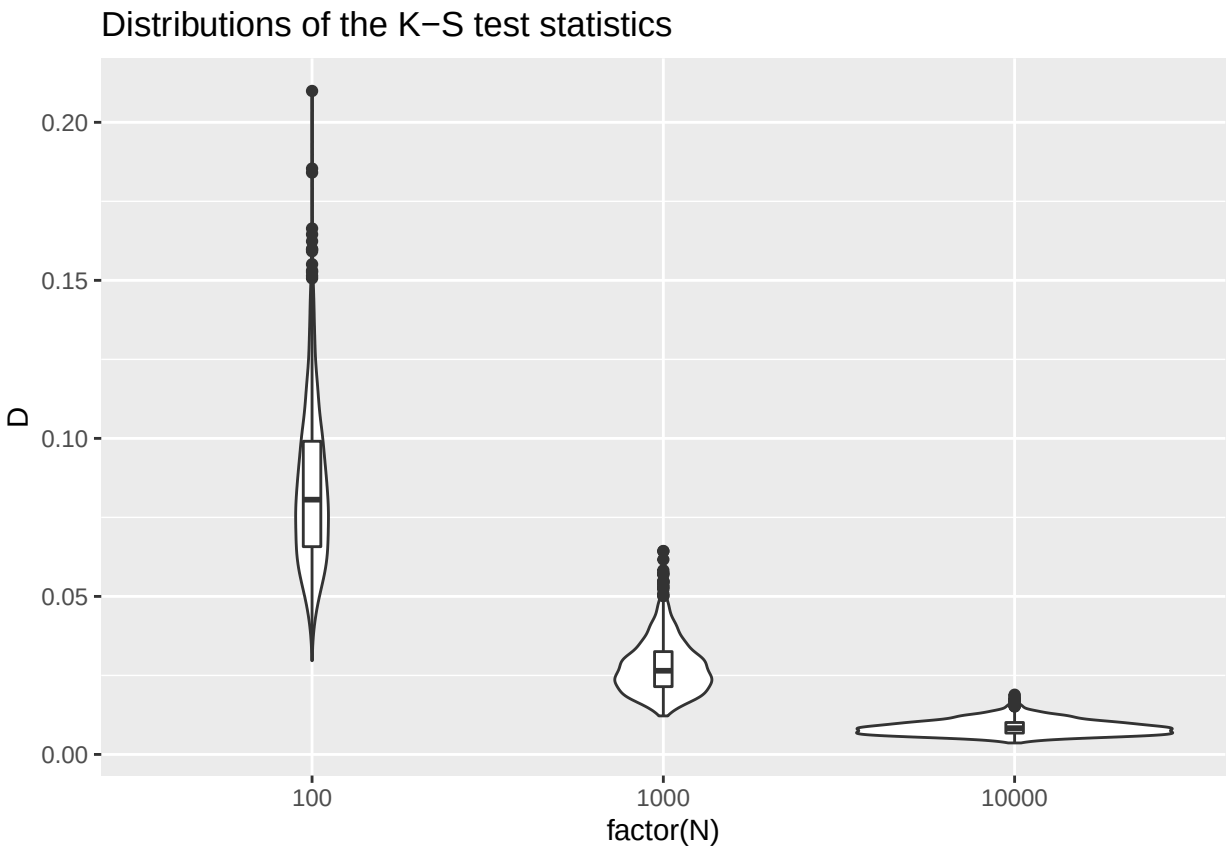
$$D = \sup_x |F_{X_n}(x) - F_X(x)|\,.$$

We then compare D to the critical values of the Kolmogorov distribution to calculate p-values of the test.

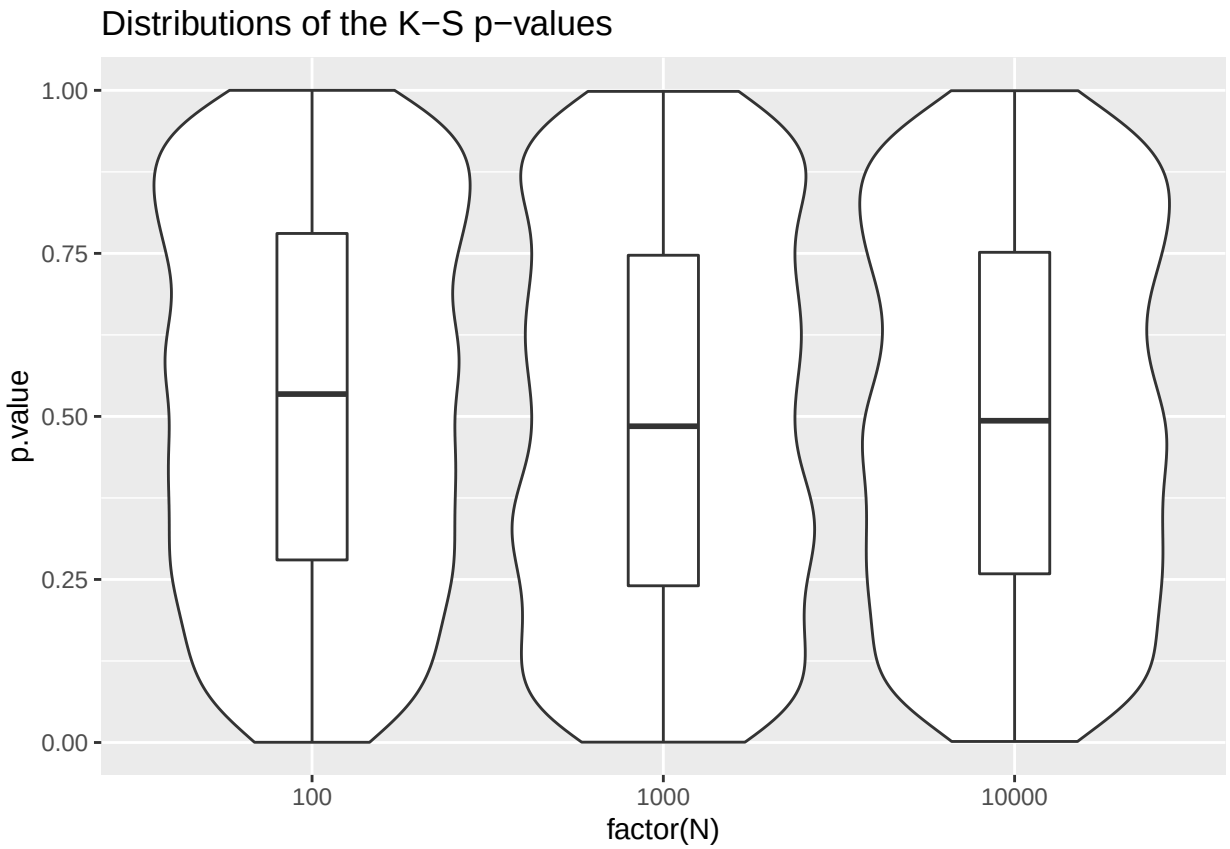
```
n_vals = c(100,1000, 10000)
nn_vals = length(n_vals)
m=1000
```

```
ks.testrhn <- function(x){
  kstestx = ks.test(x,appCDf)

  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 rhn, then split into matrix in order to use the apply function
  x1 = gen_1(n1*m)$x
  for(i in 1:m)  x[,i] <- x1[((i-1)*n1+1):(i*n1)]
  ks.testx= apply(x,2,ks.testrhn)
  ks.results = rbind(ks.results, data.frame(p.value=ks.testx[1,], D = ks.testx[2,], N=rep(n1,m)))
}
```

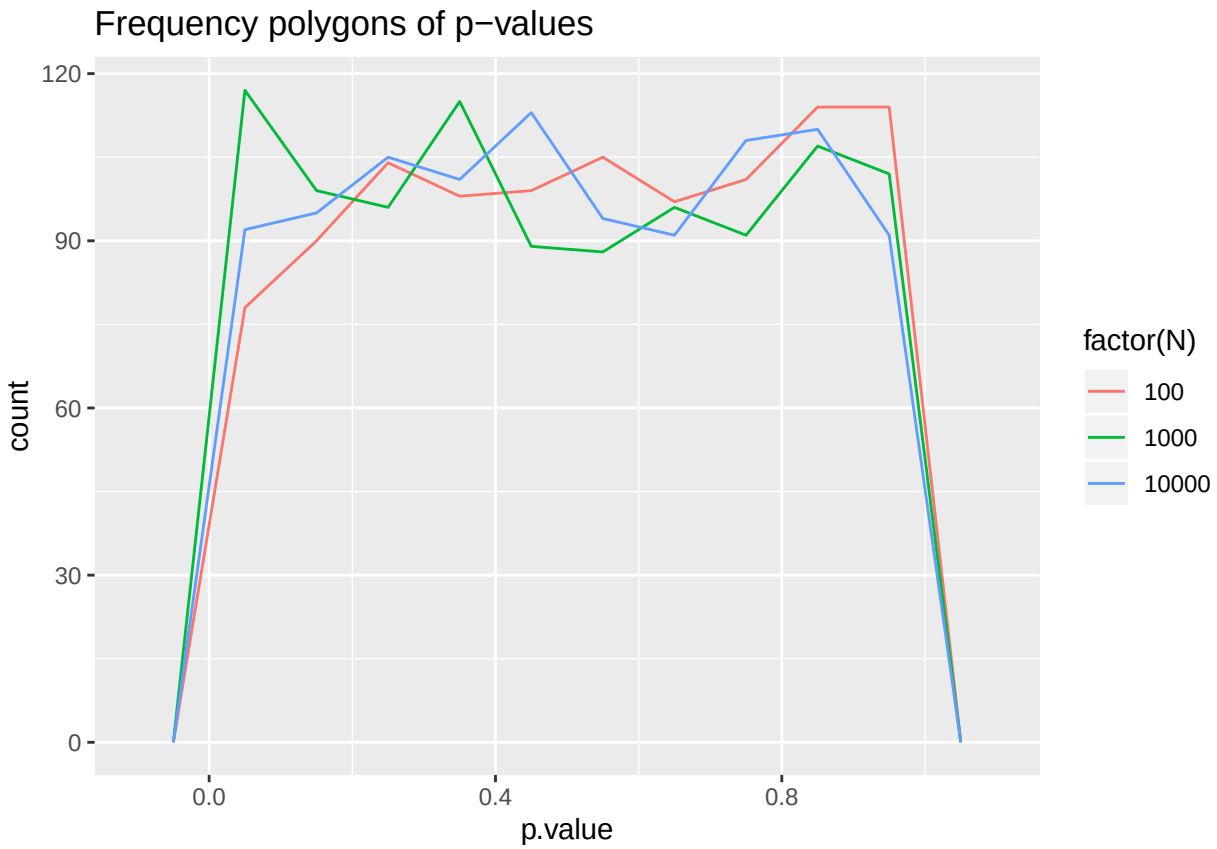
```
ggplot(ks.results, aes(factor(N), D))+
  geom_violin()+
  geom_boxplot(width=0.05)+
  labs(title="Distributions of the K-S test statistics")
```




```
ggplot(ks.results, aes(factor(N), p.value))+
  geom_violin()+
  geom_boxplot(width=0.2)+
  labs(title="Distributions of the K-S p-values")
```




```
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")
```




```
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)), 3), "Mean D"=round(mean(D), digits=3),
"Std Dev (D)"= round(sqrt(var(D)), 3))
kable(ks.table)
```

N	Mean p-value	Std Dev (p)	Mean D	Std Dev (D)
100	0.526	0.286	0.084	0.025
1000	0.493	0.295	0.028	0.009
10000	0.503	0.284	0.009	0.002

```
ks.test.critical.value(100, 0.95)
```

```
## [1] 0.13581
```

```
ks.test.critical.value(1000, 0.95)
```

```
## [1] 0.04294688
```

```
ks.test.critical.value(10000, 0.95)
```

```
## [1] 0.013581
```

We observe from the distribution plots of p-values and the D statistic shown alongside the table showing the overall statistics and the rejection values for D at $\alpha = 0.95$ at according size samples, that there is no evidence to reject the data as coming from the F_X distribution. As we would expect, the larger the sample size the closer $F_{X_n}(x)$ is to $F_X(x)$ and so the smaller is D. Furthermore, our mean p value is around 0.5 which shows that the p values are roughly uniformly distributed which we would expect under our H_0. ###

Anderson-Darling Test

The Anderson-Darling (A-D) test quantifies the difference between the empirical cumulative distribution function (ecdf), $F_{X_n}(x)$ and the theoretical cumulative distribution function, $F_X(x)$ but with more emphasis on the tails. For our density above, we have,

$$F_X(x) = \int_0^x f(x)dx = \int_0^x \frac{1}{x(2-x)\int_0^2 h(x)dx} \mathrm{e}^{-0.2(-0.6+\log(\frac{x}{2-x}))^2}, \quad 0 < x < 2,$$

and the ecdf is defined:

$$F_{X_n}(t) = \frac{\text{number of elements in the sample} \leq t}{n} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{X_i \leq t},$$

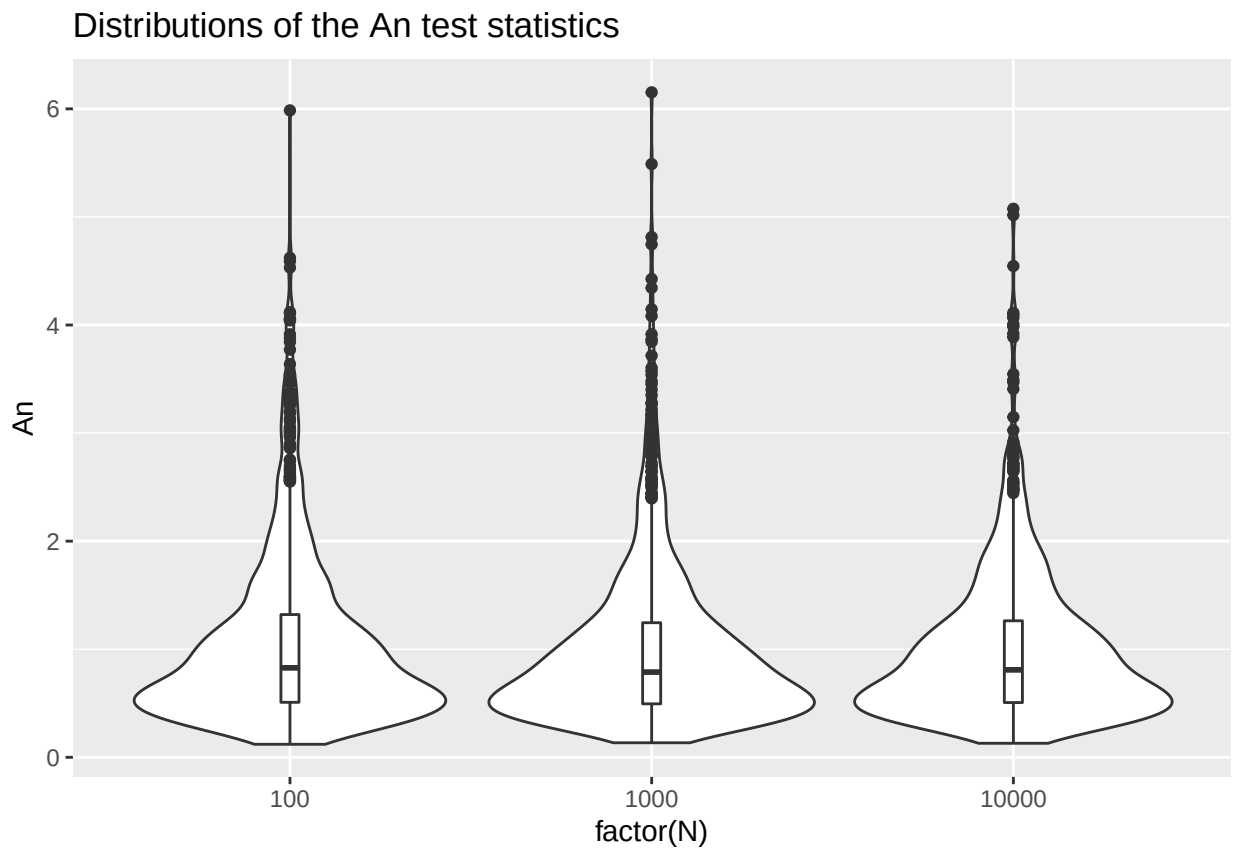
The A-D statistic is given by

$$A^2 = n\int_{-\infty}^{\infty} \frac{(F_n(x) - F(x))^2}{F(x)\,(1-F(x))}\, dF(x),$$

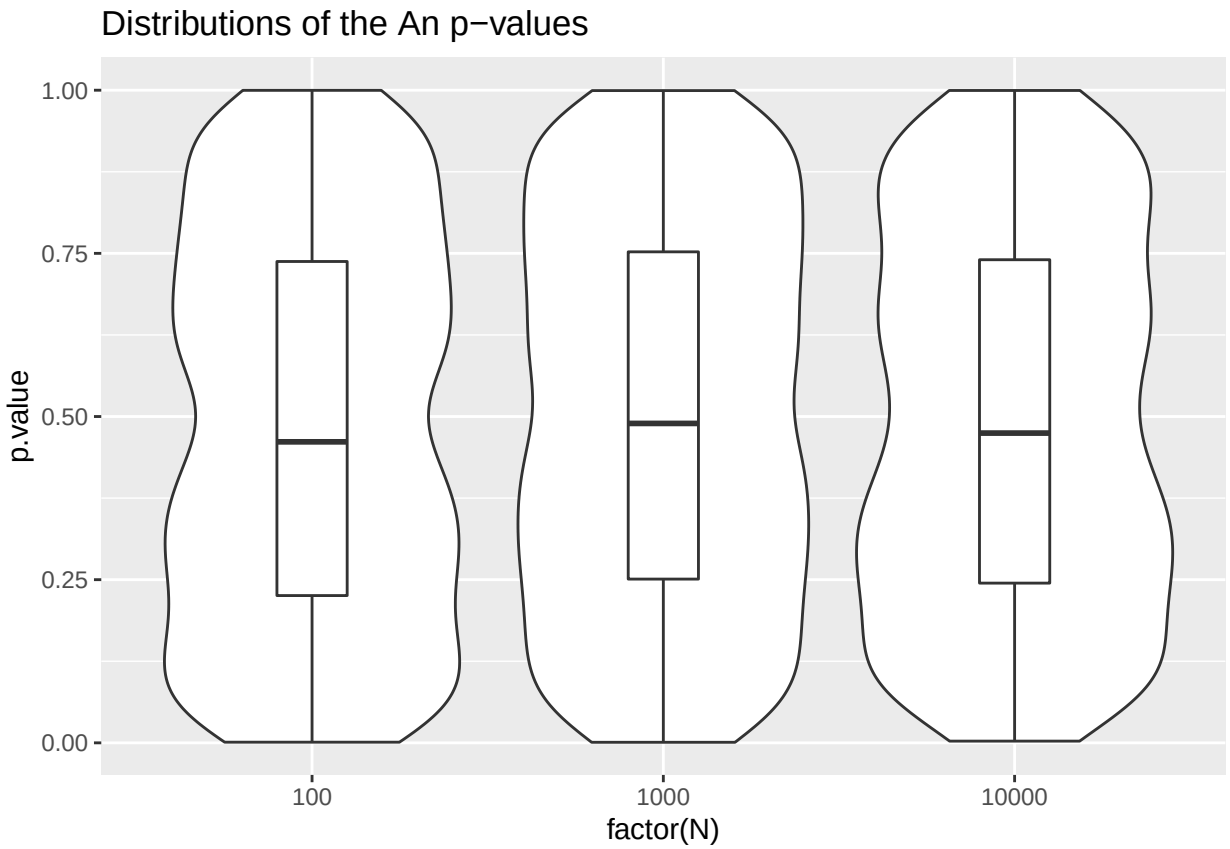
We then compare A to the critical values of the A-D distribution to calculate p-values of the test.

```
n_vals = c(100,1000, 10000)
nn_vals = length(n_vals)
m=1000

ad.testrhn <- function(x){
  adtestx = ad.test(x,appCDf)

  return(c(adtestx$p.value, adtestx$statistic))
}

ad.results=data.frame()
for(n in 1:nn_vals){
  n1=n_vals[n]
  x <- matrix(0, nrow=n1, ncol=m)
  # one call to rhn, then split into matrix in order to use the apply function
  x1 = gen_1(n1*m)$x
  for(i in 1:m)  x[,i] <- x1[((i-1)*n1+1):(i*n1)]
  ad.testx= apply(x,2,ad.testrhn)
  ad.results = rbind(ad.results, data.frame(p.value=ad.testx[1,], An = ad.testx[2,], N=rep(n1,m)))
}
```

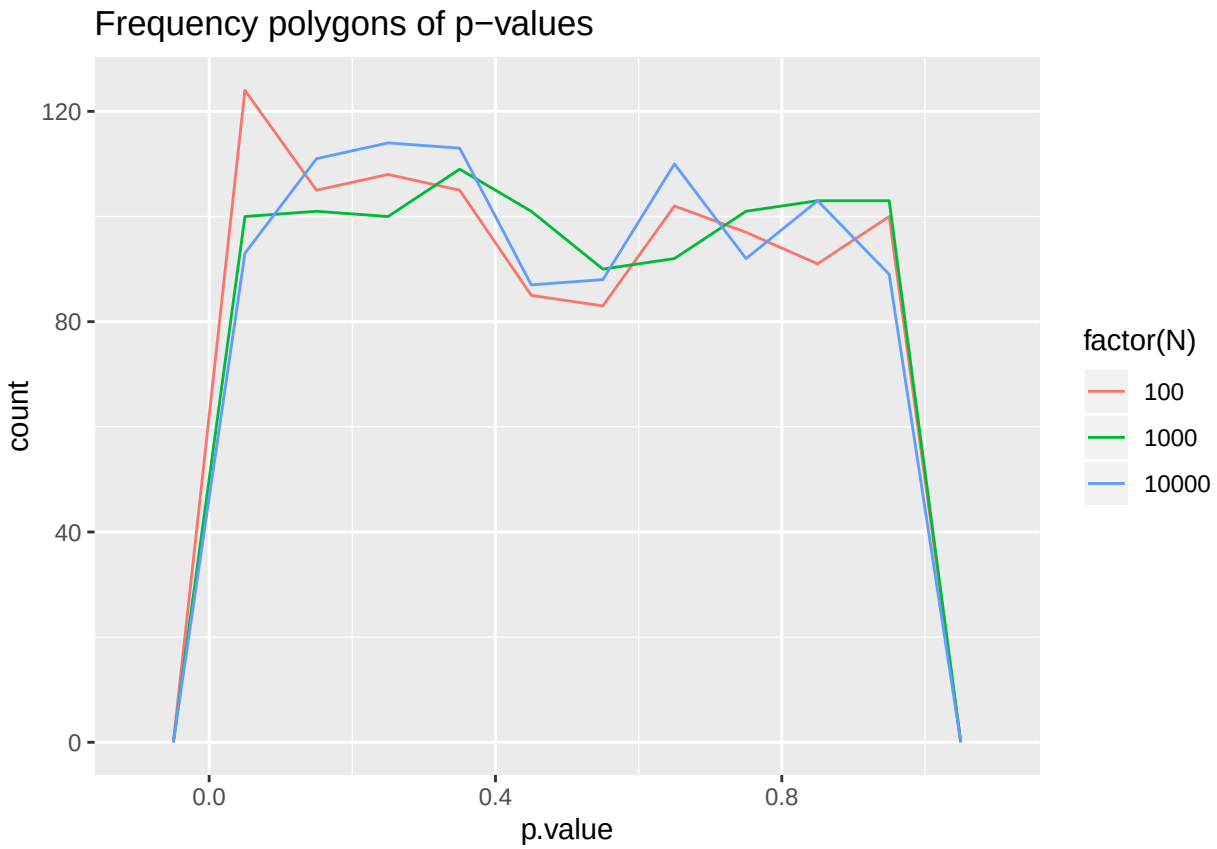
```
ggplot(ad.results, aes(factor(N), An))+
  geom_violin()+
  geom_boxplot(width=0.05)+
  labs(title="Distributions of the An test statistics")
```




```
ggplot(ad.results, aes(factor(N), p.value))+
  geom_violin()+
  geom_boxplot(width=0.2)+
  labs(title="Distributions of the An p-values")
```




```
ggplot(ad.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")
```




```
ad.table <- ad.results %>% group_by(N) %>% summarise("Mean p-value" = round(mean(p.value),digits=3),
"Std Dev (p)" = round(sqrt(var(p.value)), 3), "Mean An"=round(mean(An), digits=3),
"Std Dev (An)"= round(sqrt(var(An)), 3))
kable(ad.table)
```

N	Mean p-value	Std Dev (p)	Mean An	Std Dev (An)
100	0.482	0.296	1.059	0.800
1000	0.499	0.290	1.003	0.755
10000	0.490	0.287	0.994	0.690

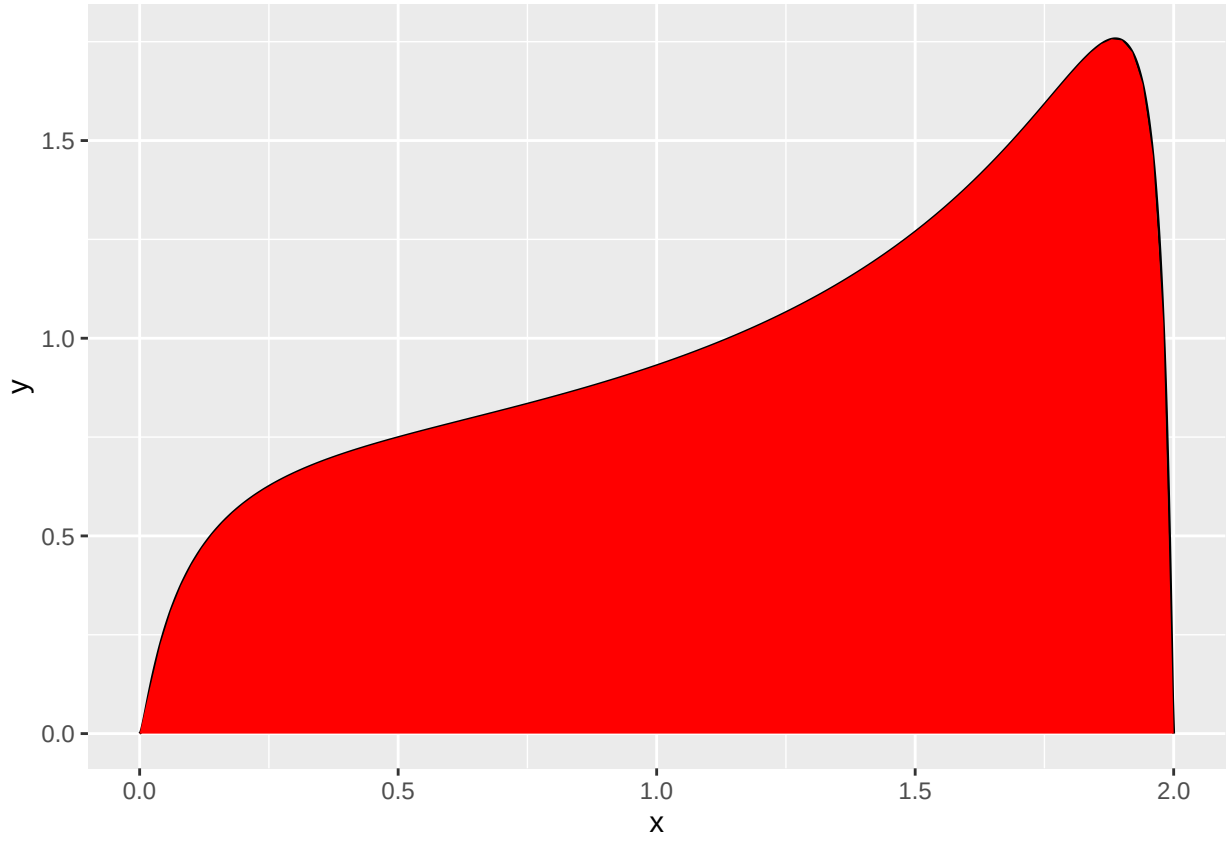
We observe from the distribution plots of p-values and the An statistic shown alongside the table showing the overall statistics at according size samples, that there is no evidence to reject the data as coming from the F_X distribution. As we would expect, the larger the sample size the closer $F_{X_n}(x)$ is to $F_X(x)$ and so the smaller is An. Furthermore, our mean p value is around 0.5 which shows that the p values are roughly uniformly distributed which we would expect under our H_0.

Monte Carlo

$\theta = \int_0^2 h(x)dx = \int_0^2 \frac{1}{x(2-x)} \mathrm{e}^{-0.2(-0.6+\log(\frac{x}{2-x}))^2} dx$ We visualize the value of θ.

```
require("ggplot2")
require("dplyr")
theta <- 0.5 - atan(2)/pi
h1<-function(x){
```

```
  if (x<=0 | x>=2){0}

  else{1/(x*(2-x))*exp(-0.2*(-0.6+log(x/(2-x)))^2)}
}
h <- Vectorize(h1, vectorize.args = "x")
x=seq(0,2,l=1000)
my_pdf <- data.frame(x=x, y=h(x))
ggplot(my_pdf)+
  geom_line(aes(x,y))+
  stat_function(fun = h, xlim = c(0,2), geom = "area", fill="red")
```



Method 1)

$$f_X(x) = \frac{1}{2}, x \in (0,2), \phi(x) = 2h(x)$$

```
n=100
y = runif(n, 0, 2)
t1 <- sum(2*h(y))/n
```

Method 2) (Hit or Miss) We find $c = sup_{x\in(0,2)}h(x) = 1.757373$.

```
MCc<-optimise(h, c(0,2), maximum = TRUE)[[2]]
MCc
```

```
## [1] 1.757373
```

```
U = runif(n, 0, 2)
V = runif(n, 0, MCc)
```

```r
t2 <- MCc * 2 * sum(V <= h(U)) / n
```

```r
cat("Method 1 (Crude) Theta=",t1, "\n")
```

```
## Method 1 (Crude) Theta= 1.998209
```

```r
cat("Method 2 (Hit or Miss) Theta=",t2, "\n")
```

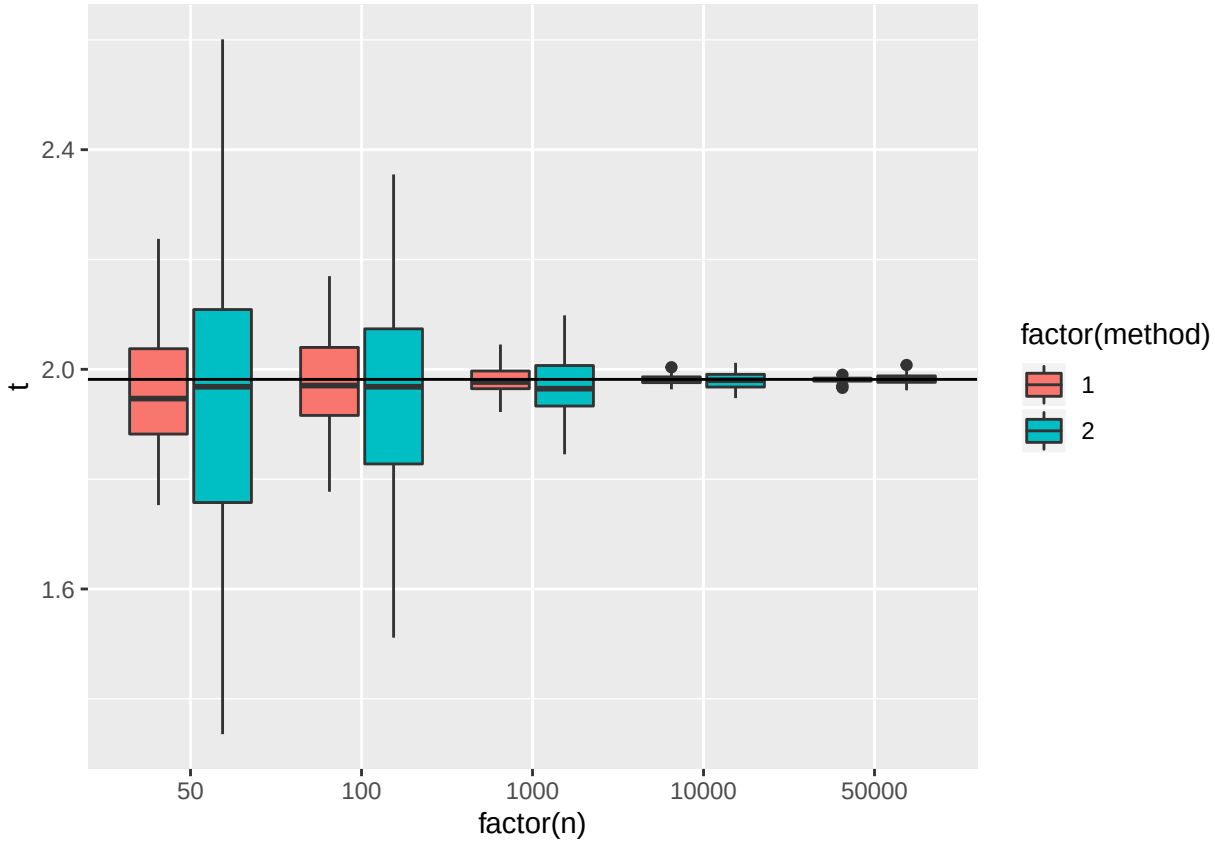
```
## Method 2 (Hit or Miss) Theta= 1.862816
```

```r
cat("True value = ",hint)
```

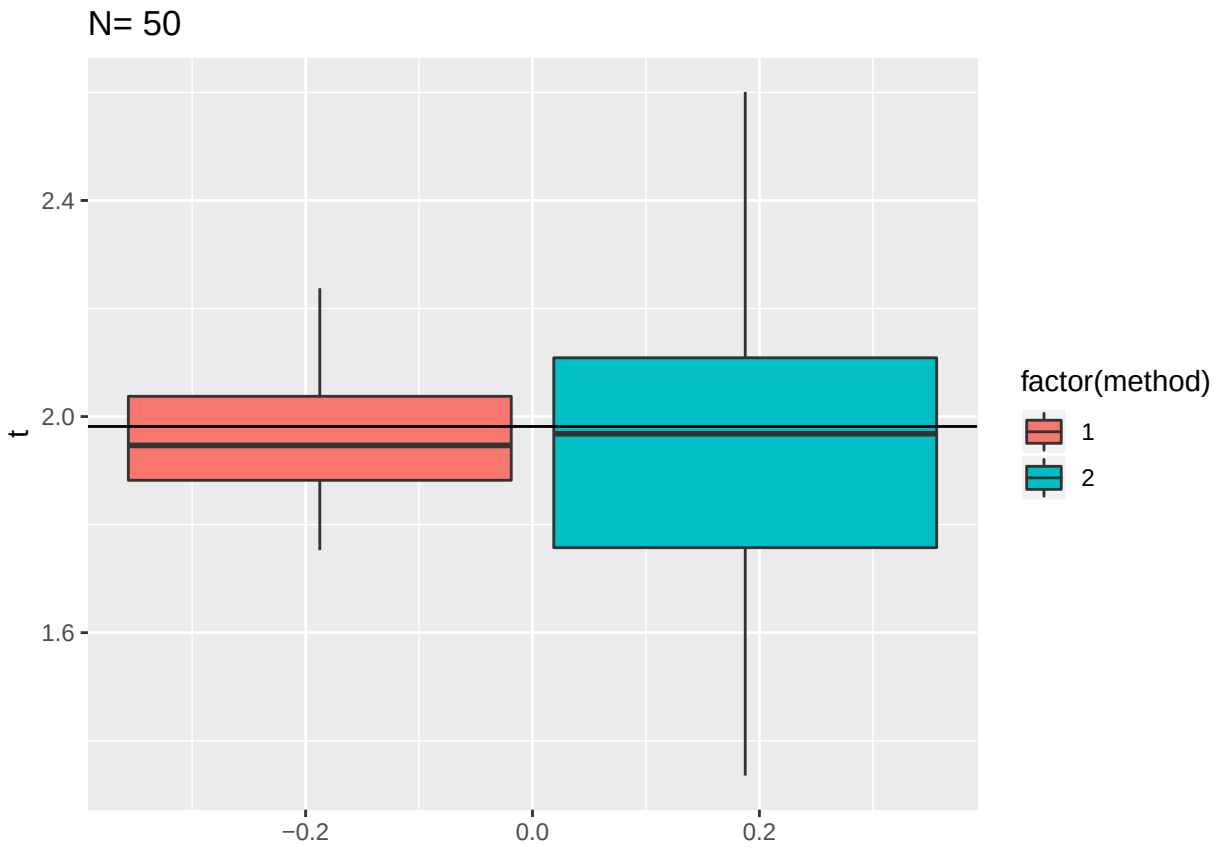
```
## True value =  1.981664
```

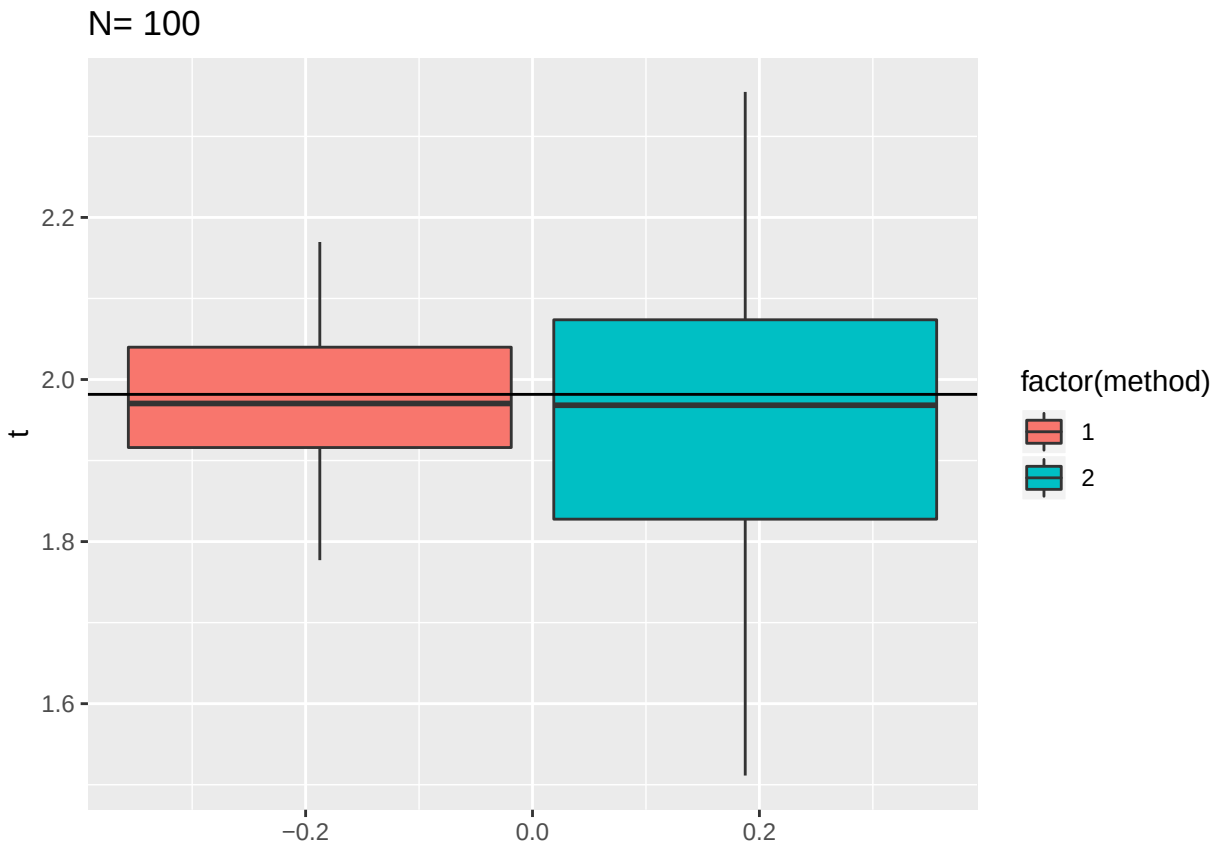
```r
# simulation study
nvals=c(50,100,1000, 10000, 50000)
results = data.frame()
nnvals=length(nvals)
m=100
for(n in 1:nnvals){
  n1=nvals[n]
  for(i in 1:m){
    y=runif(n1, 0, 2)
    t1 <- sum(2*h(y))/n1
    U = runif(n1, 0, 2)
    V = runif(n1, 0, MCc)
    t2 <- MCc * 2 * sum(V <= h(U)) / n1
    results=rbind(results, data.frame(n=n1, t=t1,method=1))
    results=rbind(results, data.frame(n=n1, t=t2,method=2))

  }
}
```

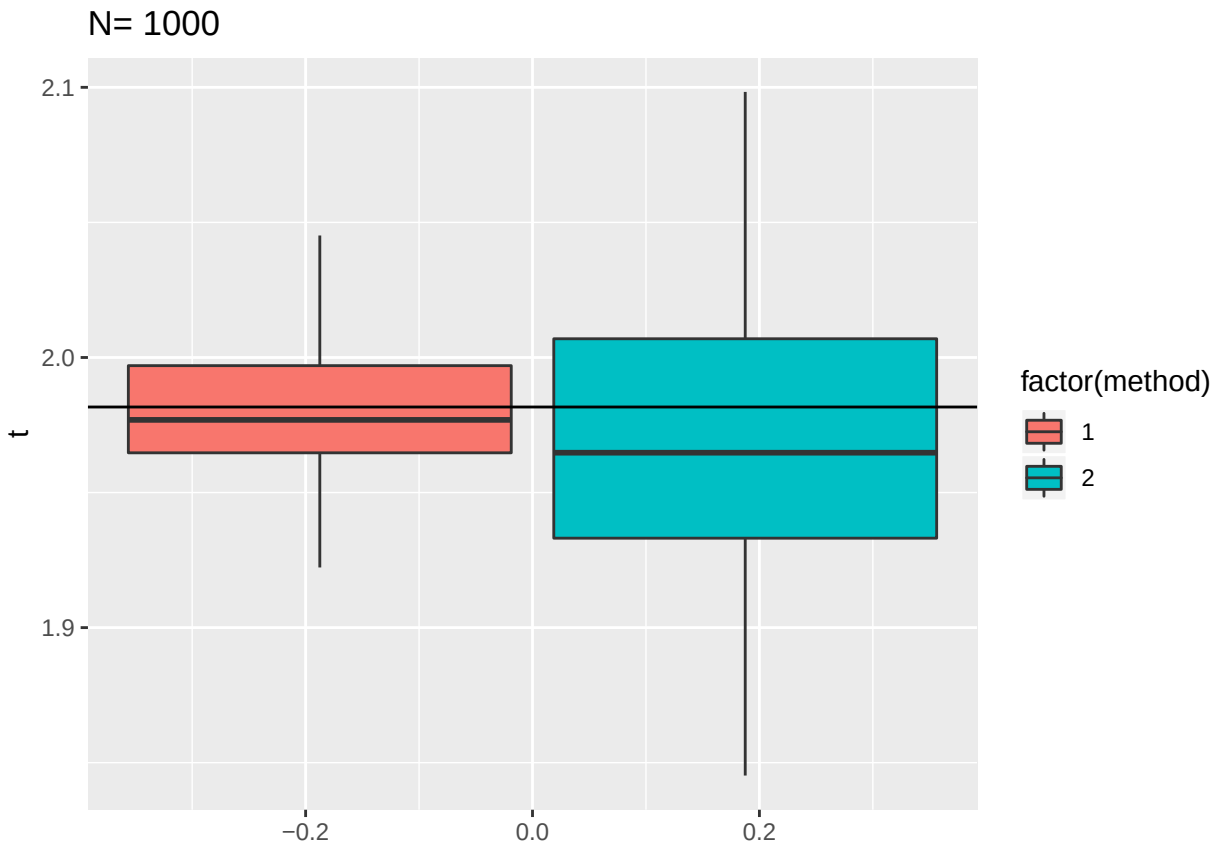
```r
ggplot(results,aes(x=factor(n),y=t,fill=factor(method)))+
  geom_boxplot()+
  geom_hline(yintercept=hint)
```

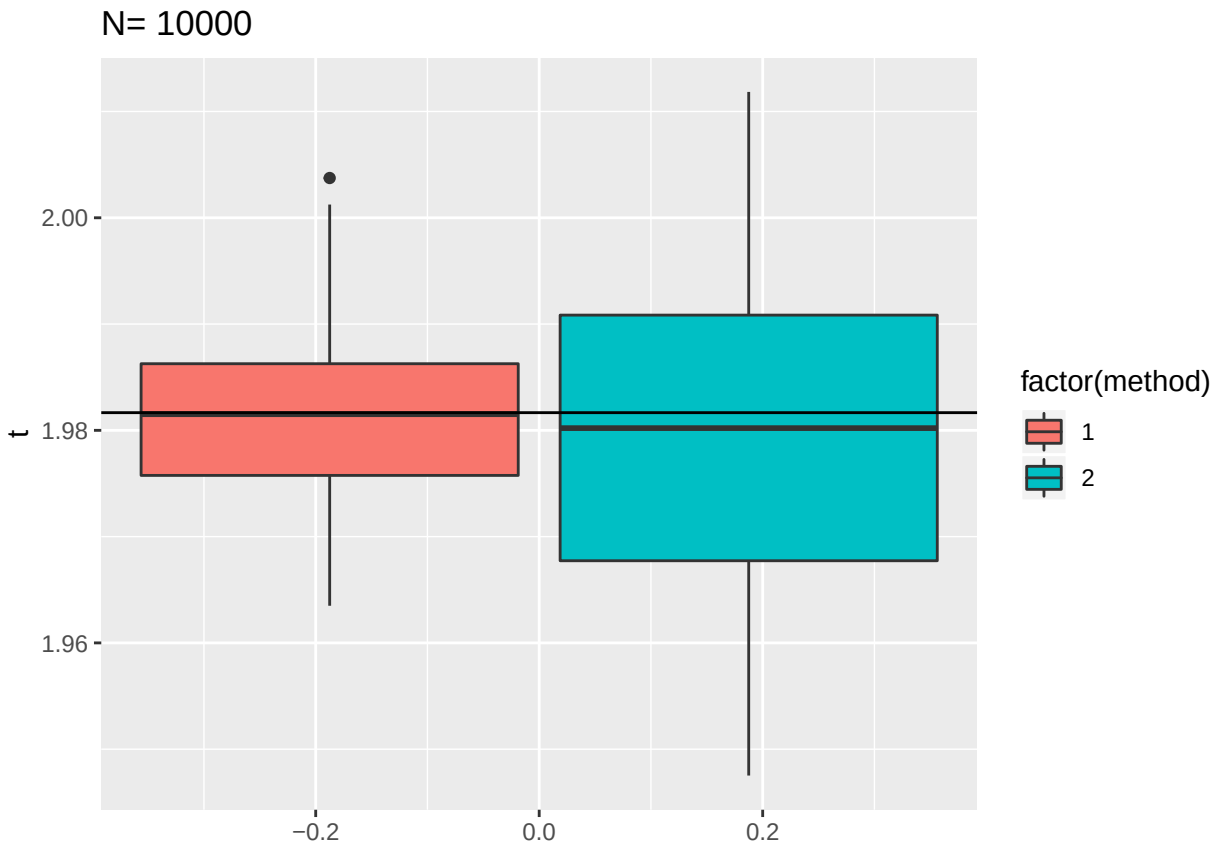


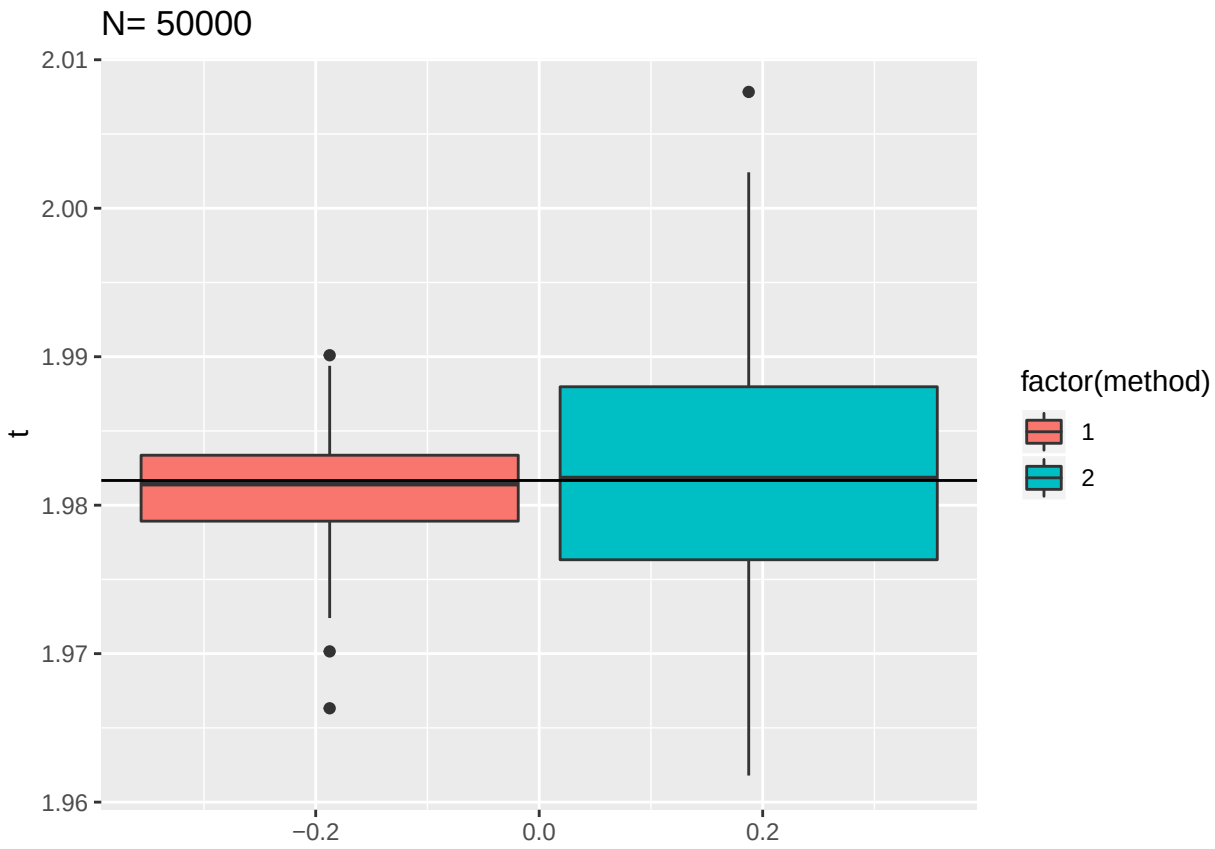

```
for(i in 1:length(nvals)){
  p1 <- filter(results, n==nvals[i])
  plot <- ggplot(p1,aes(y=t,fill=factor(method)))+
    geom_boxplot()+
    geom_hline(yintercept=hint)+
    labs(title=paste("N=",nvals[i]))
  print(plot)
  }
```


N= 50
t
2.4
2.0
1.6
−0.2
0.0
0.2
factor(method)
1
2


N= 100
t
2.2
2.0
1.8
1.6
−0.2
0.0
0.2
factor(method)
1
2

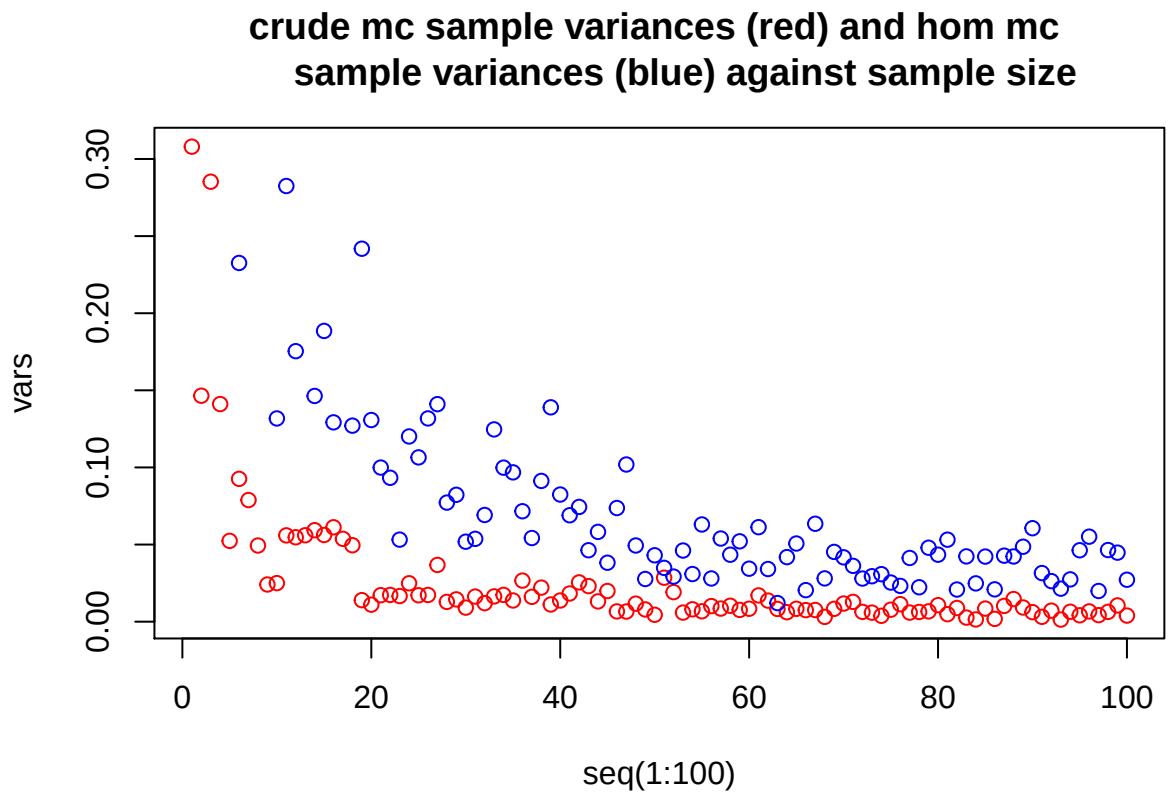

N= 1000
2.1
2.0
1.9
t
−0.2
0.0
0.2
factor(method)
1
2


N= 10000
2.00
1.98
1.96
t
−0.2
0.0
0.2
factor(method)
1
2

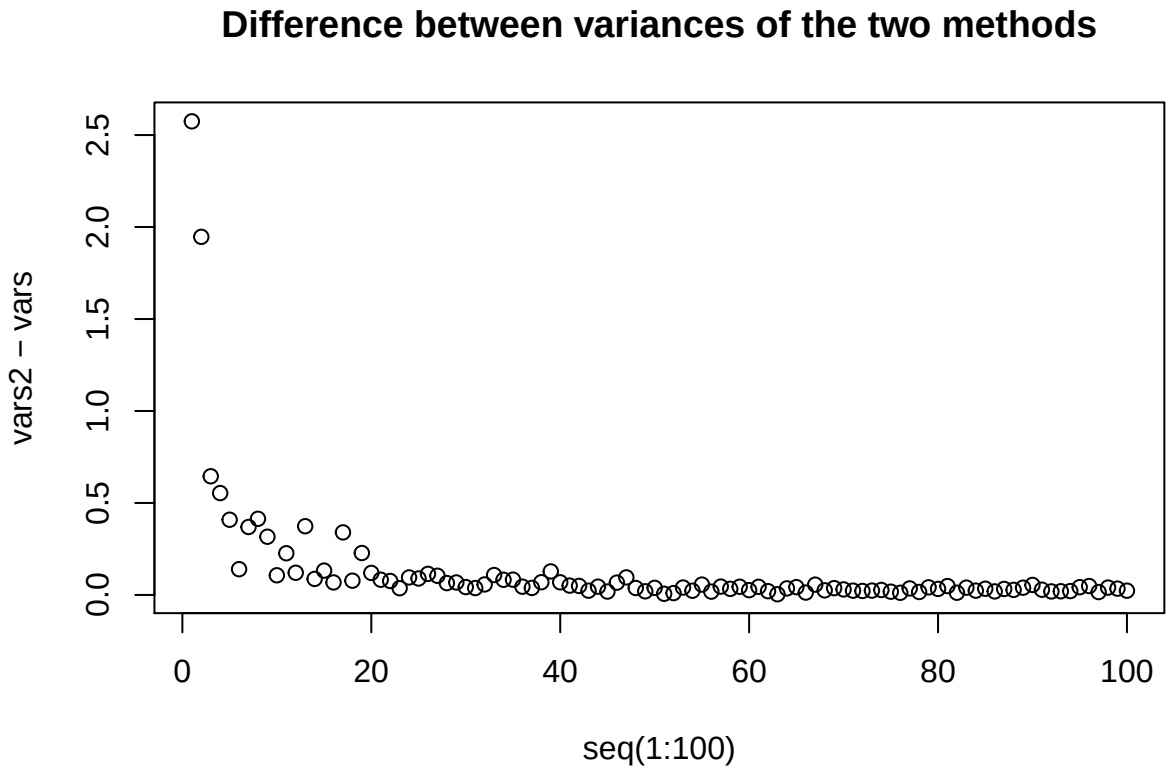



```
vars <- c()
for (i in seq(1:100)){
  val <- c()
  for (j in 1:10){
    y = runif(i, 0, 2)
    t1 <- sum(2*h(y))/i
    val<-c(val, t1)
  }
  vars <- c(vars, var(val))

}
plot(seq(1:100), vars, col = "red", main = "crude mc sample variances (red) and hom mc
     sample variances (blue) against sample size")


vars2 <- c()
for (i in seq(1:100)){

  val2 <- c()
  for (j in 1:10){

    U = runif(i, 0, 2)
    V = runif(i, 0, MCc)
    t2 <- MCc * 2 * sum(V <= h(U)) / i
    val2<-c(val2, t2)
  }
```

```
  vars2 <- c(vars2, var(val2))
}
points(seq(1:100), vars2, col = "blue")
```




```
plot(seq(1:100), vars2 - vars, main = "Difference between variances of the two methods")
```



As we expected we see that the variances associated with crude Monte Carlo are significantly lower than Hit or Miss for approximating the integrating constant.