Final_Assignment

Manuel Bottino

2023-05-15

Contents

1 Risiko

1.0.1 Rules:

Goal of the game -> You (the attacker) want to conquer your opponent's territory (the defender), and you do so by using your army, made of units.You decide how many units to use in the attack, while the defender uses their available units in the territory.

Battle algorithm 1. both players toss some dice (with six equiprobable faces). In particular, the attacker will throw three dice if the number of units at their disposal exceeds three. The same holds for the defender (we consider the European version, where the defender can toss up to three dice).

2. they sort their outcomes and compare the maximum values they scored. If the attacker's highest value is strictly greater than the defender's, the attacker LOSES a unit. Otherwise, the defender loses one.
3. they remove these dice and repeat the operation (comparing the second-highest values and so on). The first player that loses all their units is the loser of the battle.

1.1 Point a)

Given two i.i.d variables X_1, X_2, Y with sample space $\Omega = \{1, 2, 3, 4, 5, 6\}$. We define $M = \max(X_1, X_2)$, our aim is to obtain $P(M > y)$. In other words, the probability that an attacker with two units of defeating a defender with one unit. First of all defint the probability density function of M

$$\mathbb{P}(M = m) = 2\frac{(m-1)+1}{36}$$

It is easy to see just by looking at some instances. The probability of 3 being the maximum is: -

$$\mathbb{P}(m = 3) = \frac{1}{6}\frac{2}{6} + \frac{2}{6}\frac{1}{6} + \frac{1}{6}\frac{1}{6}$$

In words, the probability of getting 3 with the first dice and any other number lower than three (1,2) for the second ($\frac{1}{6}\frac{2}{6}$) and viceversa ($\frac{2}{6}\frac{1}{6}$), then we consider the probability of both being equal ($\frac{1}{6}\frac{1}{6}$)

Finally, we consider the probability of interest: -

$$\mathbb{P}(M > y) = 1 - \mathbb{P}(M \leq y)$$

Thus we should consider each possible combination of y and m and compute it:

$$\mathbb{P}(M \leq y) = 6\frac{1}{36}\frac{1}{6} + 5\frac{3}{36}\frac{1}{6} + 4\frac{5}{36}\frac{1}{6} + 3\frac{7}{36}\frac{1}{6} + 2\frac{9}{36}\frac{1}{6} + \frac{11}{36}\frac{1}{6}$$

Which I have computed using R:

```r
X_i=sapply(1:6, function(i) (2*(i-1)+1)/36) # P(Z=z)

a=(6:1)
X_lessthanorequalto_Y=sapply(1:6, function(i) a[i]*X_i[i]*1/6)

1-sum(X_lessthanorequalto_Y)
```

```
## [1] 0.5787037
```

1.2 Point b)

```r
War=function(att_units, def_units){
  while((att_units>0 & def_units>0)){
```

```
  dices_att=min(att_units,3)
  dices_def=min(def_units,2)
  dices=min(dices_att, dices_def)

  att_rand=sort(sample(1:6, dices_att), decreasing=T)[1:dices]
  def_rand=sort(sample(1:6, dices_def), decreasing=T)[1:dices]

  att_units=att_units-sum(def_rand>=att_rand)
  def_units= def_units- sum(att_rand>def_rand)

  }
  if(def_units<=0){
    return(1)
  }else{
    return(0)
  }
}


Risiko=function(att_units, def_units, sim=1){

  count=sapply(1:sim, function(i) War(att_units=att_units, def_units=def_units))

  return(sum(count)/sim)
}
```

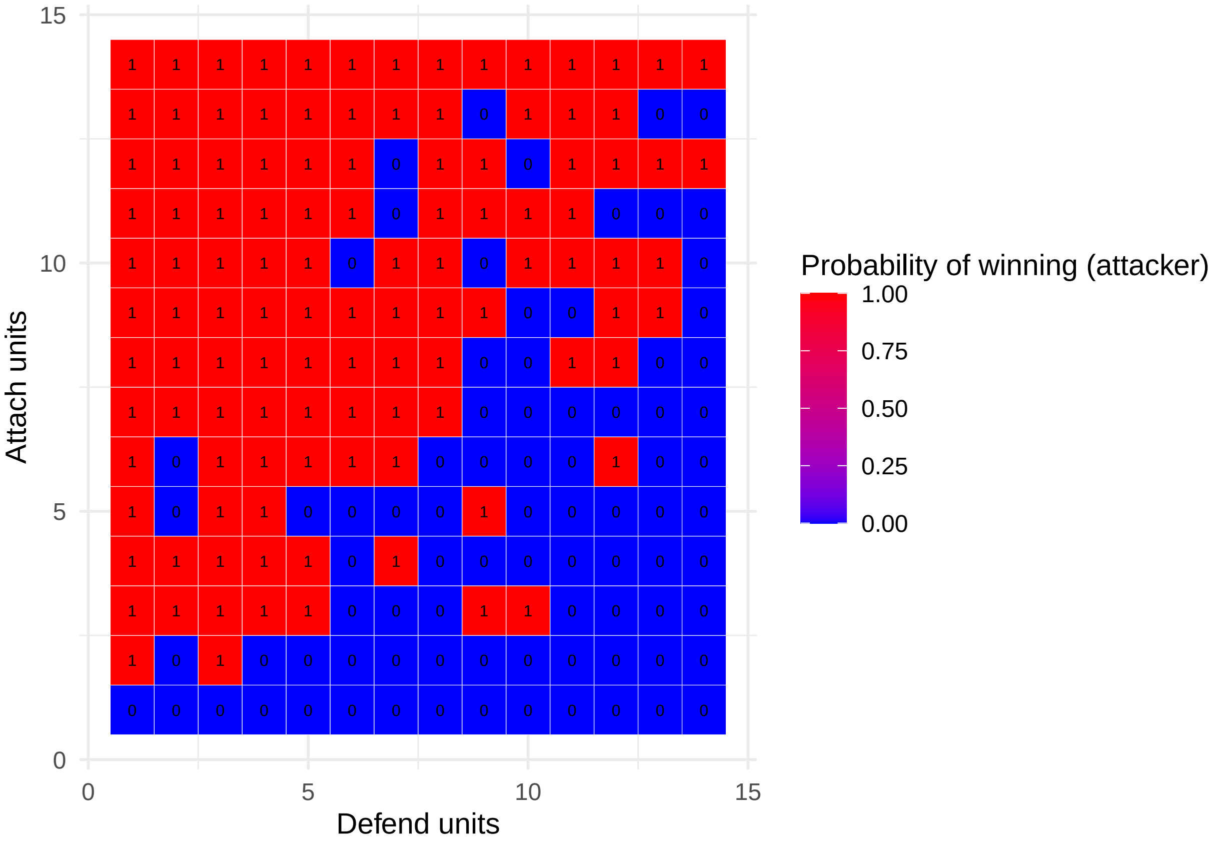
1.3 Point c)

```
set.seed(123)

a=c(1:14)
outcomes=outer(a, a, Vectorize(Risiko)) # apply the function Risiko to all possible combination of atta
```

1.4 Point d)

Before plotting we ought to point out the fact our table will surely be different from the one computed on this paper at table 3. The reason behind this is in the assumptions we are making about the game, the defender can use up to 3 dice in our version, not 2. It is easy to change this assumption in the function and the results will almost coincide.

```
heatmap <- ggplot(data = melt(outcomes)) + # melt() function from reshape2 library to go from a matrix
  geom_tile(aes(x = Var2, y = Var1, fill = value), color = "white") +
  scale_fill_gradient(low = "blue", high = "red") +
  geom_text(aes(x = Var2 , y = Var1, label = round(value, 2)), color = "black", size=2) +
  labs(x = "Defend units", y = "Attach units", fill = "Probability of winning (attacker)") +
  theme_minimal() # to make the plot look better

heatmap
```



The probability obtained using simulations is higher from the one computed analically in point a. The intuition behind could be the fact that in the first scenario we are computing only one battle (the maximum of

$$X_1, X_2$$

against Y). Instead, simulation also includes the possibility for the attacker of losing one unit in the first battle, but then winning the second battle, thus winning what the "war". In other words, simulation includes another chance for the attacker to roll the dice, so it follows an higher probability of winning.

2 Montecarlo Simulations I

2.1 Point a)

An old and naive algorithm for the generation of Normally distributed random numbers is the following:

$$U_1, \dots, U_{12} \sim U\left(-\frac{1}{2}, \frac{1}{2}\right); \quad Z = \sum_{i=1}^{12} U_i$$

The algorithm generates twelve independent uniform variables between $(-1/2, 1/2)$ and then sets Z as the sum of them. The rationale here is that 12 realizations are usually enough to exploit the CLT. Let's firstly consider the expected value of Z: since our variables are independent and uniformly distributed between $[-1/2, 1/2]$ the expected value of the sum is the sum of the expected value by linearity of the operator.

$$\mathbb{E}(Z) = E(U_1 + U_2 + \cdots + U_{12}) = \mathbb{E}(U_1) + E(U_2) + \cdots + \mathbb{E}(U_{12}) = 0 + 0 + \cdots + 0 = 0$$

Because the expected value of a Uniform random variable is $E(U_i) = \frac{a+b}{2}$ so it is zero in the case of a Uniform centered in the origin. More formally we can write:

$$\mathbb{E}\left(\sum_{i=1}^{12} U_i\right) = \sum_{i=1}^{12} \mathbb{E}(U_i) = \sum_{i=1}^{12} 0 = 0$$

Looking at the variance of Z:

$$\mathbb{V}(Z) = \mathbb{V}(U_1 + U_2 + \cdots + U_{12}) = \mathbb{V}(U_1) + \mathbb{V}(U_2) + \cdots + \mathbb{V}(U_{12}) \qquad + 2Cov(U_1, U_2) + 2Cov(U_1, U_3) + \cdots + 2Cov(U_{11}, U_{12})$$

Where $Cov(U_i, U_j)$ denotes the covariance between U_i and U_j, $i \neq j$.

Since $U_1, U_2, \ldots, U_{12}$ are again independent we have:

$$Cov(U_i, U_j) = 0 \quad \forall i \neq j$$

Therefore,

$$\mathbb{V}(Z) = \mathbb{V}(U_1) + \mathbb{V}(U_2) + \cdots + \mathbb{V}(U\_12) = 12 \cdot \mathbb{V}(U_1)$$

Because the variance of the Uniformm random variable is

$$V(U) = \frac{(b-a)^2}{12}$$

, we have::

$$\mathbb{V}(U_1) = \left(\frac{\frac{1}{2} - (-\frac{1}{2})}{\sqrt{12}}\right)^2 = \frac{1}{12}$$

Therefore,

$$\mathbb{V}(Z) = 12 \cdot \frac{1}{12} = 1$$

So we proved that $E(Z) = 0$ e \$ Var}(Z) = 1\$.

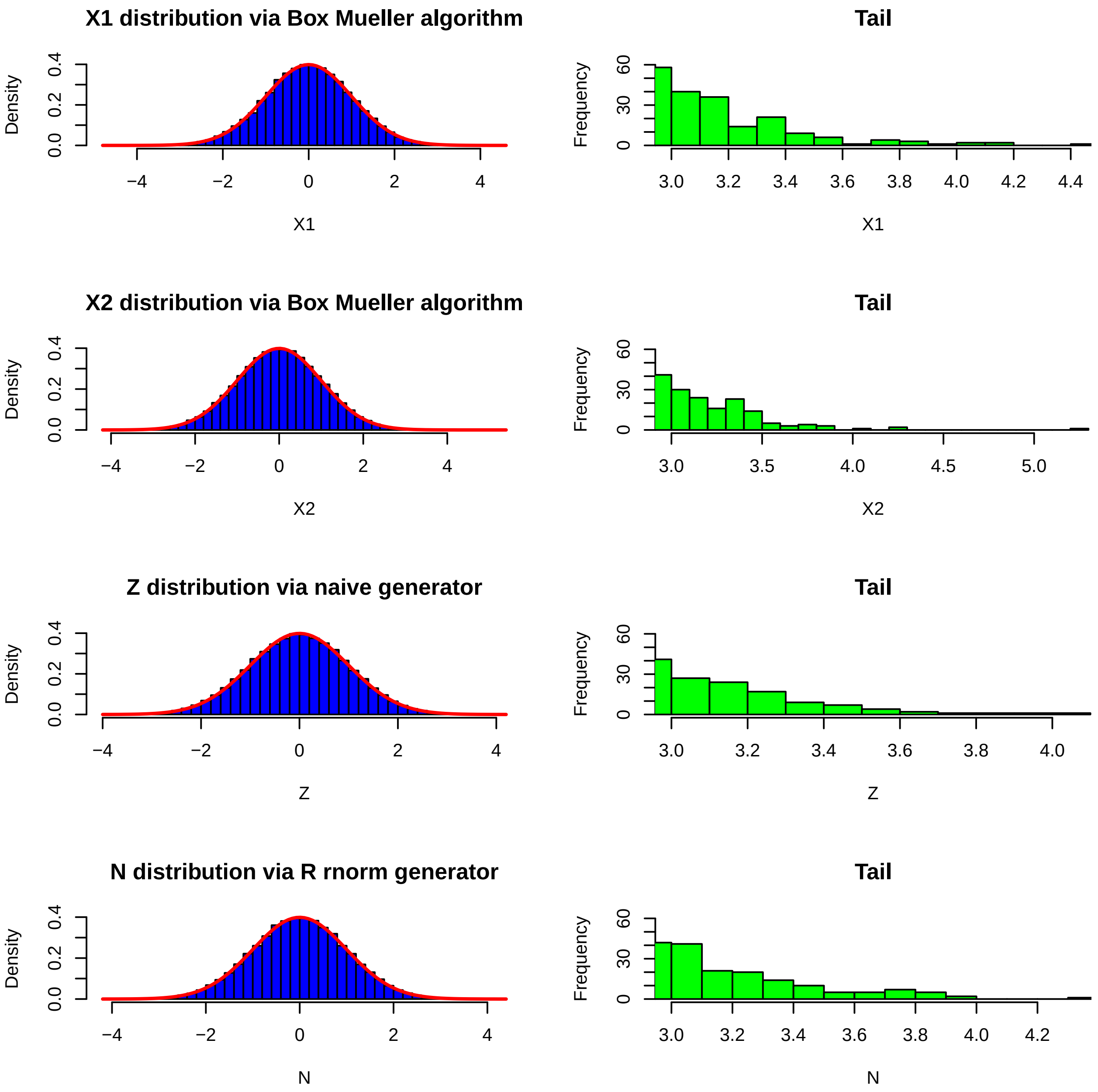
2.2 Point b) e c)

Using histograms, compare the above Normal generator with the Box–Mueller algorithm. Comment on the results, and pay particular attention to tail probabilities (e.g., what happens to the estimate of

$$\mathbb{P}(Z >= 3)$$

).

```r
# Box-Mueller
set.seed(123)
u1 = runif(100000)
u2 = runif(100000)

unif.0.2pi = 2*pi*u2
exp.1over2 = - 2 * log(1-u1)

X1 = sqrt( - 2 * log(1-u1)) * cos(2*pi*u2)
X2 = sqrt( - 2 * log(1-u2)) * sin(2*pi*u1)

# Uniform sum generator
set.seed(123)
n = 100000
U_i = matrix(runif(n*12, min=-1/2, max=1/2), nrow=n)
Z = rowSums(U_i)

# R rnorm generator
set.seed(123)
N = rnorm(100000, 0, 1)

par(mfrow = c(4, 2))
hist(X1, breaks=50, col="blue", main="X1 distribution via Box Mueller algorithm", freq = FALSE )
curve(dnorm(x, mean = 0, sd = 1), add = TRUE, col = "red", lwd = 2)
hist(X1, breaks=100, col="green", main="Tail", xlim=c(3, max(X1)), ylim=c(0, 60))
hist(X2, breaks=50, col="blue", main="X2 distribution via Box Mueller algorithm", freq = FALSE)
curve(dnorm(x, mean = 0, sd = 1), add = TRUE, col = "red", lwd = 2)
hist(X2, breaks=100, col="green", main="Tail", xlim=c(3, max(X2)), ylim=c(0, 60))
hist(Z, breaks=50, col="blue", main="Z distribution via naive generator", freq = FALSE)
curve(dnorm(x, mean = 0, sd = 1), add = TRUE, col = "red", lwd = 2)
hist(Z, breaks=100, col="green", main="Tail", xlim=c(3, max(Z)), ylim=c(0, 60))
hist(N, breaks=50, col="blue", main="N distribution via R rnorm generator", freq = FALSE)
curve(dnorm(x, mean = 0, sd = 1), add = TRUE, col = "red", lwd = 2)
hist(N, breaks=100, col="green", main="Tail", xlim=c(3, max(N)), ylim=c(0, 60))
```



```
X1_tail = mean(X1>3)
Z_tail = mean(Z>3)
N_tail = mean(N>3)
Norm = 1-pnorm(3, 0, 1)
X1_tail; Z_tail; N_tail; Norm
```

```
## [1] 0.0014
```

```
## [1] 0.00094
```

```
## [1] 0.00131
```

```
## [1] 0.001349898
```

It can be concluded that the Box Muller algorithm and our generator built by a sum of Uniform distributed random variables are both able to properly approximate a sample drawn from a Normal distribution, but for large numbers (n = 100.000) the Box Mueller Algorithm can better approximate the tails. The R built-in random sampling algorithm "rnorm" is even more precise though.

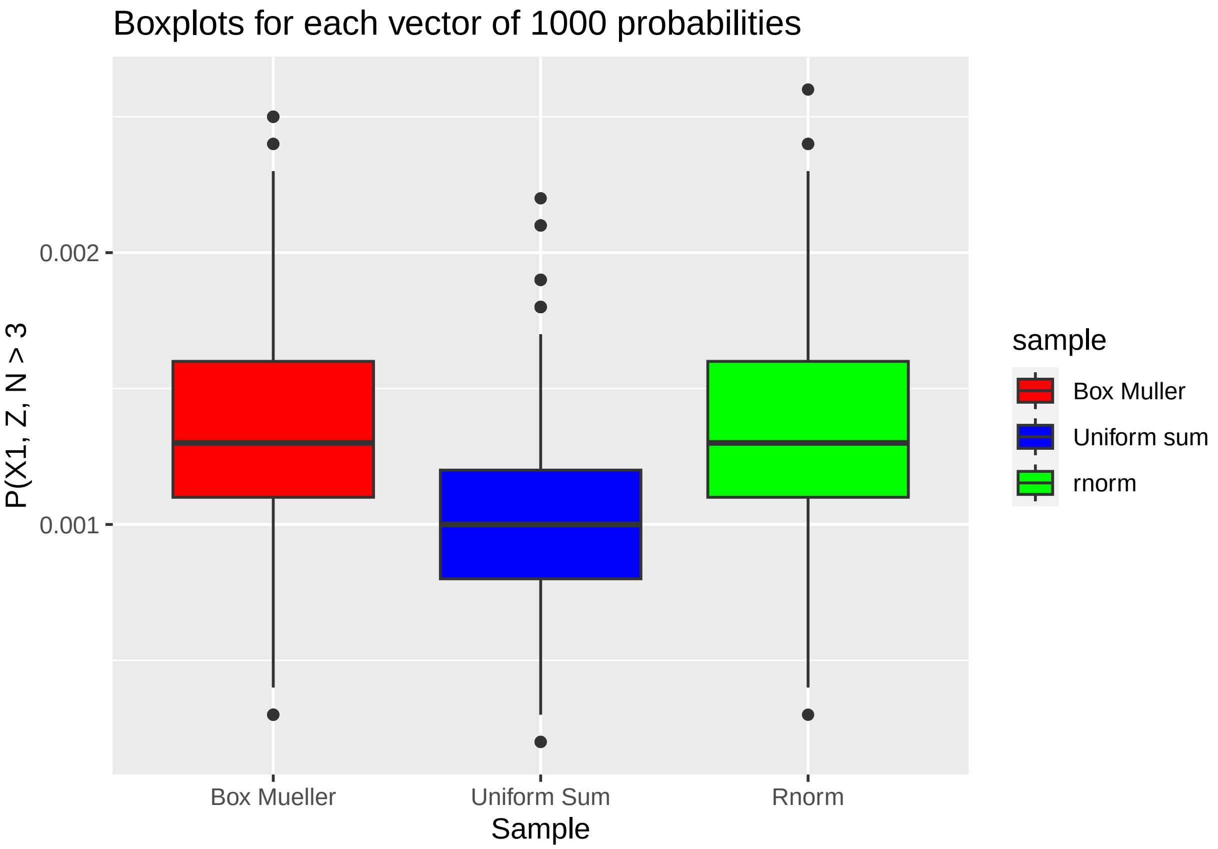
Let's now try to see the behavior of our generators drawing S = 1000 samples of size n = 1000 for each method and comparing the results stability.

```r
set.seed(123)

# Box Mueller
n <- 10000
S1 <- matrix(0, nrow=n, ncol=1000)

for (i in 1:1000) {
  u1 <- runif(n)
  u2 <- runif(n)
  S1[, i] <- sqrt(-2*log(u1))*cos(2*pi*u2)
}
i=1
# Uniform Sum
S2 <- matrix(0, nrow=n, ncol=1000)
for (i in 1:1000) {
  U_i <- matrix(runif(n*12, min=-1/2, max=1/2), nrow=n)
  S2[, i] <- rowSums(U_i)
}

# rnorm
S3 <- matrix(rnorm(n*1000, 0, 1), nrow=n, ncol=1000)
```

Now we can compute the tail probability for each sample of each method.

```r
prob_S1 <- apply(S1, 2, function(x) mean(x > 3))
prob_S2 <- apply(S2, 2, function(x) mean(x > 3))
prob_S3 <- apply(S3, 2, function(x) mean(x > 3))
```

```r
# dataframe with the sampled distributions for each method
df <- data.frame(
  sample = rep(c("Box Muller", "Uniform sum", "rnorm"), each = 1000),
  prob = c(prob_S1, prob_S2, prob_S3)
)
df$sample <- factor(df$sample, levels = c("Box Muller", "Uniform sum", "rnorm"))

# plot
ggplot(df, aes(x=sample, y=prob, fill=sample)) +
  geom_boxplot() +
  scale_fill_manual(values = c("red", "blue", "green")) +
  labs(x="Sample", y="P(X1, Z, N > 3") +
  ggtitle("Boxplots for each vector of 1000 probabilities") +
  scale_x_discrete(labels = c("Box Mueller", "Uniform Sum", "Rnorm"))
```

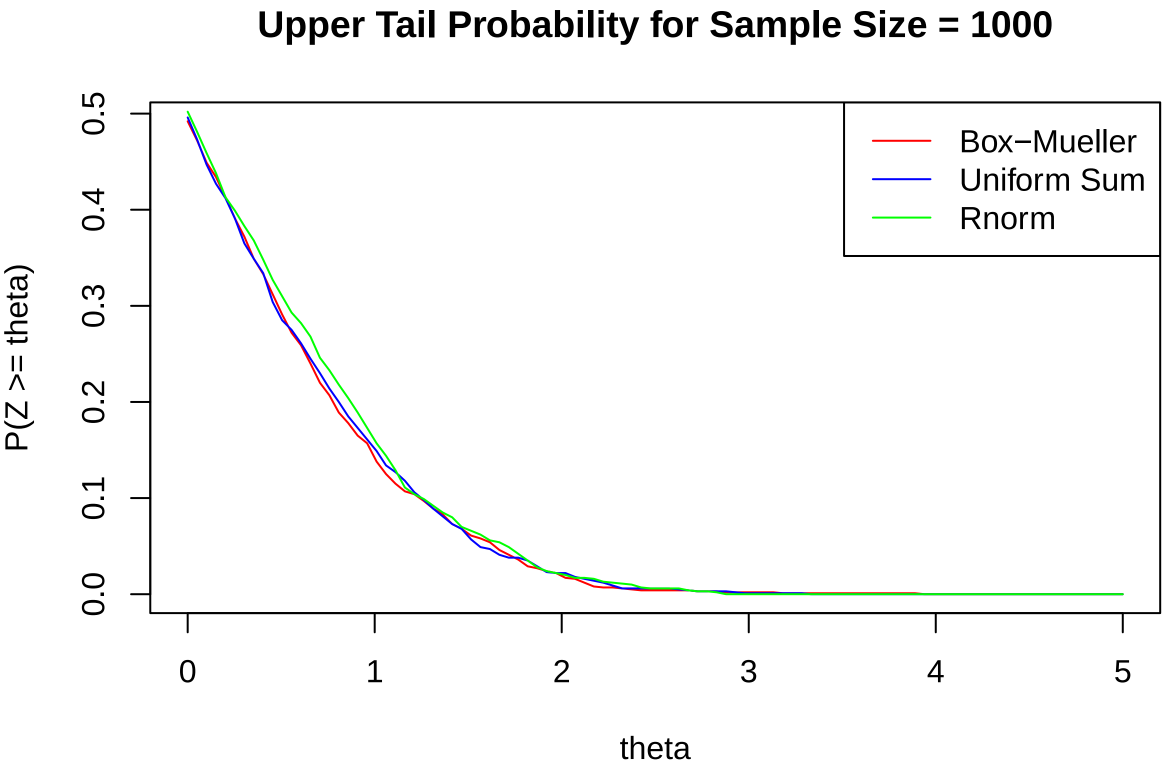


This result confirm what was inferred before, the box muelle algorithm “beats” the normal generator via summing 12 uniform random variables.

2.3 Point d)

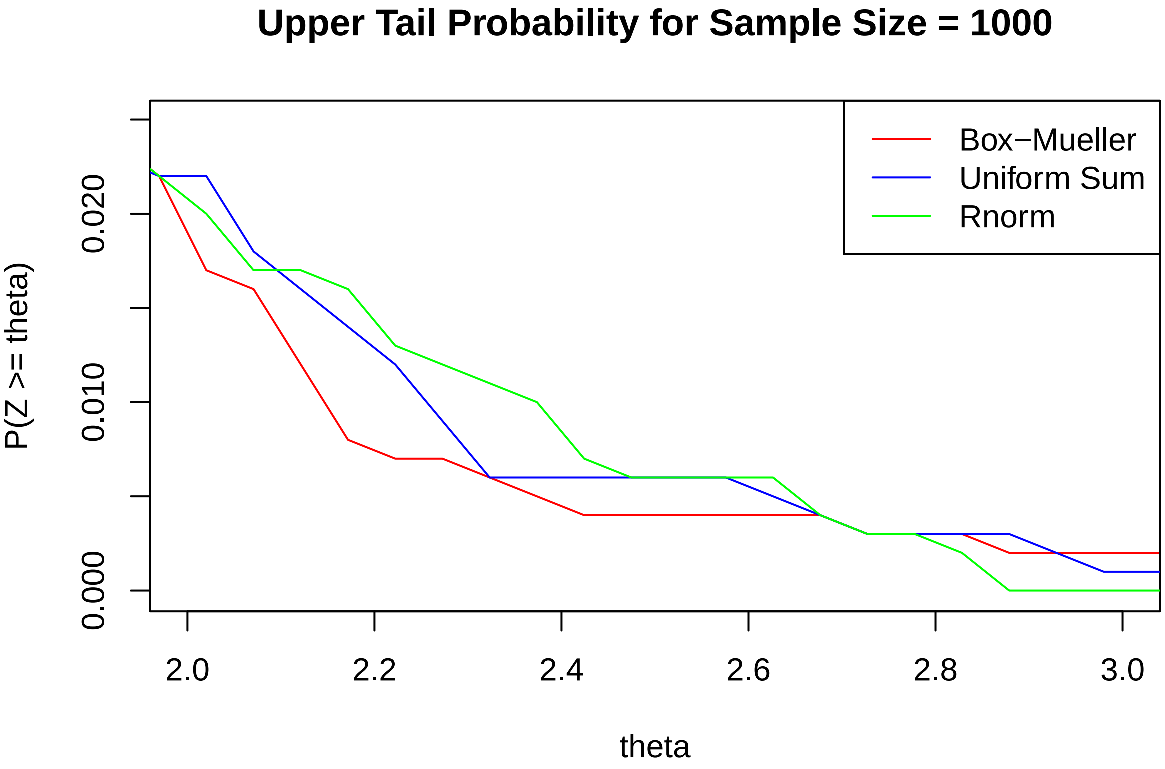
```
tail_prob_plot <- function(sample_size) {

  # generate samples
  set.seed(123)
  u1 = runif(sample_size)
  u2 = runif(sample_size)
  X1 = sqrt(-2*log(u1))*cos(2*pi*u2)
  Z = rowSums(matrix(runif(sample_size*12, min=-1/2, max=1/2), nrow=sample_size))
  N = rnorm(sample_size, 0, 1)

  # define range of theta values
  theta <- seq(0, 5, length.out = 100)

  # initialize vector for storing probabilities
  prob_X1 <- rep(0, length(theta))
  prob_Z <- rep(0, length(theta))
  prob_N <- rep(0, length(theta))

  # compute probabilities for each value of theta
  for (i in 1:length(theta)) {
    prob_X1[i] <- sum(X1 >= theta[i])/sample_size
    prob_Z[i] <- sum(Z >= theta[i])/sample_size
    prob_N[i] <- sum(N >= theta[i])/sample_size
  }

  # plot results
  plot(theta, prob_X1, type="l", col="red", xlab = "theta", ylab = "P(Z >= theta)",
       main = paste("Upper Tail Probability for Sample Size =", sample_size))
  lines(theta, prob_Z, type="l", col="blue")
  lines(theta, prob_N, type="l", col="green")
  legend("topright", legend = c("Box-Mueller", "Uniform Sum", "Rnorm"), col = c("red", "blue", "green")
}
```

```
tail_prob_plot(1000)
```



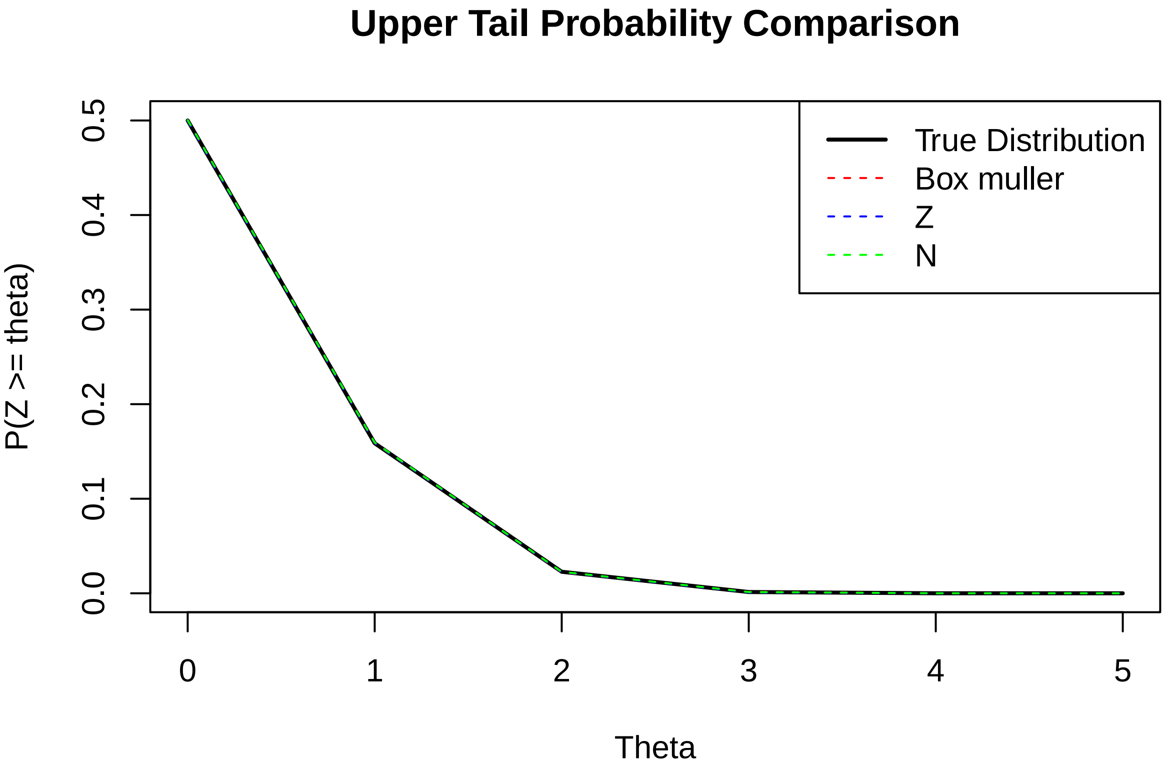
Zoom on the tail:

```
tail_prob_plot_lim <- function(sample_size) {

  # generate samples
  set.seed(123)
  u1 = runif(sample_size)
  u2 = runif(sample_size)
  X1 = sqrt(-2*log(u1))*cos(2*pi*u2)
  Z = rowSums(matrix(runif(sample_size*12, min=-1/2, max=1/2), nrow=sample_size))
  N = rnorm(sample_size, 0, 1)

  # define range of theta values
  theta <- seq(0, 5, length.out = 100)
  # initialize vector for storing probabilities
  prob_X1 <- rep(0, length(theta))
  prob_Z <- rep(0, length(theta))
  prob_N <- rep(0, length(theta))

  # compute probabilities for each value of theta
  for (i in 1:length(theta)) {
    prob_X1[i] <- sum(X1 >= theta[i])/sample_size
    prob_Z[i] <- sum(Z >= theta[i])/sample_size
    prob_N[i] <- sum(N >= theta[i])/sample_size
  }

  # plot results
  plot(theta, prob_X1, type="l", col="red", xlab = "theta", ylab = "P(Z >= theta)",
      main = paste("Upper Tail Probability for Sample Size =", sample_size), xlim = c(2, 3), ylim=c(-0
  lines(theta, prob_Z, type="l", col="blue")
  lines(theta, prob_N, type="l", col="green")
  legend("topright", legend = c("Box-Mueller", "Uniform Sum", "Rnorm"), col = c("red", "blue", "green")
}
tail_prob_plot_lim(1000)
```

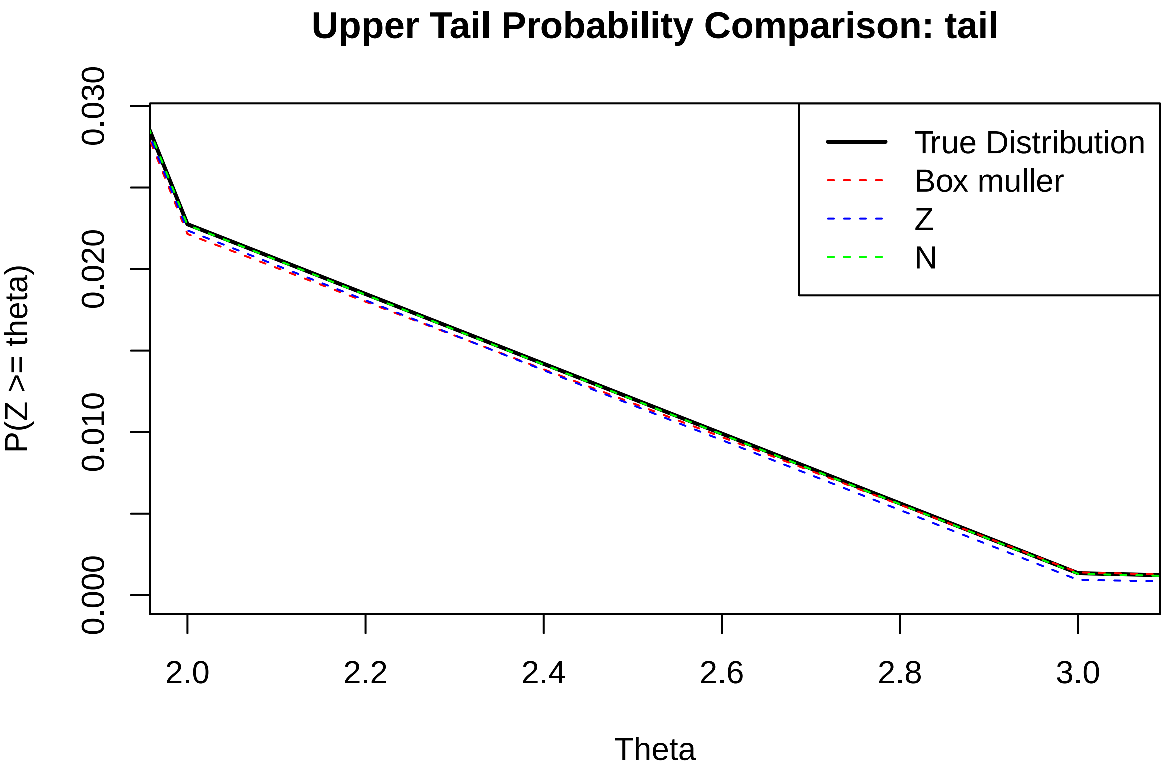


2.4 Point e)

```
# define the function to compute the upper tail probability
tail_prob <- function(theta) {
  return(pnorm(-theta))
}

# range for theta
theta <- seq(0, 5, length.out = 6)

# compute the upper tail probabilities for each theta
true_prob <- tail_prob(theta)
mc_prob_X1 <- sapply(theta, function(t) sum(X1 >= t) / length(X1))
mc_prob_Z <- sapply(theta, function(t) sum(Z >= t) / length(Z))
mc_prob_N <- sapply(theta, function(t) sum(N >= t) / length(N))

# plot using matplot
matplot(theta, cbind(true_prob, mc_prob_X1, mc_prob_Z, mc_prob_N),
        type = "l", lty = c(1,2,2,2), lwd = c(2,1,1,1),
        xlab = "Theta", ylab = "P(Z >= theta)",
        main = "Upper Tail Probability Comparison",
        col = c("black", "red", "blue", "green"))
legend("topright", c("True Distribution", "Box muller", "Z", "N"),
       lty = c(1,2,2,2), lwd = c(2,1,1,1),
       col = c("black", "red", "blue", "green"))
```



```
# Tail

matplot(theta, cbind(true_prob, mc_prob_X1, mc_prob_Z, mc_prob_N),
        type = "l", lty = c(1,2,2,2), lwd = c(2,1,1,1),
        xlab = "Theta", ylab = "P(Z >= theta)",
        main = "Upper Tail Probability Comparison: tail",
        col = c("black", "red", "blue", "green"), xlim=c(2,3.05), ylim=c(0,0.029))
legend("topright", c("True Distribution", "Box muller", "Z", "N"),
       lty = c(1,2,2,2), lwd = c(2,1,1,1),
       col = c("black", "red", "blue", "green"))
```



3 Montecarlo Simulation II

3.1 Point a)

We are showing that a random variable Y distributed as a Pareto distribution is only the transformation of a uniform U in particular: $Y = U^{-1/\gamma}$

$$\mathbb{P}(Y < y) = \mathbb{P}(U^{-1/\gamma} < y) = \mathbb{P}(U > y^{-\gamma}) =$$

$$= 1 - F_U(y^{-\gamma}) = 1 - \int_0^{y^{-\gamma}} 1dt = 1 - y^{-\gamma}$$

At the end of the day: $F_Y(y) = 1 - y^{-\gamma}$ deriving we obtain $f(x) = \gamma x^{-(\gamma+1)}$, the support is $I = (1, +\infty)$ because $f(x) \in (0, 1)$ only in I

3.2 point b)

We can find the distribution of $Y = log(X)$ with $X \sim \mathcal{P}\dashv\nabla\rceil\sqcup\wr(\gamma)$ in a similar way.

$$\mathbb{P}(Y < y) = \mathbb{P}(X < e^y) = \int_1^{e^y} \gamma x^{-(\gamma+1)} dx = 1 - e^{-\gamma y}$$

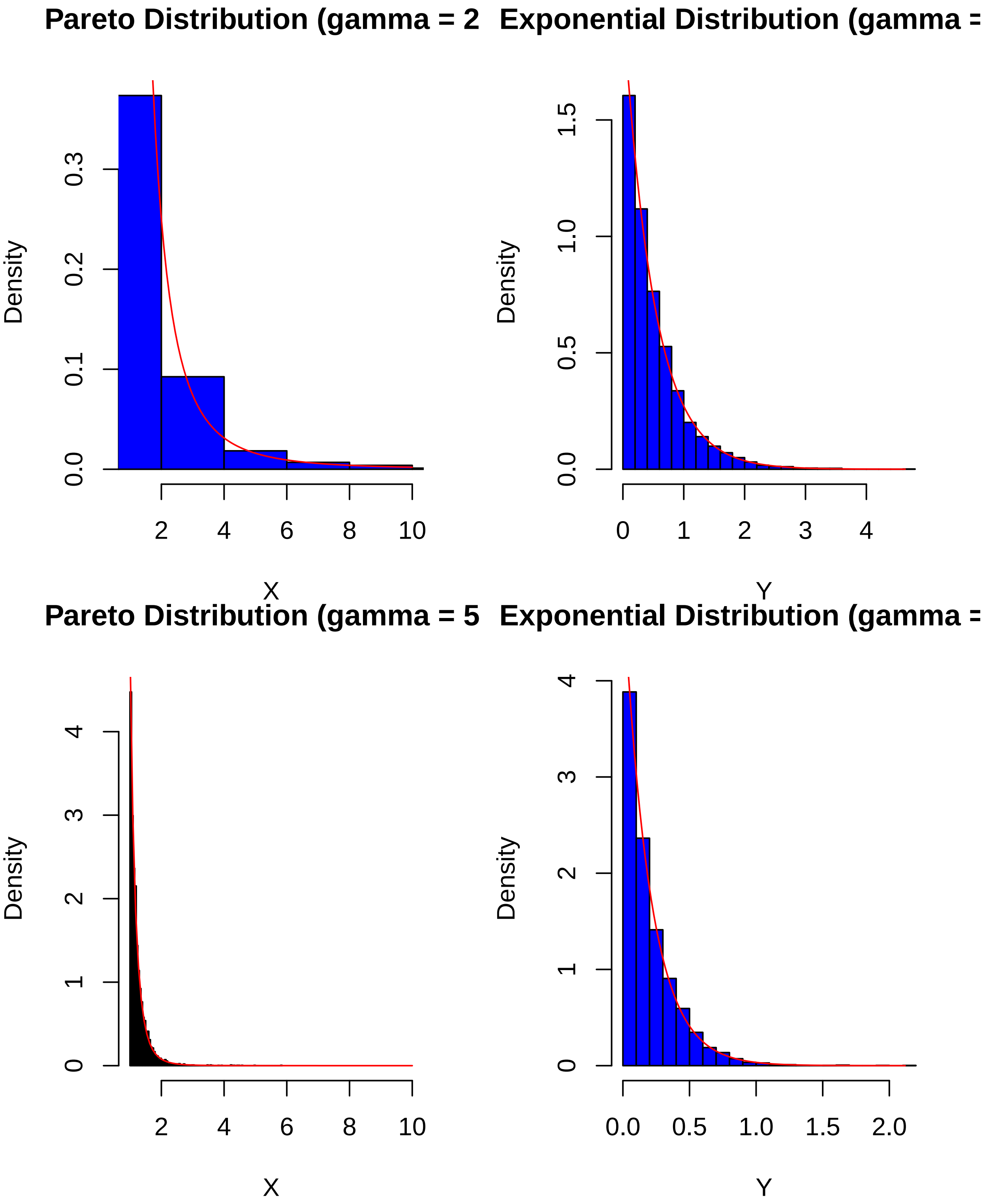
It means that $F_Y(y) = 1 - e^{-\gamma y}$; deriving we obtain $f(y) = \gamma e^{-\gamma * y}$, the support is $S = (0; +\infty)$ because $f(y)$ is in the support of the Pareto I only in that interval. With this PDF we will see that

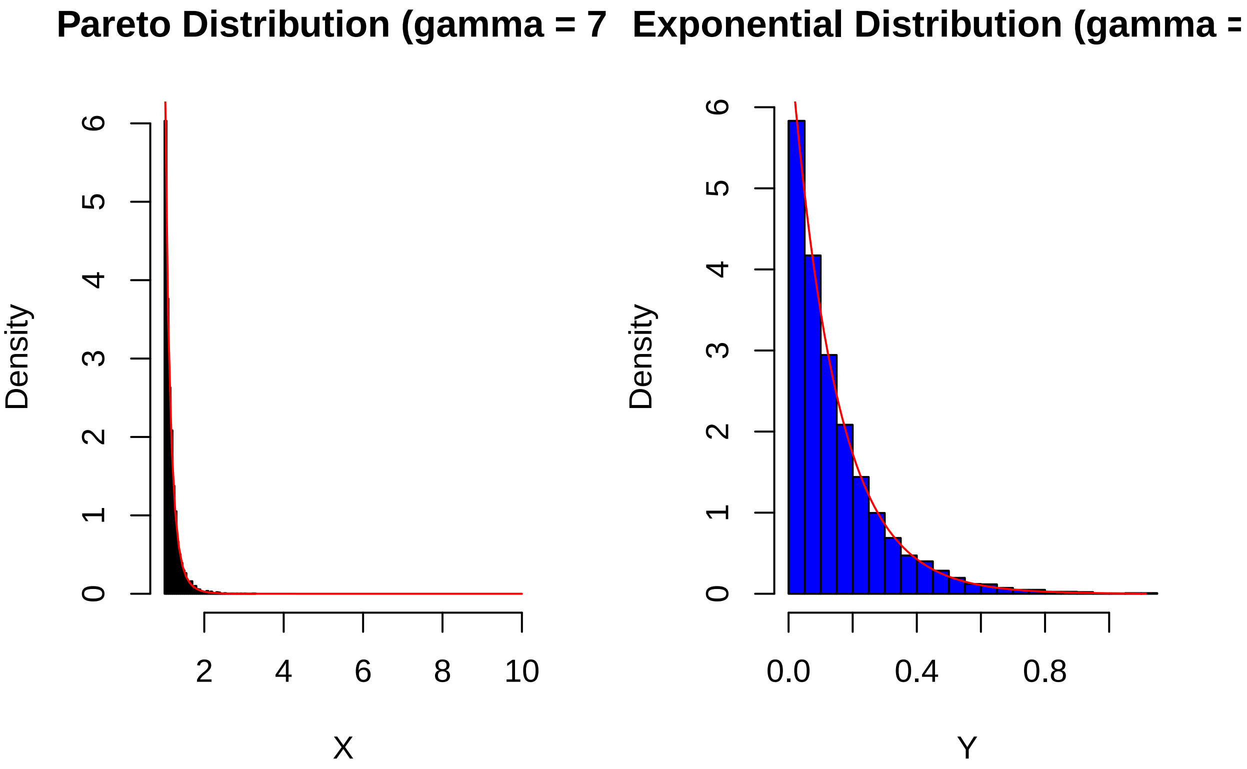
$$Y \sim \mathcal{E}(\gamma)$$

3.3 Point c)

```r
set.seed(123)

# Function to generate samples from a Pareto distribution
pareto_sampler <- function(n, gamma) {
  u <- runif(n)  # Generate n samples from a uniform distribution
  x <- (1/u)^(1/gamma)  # Transform the uniform samples to follow a Pareto distribution
  return(x)
}

# Function to generate samples from the exponential distribution
exponential_sampler <- function(n, gamma) {
  x <- pareto_sampler(n, gamma)  # Generate n samples from a Pareto distribution
  y <- log(x)  # Transform the Pareto samples to follow an exponential distribution
  return(y)
}

# Parameters
n <- 5000  # Number of samples
```

```r
gammas <- c(2, 5, 7)  # Different values of gamma to explore

# Generate samples and plot for each value of gamma
for (gamma in gammas) {
  # Generate samples from the Pareto distribution
  x_samples <- pareto_sampler(n, gamma)

  # Generate samples from the exponential distribution
  y_samples <- exponential_sampler(n, gamma)

  # Plot histograms
  par(mfrow = c(1, 2))
  hist(x_samples, breaks = 70, main = paste("Pareto Distribution (gamma =", gamma, ")"), col="blue", xla
  curve(dpareto(x, scale = 1, shape = gamma), col="red", add = TRUE)
  hist(y_samples, breaks = 30, main = paste("Exponential Distribution (gamma =", gamma, ")"), col="blue
curve(dexp(x, rate = gamma), col="red", add = TRUE, ylim = c(0, max(dexp(y_samples, rate = 1/gamma))))
}
```


Pareto Distribution (gamma = 2
Density
0.0
0.1
0.2
0.3
2
4
6
8
10
X
Exponential Distribution (gamma =
Density
0.0
0.5
1.0
1.5
0
1
2
3
4
Y
Pareto Distribution (gamma = 5
Density
0
1
2
3
4
2
4
6
8
10
X
Exponential Distribution (gamma =
Density
0
1
2
3
4
0.0
0.5
1.0
1.5
2.0
Y



Increasing Gamma leads to smaller tails in the Pareto distributions, meaning that the probability of observing larger values decreases a lot. The same story holds for the exponential distribution which presents a smaller right tail as gamma increases.

3.4 point d)

The Pareto cumulative distribution function is defined as:

$F(x) = 1 - \left(\frac{1}{x}\right)^{\gamma}, \quad x \geq 1$

To compute $\mathbb{P}(X < 5)$ we have to define:

$\mathbb{P}(X < 5) = F(5) = 1 - \left(\frac{1}{5}\right)^{\gamma}$

Now we just have to substitute a gamma value in this simple computation to find the value for $\mathbb{P}(X < 5)$.

Taking for example $\gamma = (2, 5, 7)$ we find:

$\mathbb{P}(X < 5) = 1 - \left(\frac{1}{5}\right)^{2} = 1 - \frac{1}{25} = \frac{24}{25} = 0.96$ $\mathbb{P}(X < 5) = 1 - \left(\frac{1}{5}\right)^{5} = 1 - \frac{1}{3125} \approx 0.99968$ $\mathbb{P}(X < 5) = 1 - \left(\frac{1}{5}\right)^{7} = 1 - \frac{1}{78125} \approx 0.99998$

These results can be checked via ppareto function:

```
ppareto(5, shape = gammas, scale=1)
```

```
## [1] 0.9600000 0.9996800 0.9999872
```

We now assess the precision of an MC estimate based on our Pareto Sampler, taking $\gamma = 2$.

```
# Monte Carlo estimate
gamma = 2
num = 100000  # Number of samples
samples = pareto_sampler(num, gamma)
p_mc = mean(samples < 5)
standard_error = sd(samples) / sqrt(num)
# Output results
p_mc
```

```
## [1] 0.95986
```

```
standard_error
```

```
## [1] 0.008581381
```

Running a big but still reasonable amount of samples (100.000) leads to a nice estimate. In particular the MC estimate based on our sampler underestimates the true value of the wanted probability by only a 0.004%, namely 0.95996 against 0.96. The standard error of our estimate is 0.0134.

4 Exercise 4: MC integration

Consider a standard Normal random variable X:

$$X \sim \mathcal{N}(\mu = 0, \sigma^2 = 1) \tag{1}$$

4.1 Point a

If we want to calculate $\mathbb{P}(X > 20)$ we could try to integrate the PDF of X in this way:

$$\mathbb{P}(X > 20) = 1 - \Phi(20) = 1 - \int_{-\infty}^{20} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx = \int_{20}^{+\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx \tag{2}$$

We immediately see that the integrand function is not so easy to be integrated (it is the son of the Gaussian function e^{x^2}). We will try to estimate this with the Monte Carlo method but, although this method is extremely powerful and flexible, there are some situations in which it can fail.

The reason why the Monte Carlo crude method can fail in this case, in particular in the estimation of the quantity $\mathbb{P}(X > 20)$, is the structure of this distribution. The standard normal is continuous long-tailed distribution (extended to infinity and decreasing very slowly), and the probability of getting a value (in this case, 20) very far from the mean (0 for definition) is extremely low, but not exactly zero. This means that to obtain an accurate estimate of the probability that this random variable is greater than 20, this method requires a large number of random numbers, which would require a significant amount of computation time and the approximation error could be very large.

Additionally, the Monte Carlo method requires the functions to be evaluated to be integrated or summed over the entire sampled space. The probability density function of a standard normal distribution does not have an analytical solution and therefore integration would require the use of numerical techniques that would further increase computational complexity.

Furthermore, the Crude Monte Carlo method uses a uniform distribution to generate random samples, which means that the samples are generated evenly spaced in the specified interval. However, the standard normal distribution has a particular shape this particular shape of the distribution makes it difficult to generate random samples sufficiently far from the mean to estimate the probability of very rare events.

4.2 Point b

Considering the change of variable $Y = \frac{1}{X}$ the integral will become:

$$\mathbb{P}(X > 20) = \mathbb{P}(Y < 1/20) = \int_0^{1/20} \frac{1}{\sqrt{2\pi} * y^2} e^{-\frac{(1/y)^2}{2}} dy \tag{3}$$

because $f_Y(y) = f_X(g(y))|\frac{\delta g(y)}{\delta y}|$ where:

- $g(y) = \frac{1}{y}$
- $\frac{\delta g(y)}{\delta y} = -\frac{1}{y^2}$
- $f_X(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$

As the first one, this integral has not have an analytic solution so we use the Monte Carlo method to estimate. The idea is to generate N random numbers from a uniform distribution between 0 and 1/20 that we obtain from the variable change. We can do this because now we have finite support where we can generate value.

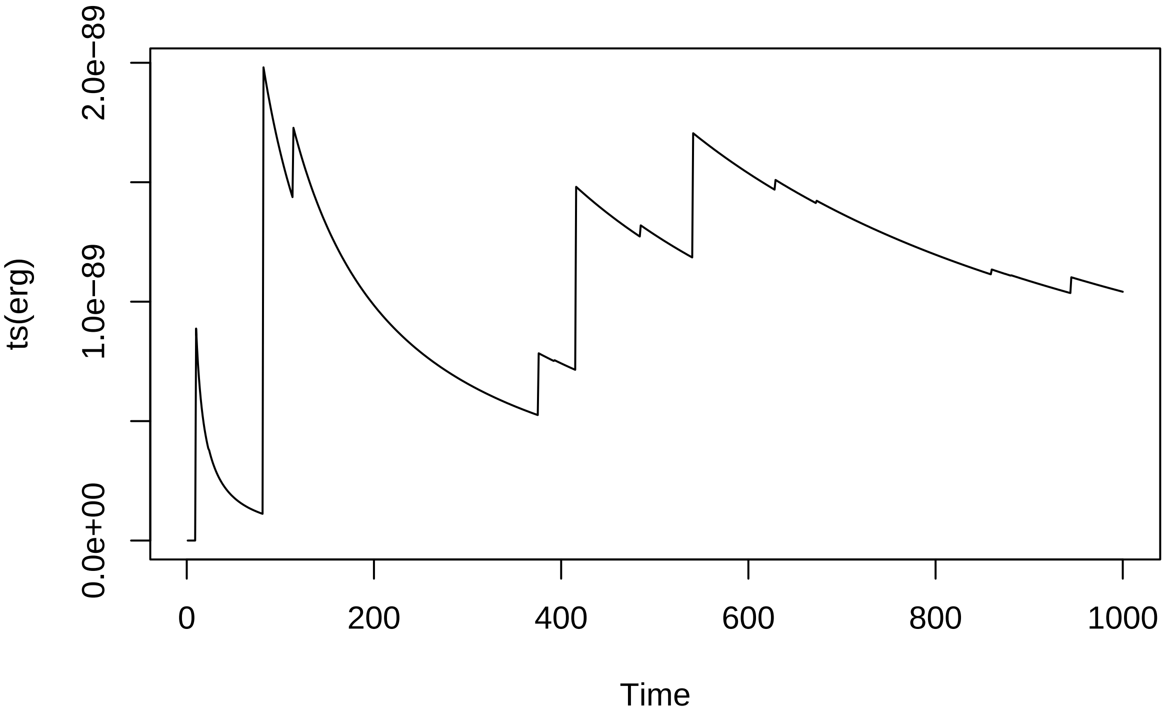
$$\mathbb{P}(X > 20) \approx \frac{0.05 - 0}{N} \sum_{i=1}^{N} \frac{1}{\sqrt{2\pi} y_i^2} \cdot e^{-\frac{(1/y_i)^2}{2}} \tag{4}$$

```
rm(list = ls())
set.seed(1234)

# Define the PDF of N(0,1)
f = function(x) 1/(sqrt(2*pi)*x^2)*exp(-(1/x)^2/2)

# Function that generate n MC simulations in the interval (a,b)
mc = function(n, a=0, b=1/20){

  if(a>b){stop("reconsider your interval")}

  x = runif(n,a,b)

  mean(f(x)) * (b-a)

}

# Simulate
mc(1000)
```

```
## [1] 3.258382e-89
```

```
# Function to create the ergodic plot
ergodic = function(n_max, a=0, b=1/20){

  x = runif(n_max,a,b)

  (b-a) * cumsum(f(x))/c(1:n_max)

}
```

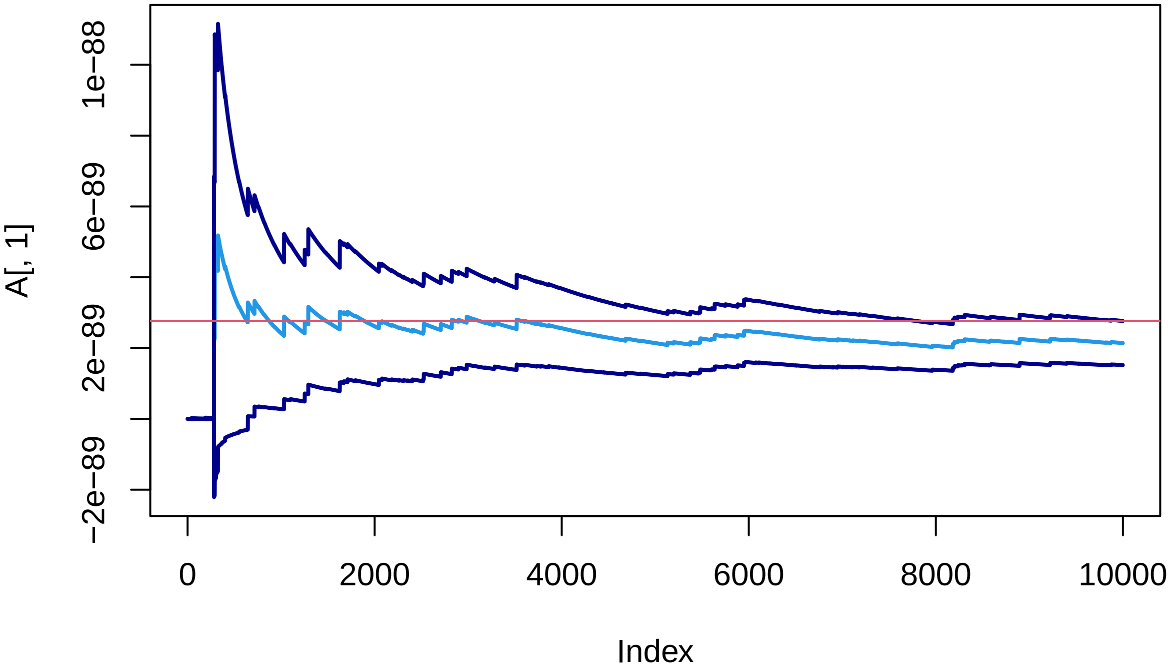
```
# Lauch the function
erg = ergodic(1000)

# Plot results
plot(ts(erg))
```



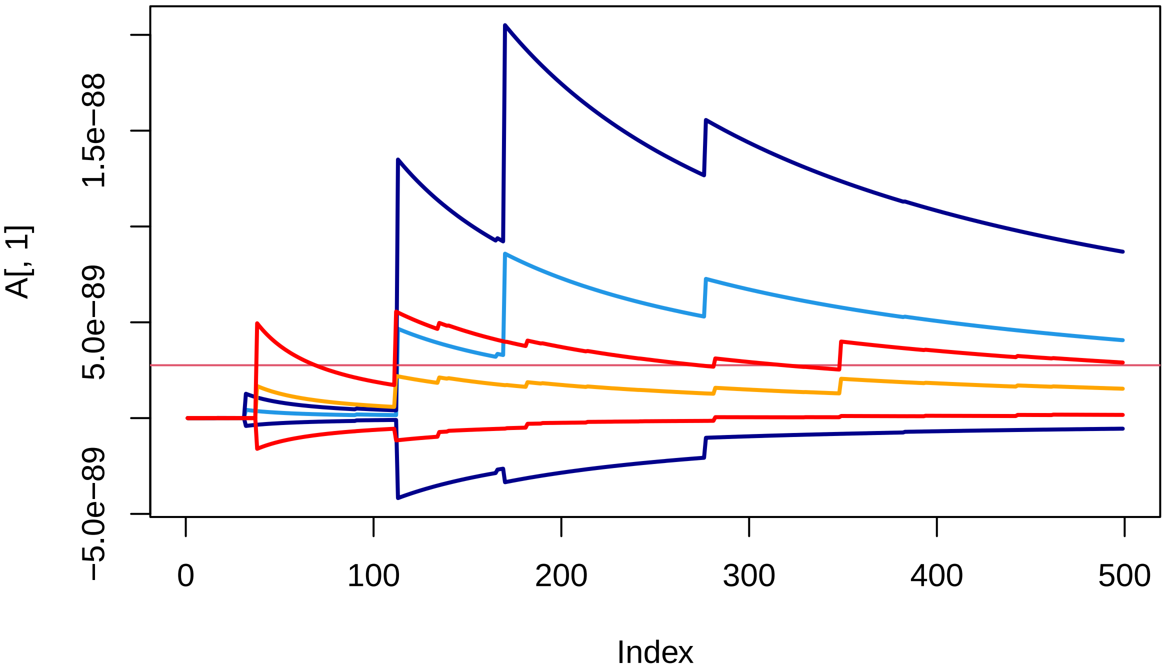
The ergodic plot a tool used to assess the convergence of results obtained. Ergodicity is an important property of the Monte Carlo method, which states that the system reaches a statistically representative equilibrium state after a sufficient number of iterations and visualize the convergence of results to the expected value or correct solution.

##Point c Now we try to construct a more efficient estimator using antithetic variables it will be more efficient because it reduces the variance of the estimator because we are using 2 unbiased estimators identically distributed but negatively correlated and for definition the variance decreases.

```
# Calculate the SD of each iteration
cumsd_f = function(x){

  sapply(2:length(x), function(r) sd(f(x[1:r])))

}

# Implement the ergodic function
ergodic2 = function(n_max, a=0, b=1/20){

  x = runif(n_max,a,b)

  thetahat = (b-a) * cumsum(f(x))/c(1:n_max)


  sdthetahat = (b-a)/sqrt(2:n_max) * cumsd_f(x)
  return(cbind(thetah = thetahat[-1], sd = sdthetahat))
}

# Plot the ergodic function and its uncertainty
plot_ergodic = function(output,theta){

  A = output
  x_top =    max(A[,1]+ 1.96 * A[,2])
  x_bottom = min(A[,1]- 1.96 * A[,2])

  plot(A[,1],type="l",col=4,lwd=2,ylim = c(x_bottom,x_top))
  lines(A[,1] + 1.96 * A[,2],
        type="l",col="darkblue",lwd=2)
  lines(A[,1] - 1.96 * A[,2],
        type="l",col="darkblue",lwd=2)

  abline(h=theta,col=2)

}
# use the output of the integrate function as the real value
theta = integrate(f,0,1/20)
output = ergodic2(10000)
plot_ergodic(output,theta = theta[1])
```



```
# Repeat everything but using antithetic variables
MC_ergodic = function(n_max, a=0, b=1/20){

  x = runif(n_max,a,b)

  thetahat = (b-a) * cumsum(f(x))/c(1:n_max)
  sdthetahat = (b-a)/sqrt(2:n_max) * cumsd_f(x)
  return(cbind(thetah = thetahat[-1], sd = sdthetahat))
}

cumsd_AV = function(x,xprime){
  sapply(2:length(x), function(r)
    sd( (f(x[1:r])+f(xprime[1:r])) /2 ) )
}

AV_ergodic = function(n_max, a=0, b=1/20){

  x        = runif(n_max,a,b)
  xprime   = a+b-x
  h = (f(x) + f(xprime))/2
  thetahat = (b-a) * cumsum(h)/c(1:n_max)

  sdthetahat = (b-a)/sqrt(2:n_max) * cumsd_AV(x,xprime)

  return(cbind(thetah = thetahat[-1], sd = sdthetahat))
}


# Compare results
MC_res = MC_ergodic(500)
plot_ergodic(MC_res, theta[1])


AV_res = AV_ergodic(n_max = 500)
A = AV_ergodic(n_max = 500)
lines(A[,1],type="l",lwd=2,col="orange")
lines(A[,1] + 1.96 * A[,2],
      type="l",col="red",lwd=2)
lines(A[,1] - 1.96 * A[,2],
      type="l",col="red",lwd=2)
```



We obtain a more efficient point and interval estimation with a smaller standard deviation and nearest estimation of the integral to the *real* value given by the `integrate` function of `R`

```
MC_res[499,]
```

```
##       thetah           sd
## 4.065196e-89 2.356068e-89
```

```
AV_res[499,]
```

```
##       thetah           sd
## 3.674989e-89 1.546452e-89
```

```
theta[1]
```

```
## $value
## [1] 2.759158e-89
```

4.3 Point d

Finally, we can compare our results to the output of `R` function that should give the probability that we are looking for, `pnorm` but it gives the asymptotic result 0. Our method gives a more accurate result.

```
theta[1]
```

```
## $value
## [1] 2.759158e-89
```

```
1-pnorm(20,0,1)
```

```
## [1] 0
```