Monte Carlo Simulation with R

FE522

Nov 5, 2016

Contents

Briefly About Monte Carlo Simulation

Monte Carlo methods in the most basic form is used to approximate to a result aggregating repeated probabilistic experiments. For instance; to find the true probability of heads in a coin toss repeat the coin toss enough (e.g. 100 times) and calculate the probability by dividing number of heads to the total number of experiments. Here is a small example.

```
#Function of coin tossing with the given number of instances n_toss
toss_coins<-function(n_toss){
    sample(c("H","T"),n_toss,replace=TRUE)
}
#Do the experiment with 10 instances
experiment_1<-toss_coins(10)
#Print out the tosses to check the function
print(experiment_1)
```

```
##  [1] "T" "T" "T" "T" "H" "T" "H" "T" "T" "T"
```

```
sum(experiment_1=="H")/10
```

```
## [1] 0.2
```

```
#Now repeat the experiment with more instances
experiment_2<-toss_coins(10^4)
sum(experiment_2=="H")/10^4
```

```
## [1] 0.5007
```

Monte Carlo and its extensions (e.g. Markov Chain Monte Carlo) are generally used to find the results of very complex or analytically intractable calculations. For instance one of the earlier examples of MC methods, Metropolis Algorithm, is devised by Manhattan Project members and it is used in mathematical physics to understand the particle movements of the atomic bomb.

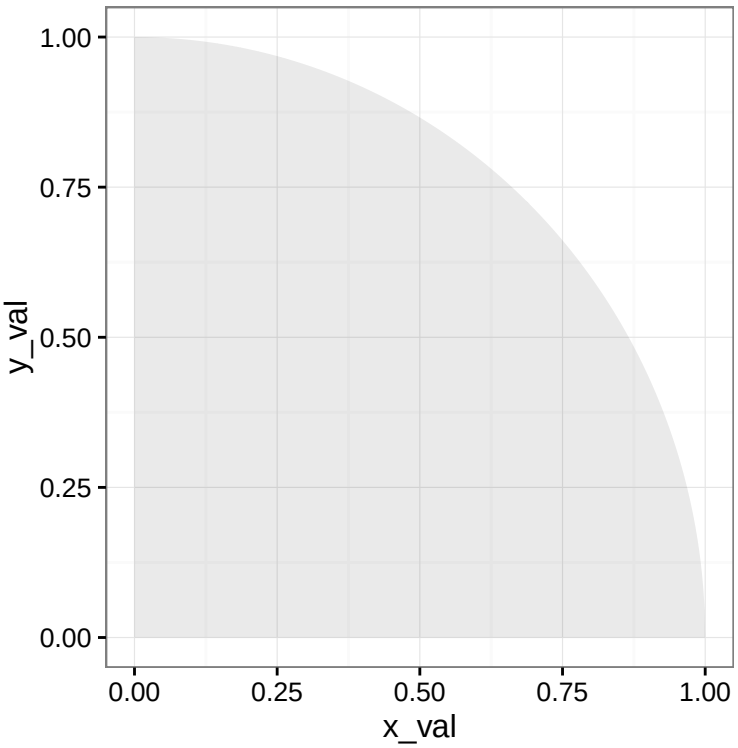
Before starting calculating options with Monte Carlo methods we will start with some toy examples and random variate generation. Aside from the lecture notes, I will also partly follow BOUN CMPE584 lecture notes and IE 586 lecture notes (not publicly available).

Toy Example: Calculation of π

To calculate the π value, think of a quarter of a unit circle (round with radius value 1).

```
#Create the points necessary to build a polygon
circle_data<-data.frame(x_val=c(0,seq(0,1,length.out=1000)),
                        y_val=c(0,sqrt(1-seq(0,1,length.out=1000)^2)))

library(ggplot2) #Load ggplot2
#create the shaded area for the quarter circle
plot_pi <- ggplot() +
geom_polygon(data=circle_data,aes(x=x_val,y=y_val),alpha=0.1) + theme_bw()
plot_pi
```



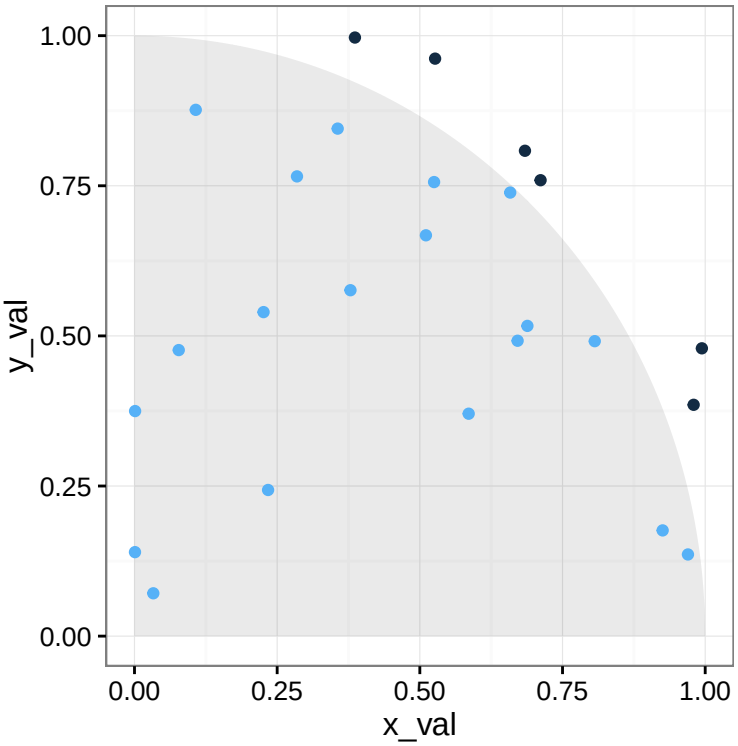
Now, let's randomly put dots on the unit square (i.e. square with side length of 1). Then, we define them "in" or "out" depending on whether they are within the circle area or not.

```
#Create random points by declaring dot positions
#with 2 random uniformly distributed values for x and y.
dot_data<-data.frame(x_val=runif(25),y_val=runif(25))
#Define in with 1 and out with 0.
```

```r
#Since circle radius is 1. We calculate in/out with the distance from origin.
dot_data$in_or_out<-ifelse(sqrt(dot_data$x_val^2+dot_data$y_val^2)<=1,1,0)
head(dot_data)
```

```
##          x_val     y_val in_or_out
## 1 0.3560381935 0.8451746         1
## 2 0.6841733987 0.8082791         0
## 3 0.0008856533 0.1399003         1
## 4 0.8064633757 0.4911957         1
## 5 0.5106650034 0.6675278         1
## 6 0.9941859178 0.4793008         0
```

```r
plot_pi + geom_point(data=dot_data,aes(x=x_val,y=y_val,color=in_or_out)) +
theme(legend.position="none")
```



Ratio of number of dots within the circle area to the total number of dots will give us the ratio of the quarter unit circle to the unit square. We know the area of the quarter circle as $\pi r^2/4$ and square with side length equal to the radius as r^2. The ratio result as $\pi/4$. So, 4 times the ratio of dots should give us approximately the value of π.

```r
#Simulated value of pi
4*sum(dot_data$in_or_out)/nrow(dot_data)
```

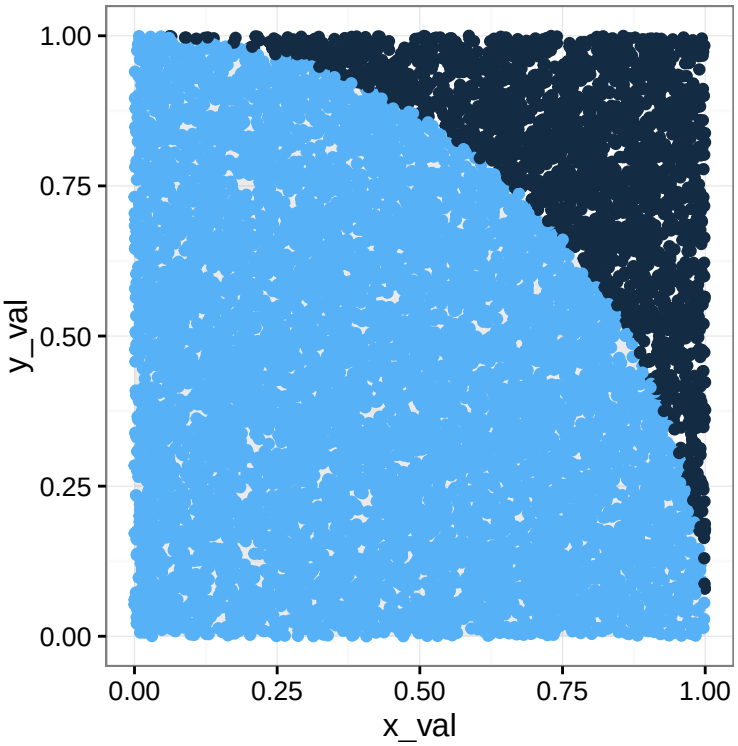
```
## [1] 3.04
```

```r
#True value of pi
pi
```

```
## [1] 3.141593
```

Our simulated value of pi is not very satisfactory. Let's increase the sample size from 25 to 10000.

```
dot_data_2<-data.frame(x_val=runif(10^4),y_val=runif(10^4))
dot_data_2$in_or_out<-ifelse(sqrt(dot_data_2$x_val^2+dot_data_2$y_val^2)<=1,1,0)
plot_pi + geom_point(data=dot_data_2,aes(x=x_val,y=y_val,color=in_or_out)) +
theme(legend.position="none")
```




```
4*sum(dot_data_2$in_or_out)/nrow(dot_data_2)
```

```
## [1] 3.1444
```

Better.

Toy Example: Chevalier de Méré

Directly quoting from Cut the Knot:

"A 17th century gambler, the Chevalier de Méré, made it to history by turning to Blaise Pascal for an explanation of his unexpectd losses. Pascal combined his efforts with his friend Pierre de Fermat and the two of them laid out mathematical foundations for the theory of probability.

Gamblers in the 1717 France were used to bet on the event of getting at least one 1 (ace) in four rolls of a dice. As a more trying variation, two die were rolled 24 times with a bet on having at least one double ace. According to the reasoning of Chevalier de Méré, two aces in two rolls are 1/6 as likely as 1 ace in one roll. (Which is correct.) To compensate, de Méré thought, the two die should be rolled 6 times. And to achieve the probability of 1 ace in four rolls, the number of the rolls should be increased four fold - to 24. Thus reasoned Chevalier de Méré who expected a couple of aces to turn up in 24 double rolls with the frequency of an ace in 4 single rolls. However, he lost consistently."

TLDR, there are two games with ordinary dice:

- Getting a 6 on 4 rolls of a single die
- Getting double 6 (*düşeş*) on 24 rolls of paired dice

Our gambling knight turned from Game 1 to Game 2 and asks why he starts losing money (assume at each game you either gain or lose 1 gold). Let's calculate the probability of win for each case with simulation.

Game 1: One six in a four roll

```
#Set number of trials
n_trial<-10^4
#Roll the dice
#Each row is the result of 4 rolls
die_toss<-matrix(sample(1:6,n_trial*4,replace=TRUE),ncol=4)
head(die_toss)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    6    1    5    5
## [2,]    6    1    5    3
## [3,]    5    3    5    6
## [4,]    1    3    4    4
## [5,]    6    6    4    1
## [6,]    1    4    2    2
```

```
#See if the dice are 6 or not
die_toss<-die_toss==6
head(die_toss)
```

```
##       [,1]  [,2]  [,3]  [,4]
## [1,]  TRUE FALSE FALSE FALSE
## [2,]  TRUE FALSE FALSE FALSE
## [3,] FALSE FALSE FALSE  TRUE
## [4,] FALSE FALSE FALSE FALSE
## [5,]  TRUE  TRUE FALSE FALSE
## [6,] FALSE FALSE FALSE FALSE
```

```
#Calculate the wins
wins<-rowSums(die_toss)>=1
head(wins)
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

```
#Calculate the winning proportion
sum(wins)/n_trial
```

```
## [1] 0.5167
```

True probability can be calculated as $1 - P(Loss)^4 = 1 - (5/6)^4 = 0.518$.

Game 2: Double-six in 24 rolls of paired dice

Exercise!

True probability can be calculated as $1 - P(Loss)^{24} = 1 - (35/36)^2 4 = 0.491$.

Example: Pricing European Options

Suppose we want to price an option with the initial asset price $S_0 = 100$, strike price $K = 100$, risk free rate $r = 0.02$, volatility $\sigma = 0.25$ and maturity $T = 1$ year. Risk-neutral pricing process is as follows.

$$S_T = S_0 * exp((r - \sigma^2/2)T + \sigma Z\sqrt{T})$$

where Z is a random standard normal variate $N(0, 1)$.

```
#Let's build a function to simulate
sim_european_call<-function(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=10^4){
    #Simulate the stock
    sim_S_T<-S_0*exp((r-0.5*vol^2)*T_years + vol*rnorm(n)*sqrt(T))
    #Calculate payoffs
    payoffs<-pmax(sim_S_T-K,0)*exp(-r*T_years)
    #Simulate results and bounds
    Price<-mean(payoffs)
    SE<-1.96*sd(payoffs)/sqrt(n)
    LowerB <- Price - SE
    UpperB <- Price + SE
    return(c(Price=Price,SE=SE,Lower=LowerB,Upper=UpperB))
}
#Simulation with 1000 instances
sim_european_call(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=10^3)
```

```
##     Price        SE     Lower     Upper
## 10.129527  1.061294  9.068233 11.190821
```

```
#Simulation with 10000 instances
sim_european_call(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=10^4)
```

```
##      Price         SE      Lower      Upper
## 10.9288232  0.3447952 10.5840280 11.2736184
```

```
#Simulation with 100000 instances
sim_european_call(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=10^5)
```

```
##      Price         SE      Lower      Upper
## 10.9078898  0.1094115 10.7984782 11.0173013
```

```
#Compare with Black Scholes value
black_scholes_eopt(s0=100,K=100,r=0.02,T_in_days=252,sig=0.25,callOrPut="call")
```

```
## [1] 10.87056
```

We will return to option pricing in the following sections in detail.

Random Number Generation

Linear Congruential Generator

$$s_{i+1} = (as_i + b), \quad \text{mod } m$$

Let's make a home made LCG

```r
#This is the generating function.
lcg_generator <- function(s_i,m,a=7^5,b=0){
    val<-(a*s_i+b)%%m
}
#This returns a vector of pseudonumbers given the seed.
lcg_rand<-function(n,s_0=123457,m=2^35-1,...){

    s_vector<-lcg_generator(s_i=s_0,m=m,...)

    for(i in 2:n){
        s_vector[i]<-lcg_generator(s_i=s_vector[i-1],m=m,...)
    }
    #We make it this way to never get a 0 or 1
    return((s_vector+0.5)/m)
}
lcg_rand(n=10)
```

```
##  [1] 0.06038875 0.95379398 0.41544445 0.37485500 0.18793875 0.68649200
##  [7] 0.87100583 0.99494424 0.02791106 0.10125041
```

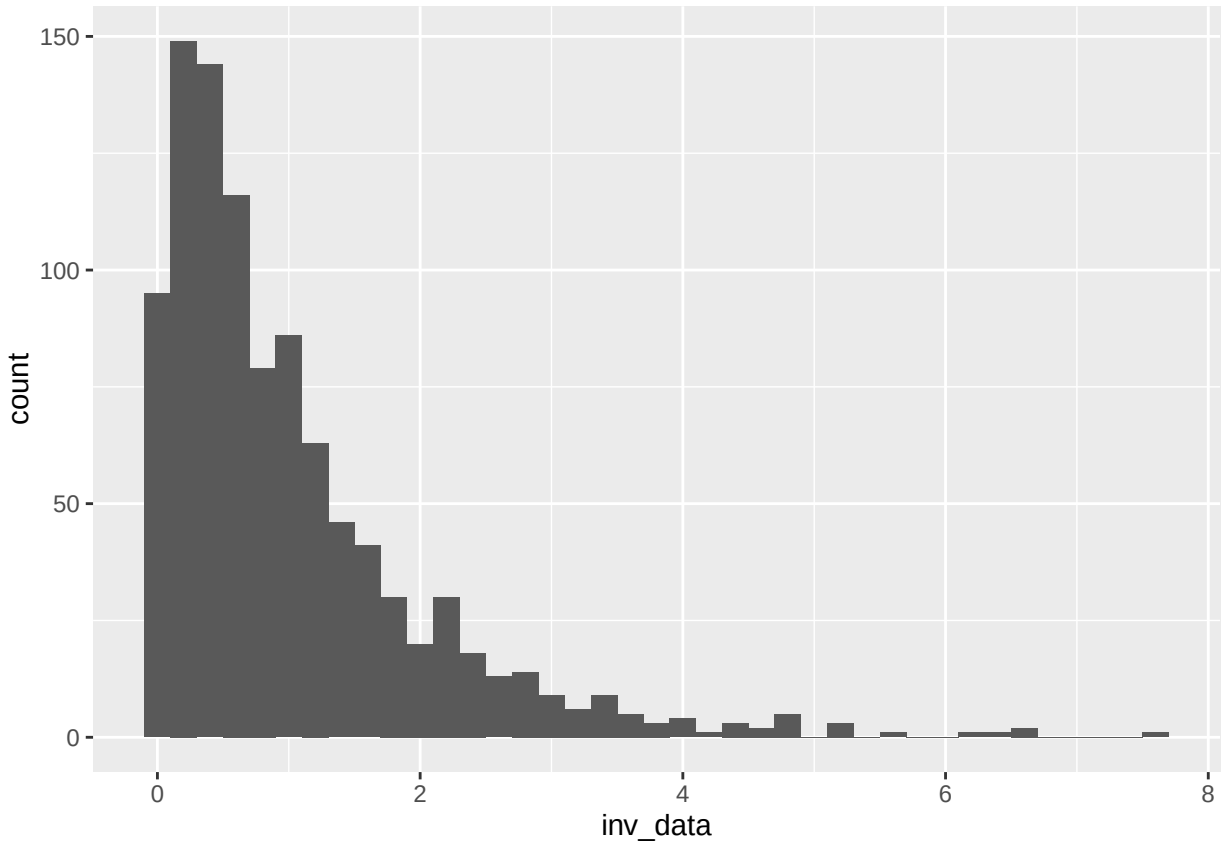
LCG is not the best pseudorandom number generator. Also, there might be some unwanted consequences, for instance someone win the lottery four times, if your randomness pattern is not so random.

Note: Mersenne Twister is the most popular random number generation algorithm with a period of $2^{19937} - 1$. It is also used by R.

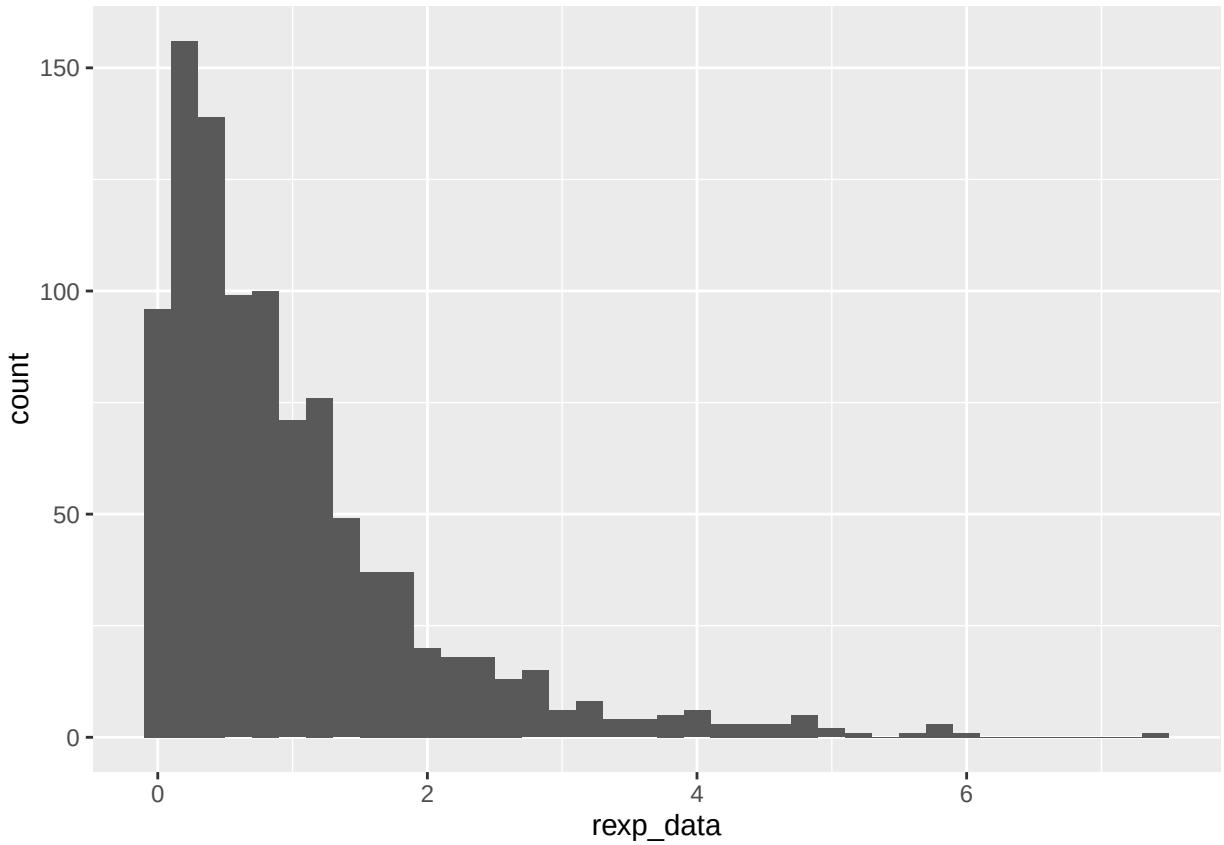
Inversion Method

Generating random variates from non-uniform distributions is not as straightforward. One way to do it is inversion method. Basically, take the inverse of the cumulative distribution function and put a uniform random number as the input. Resulting number is a random variate from the corresponding distribution.

```r
#Inverse CDF of exponential distribution
inverse_exp<-function(u,mu=1){
    return(-log(1-u)/mu)
}
#Prepare the comparison data
compare_data<-data.frame(inv_data=inverse_exp(u=runif(1000)),rexp_data=rexp(1000))
#Plot the inversion method histogram
ggplot(data=compare_data) + geom_histogram(aes(x=inv_data),binwidth=0.2)
```




```
#Plot R random exponential function histogram
ggplot(data=compare_data) + geom_histogram(aes(x=rexp_data),binwidth=0.2)
```



One disadvantage of this method is inverse CDF should be formulated. As a counter example, generating normally distributed random variates is not easy with inversion.

Simulating Normal Random Variates: Box-Müller and Marsaglia

Generating two normally distributed random numbers is possible by using two uniformly distributed random numbers. Box-Müller took advantage of the knowledge that multinormal standard distribution is "radially symmetric". It means that if we plot two standart normal distributions midpoints at the origin; for each circle centered at the origin its points have the same probability. It has the density function of $f(x, y) = exp(-0.5(x^2 + y^2))/(2\pi)$.

Let's test it. Think of the unit circle. If multinormal standard distribution is radially symmetric then each point on the unit circle should yield the same density value.

```
#sinpi calculates the sinus value in pi degrees. sinpi(1/2) == sin(pi/2)
#cospi as well
dnorm(sinpi(1/2))*dnorm(cospi(1/2))
```

```
## [1] 0.09653235
```

```
dnorm(sinpi(1/3))*dnorm(cospi(1/3))
```

```
## [1] 0.09653235
```

```
dnorm(sinpi(2/3))*dnorm(cospi(2/3))
```

```
## [1] 0.09653235
```

What we are going to do is we will generate two uniform random numbers. The first one will be used to

give a random distance from the origin. The second one will be used to give a random angle and sin and cos values will determine the two random normal variates.

```
#Generate uniform random variates
a_val<-runif(1)
a_val
```

```
## [1] 0.5054256
```

```
b_val<-runif(1)
b_val
```

```
## [1] 0.8561711
```

```
#Distance
dist<- sqrt(-2*log(a_val))
dist
```

```
## [1] 1.168208
```

```
rn_1<- dist*sin(2*pi*b_val)
rn_1
```

```
## [1] -0.9177714
```

```
rn_2<- dist*cos(2*pi*b_val)
rn_2
```

```
## [1] 0.7227755
```

Marsaglia method eliminates the necessity of calculating trigonometric functions. But we need our random uniform variates satisfy some conditions.

```
#Repeat function is a loop function
#It simoply repeats its contents unless break condition is satisfied
repeat{
    a_val<-2*runif(1)-1
    b_val<-2*runif(1)-1
    w_val<-a_val^2+b_val^2
    print(w_val)
    if(w_val < 1){
        break
    }
}
```

```
## [1] 0.2356639
```

```
#Generate standard normal random variates
z_1<-a_val*sqrt(-2*log(w_val)/w_val)
z_2<-b_val*sqrt(-2*log(w_val)/w_val)
```

Simulation with Multinormal Distribution

Assume we have a correlated multinormal distribution with covariance matrix Σ and we would like to generate multinormal random variates. We know that $\Sigma_{i,j} = \rho_{i,j} * \sigma_i * \sigma_j$ and $\Sigma = \sigma\prime.R.\sigma$ where $\rho_{i,j}$ is the correlation coefficient between i and j, σ is the standard deviation vector and R is the correlation matrix. The formula for generating a correlated multinormal vector is as follow.

$$N = \mu + AZ$$

where A is the lower triangle Cholesky decomposition of the covariance matrix Σ; $A.A^T = \Sigma$ and Z is a vector of standard normal variates. Let's code it on a toy example.

```
#First create 3 random normal processes
z1<-rnorm(100,mean=0.1,sd=0.1)
z2<-rnorm(100,mean=0.5,sd=0.4)
z3<-rnorm(100,mean=0.7,sd=0.5)
#Combine them in a matrix
z_mat<-matrix(c(z1,z2,z3),ncol=3)
#Calculate the covariance matrix
cov_mat<-cov(z_mat)
cov_mat
```

```
##             [,1]        [,2]         [,3]
## [1,] 0.008587765  0.00371732  0.003439619
## [2,] 0.003717320  0.14173397 -0.024521071
## [3,] 0.003439619 -0.02452107  0.302184919
```

```
#Calculate Cholesky matrix and get the lower triangle.
chol_mat<-t(chol(cov_mat))
#Generate the random variates by using
#mu+A.Z
n_vec <- c(0.1,0.5,0.7) + chol_mat%*%rnorm(3,mean=0,sd=1)
n_vec
```

```
##           [,1]
## [1,] 0.2355003
## [2,] 0.4778278
## [3,] 1.2131973
```

We can use correlated multinormal processes in the context of simulating a portfolio of correlated assets. Let's recall the `EuStockMarkets` data.

```
#Convert data frame to matrix
eu_mat<-as.matrix(EuStockMarkets)
head(eu_mat)
```

```
##          DAX    SMI    CAC   FTSE
## [1,] 1628.75 1678.1 1772.8 2443.6
## [2,] 1613.63 1688.5 1750.5 2460.2
## [3,] 1606.51 1678.6 1718.0 2448.2
## [4,] 1621.04 1684.1 1708.1 2470.4
## [5,] 1618.16 1686.6 1723.1 2484.7
## [6,] 1610.61 1671.6 1714.3 2466.8
```

```
#Get the log-return matrix
log_returns<-log(eu_mat[-nrow(eu_mat),]/eu_mat[-1,])
head(log_returns)
```

```
##               DAX          SMI          CAC         FTSE
## [1,]  0.009326550 -0.006178360  0.012658756 -0.006770286
## [2,]  0.004422175  0.005880448  0.018740638  0.004889587
## [3,] -0.009003794 -0.003271184  0.005779182 -0.009027020
## [4,]  0.001778217 -0.001483372 -0.008743353 -0.005771847
## [5,]  0.004676712  0.008933417  0.005120160  0.007230164
## [6,] -0.012427042 -0.006737244 -0.011714353 -0.008517217
```

```r
#Get the covariance matrix and annualize
cov_mat<-cov(log_returns)*252
cov_mat
```

```
##             DAX        SMI        CAC       FTSE
## DAX  0.02673902 0.01688290 0.02102973 0.01320932
## SMI  0.01688290 0.02156192 0.01584042 0.01084738
## CAC  0.02102973 0.01584042 0.03066341 0.01434680
## FTSE 0.01320932 0.01084738 0.01434680 0.01595801
```

```r
#Get the Cholesky decomposition lower triangle
chol_mat<-t(chol(cov_mat))
chol_mat
```

```
##             DAX        SMI        CAC       FTSE
## DAX  0.16352071 0.00000000 0.00000000 0.00000000
## SMI  0.10324625 0.10441326 0.00000000 0.00000000
## CAC  0.12860589 0.02454040 0.11628287 0.00000000
## FTSE 0.08078073 0.02401107 0.02896971 0.08953607
```

```r
#Let's make a check.
#Diagonal values should give volatilities
#So annualized standard deviation of DAX log-returns should
#give chol_mat[1,1]
sd(log_returns[,"DAX"])*sqrt(252)
```

```
## [1] 0.1635207
```

```r
#OK. Let's continue.
#Assume we would like to simulate their positions 126 trading days later.
#Approximately half a year. Remember we annualized the returns.
#Assume mu is zero.
S_0<-eu_mat[1,]
S_0
```

```
##     DAX     SMI     CAC    FTSE
## 1628.75 1678.10 1772.80 2443.60
```

```r
S_T <- S_0*exp(0 + chol_mat%*%rnorm(ncol(eu_mat))*sqrt(126/252))
S_T
```

```
##          [,1]
## DAX  1647.704
## SMI  1800.364
## CAC  1696.818
## FTSE 2240.303
```

Variance Reduction

Monte Carlo simulation require a lot of instances to come up with a good approximation to the true value. Check the following example of a coin toss.

```r
#Let's calculate the probability of getting a Heads
#First generate a uniform random vector with 10^4 instances
unif_vec<-runif(10^4)
#If unif_vec < 0.5 let's say heads and 1
```

```r
h_or_t<-ifelse(unif_vec<0.5,1,0)
head(h_or_t)
```

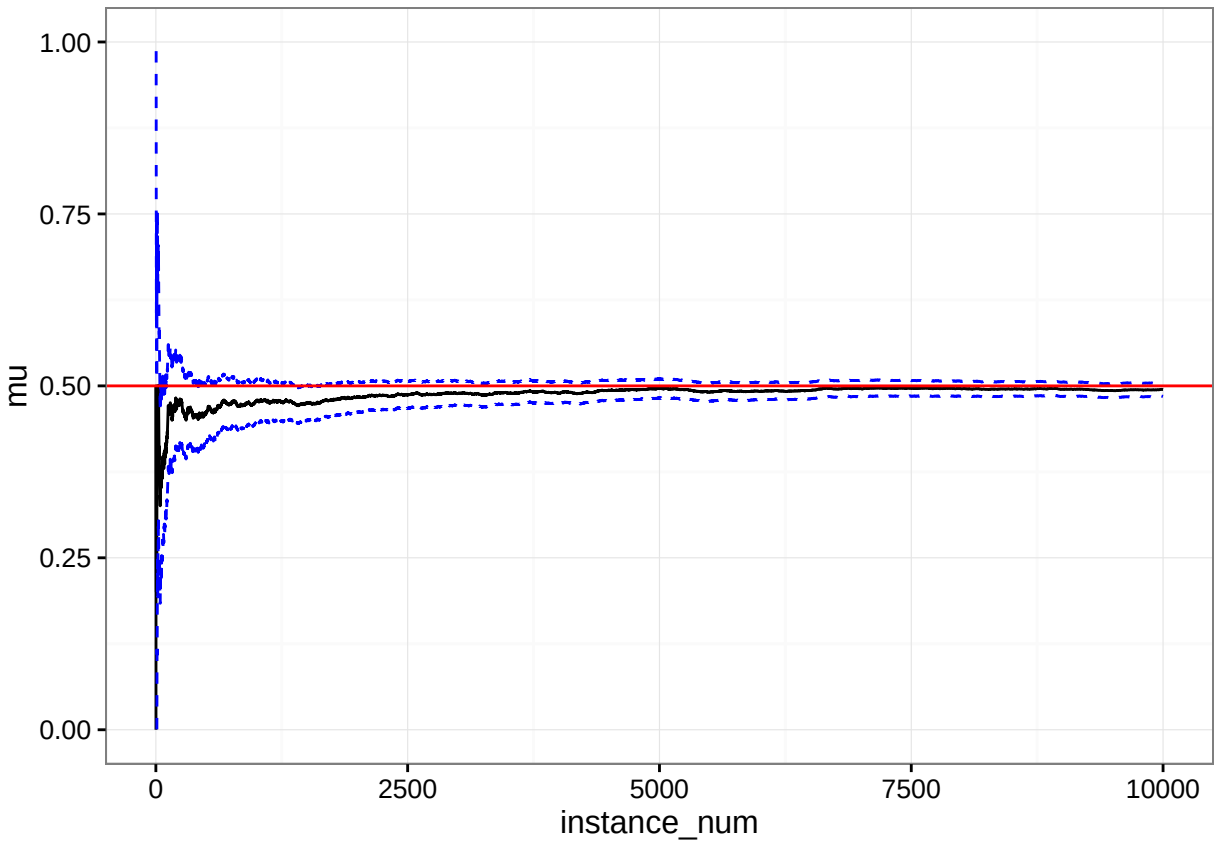
```
## [1] 0 1 0 0 0 1
```

```r
mc_process<-data.frame(instance_num=1,
                       mu=mean(h_or_t[1]),
                       upper=mean(h_or_t[1])+1.96*sd(h_or_t[1])/sqrt(1),
                       lower=mean(h_or_t[1])-1.96*sd(h_or_t[1])/sqrt(1))
for(i in 2:length(unif_vec)){
    mc_process[i,]<-data.frame(instance_num=i,
                           mu=mean(h_or_t[1:i]),
                           upper=mean(h_or_t[1:i])+1.96*sd(h_or_t[1:i])/sqrt(i),
                           lower=mean(h_or_t[1:i])-1.96*sd(h_or_t[1:i])/sqrt(i))

}
head(mc_process)
```

```
##   instance_num        mu     upper       lower
## 1            1 0.0000000        NA          NA
## 2            2 0.5000000 1.4800000 -0.48000000
## 3            3 0.3333333 0.9866667 -0.32000000
## 4            4 0.2500000 0.7400000 -0.24000000
## 5            5 0.2000000 0.5920000 -0.19200000
## 6            6 0.3333333 0.7465376 -0.07987095
```

```r
#Let's plot the progress of the bounds and probability estimate
ggplot(data=mc_process) + geom_line(aes(x=instance_num,y=mu)) +
geom_line(aes(x=instance_num,y=lower),color="blue",linetype=2) +
geom_line(aes(x=instance_num,y=upper),color="blue",linetype=2) +
geom_hline(yintercept=0.5,color="red") + ylim(c(0,1)) + theme_bw()
```



In the graph above blue lines denote the confidence interval of the simulation, black line denotes the estimate of the simulation and red line denotes the true probability. It can be seen that confidence interval converges to the true probability with diminishing returns.

We aim to increase the speed of conversion by reducing the number of instances that yield similar convergence.

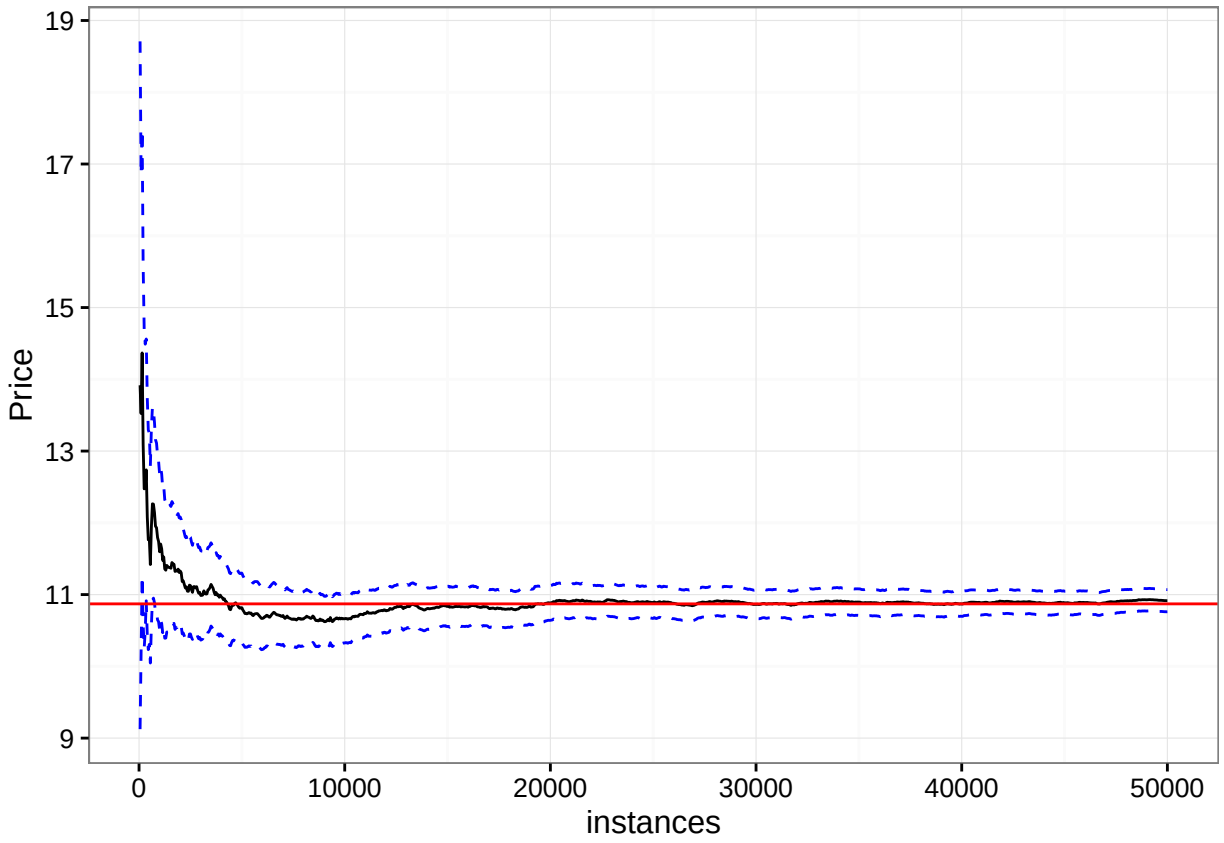
Let's recall our naive European Call Simulation.

```
#This is a similar setting to the above calculations for plotting
#We just restart the seed at each calculation
set.seed(522)
bs_simul <- data.frame(instances=50,
    t(sim_european_call(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=50)))
for(i in 2:1000){
    set.seed(522)
    bs_simul <- rbind(bs_simul,
    data.frame(instances=50*i,
        t(sim_european_call(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=50*i))))
}
#Let's calculate the true price according to BS formula
bs_price<-
black_scholes_eopt(s0=100,K=100,r=0.02,T_in_days=252,sig=0.25,callOrPut="call")
bs_price
```

```
## [1] 10.87056
```

```
#Let's plot the progress of the bounds and price estimate
ggplot(data=bs_simul) + geom_line(aes(x=instances,y=Price)) +
geom_line(aes(x=instances,y=Lower),color="blue",linetype=2) +
geom_line(aes(x=instances,y=Upper),color="blue",linetype=2) +
```

```
geom_hline(yintercept=bs_price,color="red") +
ylim(c(min(bs_simul$Lower),max(bs_simul$Upper))) + theme_bw()
```



Antithetic Variates

The logic behind the antithetic variates is the calculation of the variance. Suppose we want to calculate the mean value of a process $\mu = E[X]$. We generate two sample sets X_1 and X_2 and $\hat{\mu} = (E[X_1] + E[X_2])/2$ is an unbiased estimate of the μ. Also, the variance of the estimate is

$$V(\hat{\mu}) = \frac{V(X_1) + V(X_2) + 2Cov(X_1, X_2)}{4}$$

It is possible reduce the variance of the estimate by simply making the covariance negative $Cov(X_1, X_2) < 0$. Generating negatively correlated random variates is straightforward.

- For uniform generated variates $U = U(0, 1)$; use $V = 1 - U$.
- For normal generated variates $Z = N(0, 1)$; use $V = -Z$.
- For other distributions who are suitable for inversion, refer to inversion method $F^{-1}(1 - U)$ where $F^{-1}(X)$ is the inverse CDF of the distribution.

Let's try it in our coin toss example with a slight difference. Assume the true probability of getting a heads is 0.3.

```
n<-1000 #number of instances
#Naive calculation
h_or_t_naive <- runif(n) < 0.3
mu=mean(h_or_t_naive)
```

```r
SE=1.96*sd(h_or_t_naive)/sqrt(n)
print(c(Mean=mu,SE=SE,Lower=mu-SE,Upper=mu+SE))
```

```
##       Mean         SE      Lower      Upper
## 0.30200000 0.02847112 0.27352888 0.33047112
```

```r
#Antithetic variates
unif_vec<-runif(n/2)
#Normal process
x_1<- unif_vec < 0.3
#Antithetic process
x_2<- (1-unif_vec) < 0.3
h_or_t_av <- (x_1 + x_2)/2
mu=mean(h_or_t_av)
SE=1.96*sd(h_or_t_av)/sqrt(n/2)
print(c(Mean=mu,SE=SE,Lower=mu-SE,Upper=mu+SE))
```

```
##       Mean         SE      Lower      Upper
## 0.30600000 0.02137801 0.28462199 0.32737801
```

To benchmark, AV yields a lower standard error with half the number of instances. For the Call option with Black Scholes

```r
#Let's build a function to simulate
sim_european_call_av<-function(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=10^4){
    z_1 <- rnorm(n/2)
    z_2 <- -z_1
    #Simulate the payoffs with both processes
    sim_payoff_1<-exp(-r*T)*pmax(S_0*exp((r-0.5*vol^2)*T_years + vol*z_1*sqrt(T))-K,0)
    sim_payoff_2<-exp(-r*T)*pmax(S_0*exp((r-0.5*vol^2)*T_years + vol*z_2*sqrt(T))-K,0)
    sim_payoff <- (sim_payoff_1 + sim_payoff_2)/2
    #Calculate results and bounds
    Price<-mean(sim_payoff)
    SE<-1.96*sd(sim_payoff)/sqrt(n/2)
    LowerB <- Price - SE
    UpperB <- Price + SE
    return(c(Price=Price,SE=SE,Lower=LowerB,Upper=UpperB))
}
##Let's compare
#Naive method
sim_european_call(n=10^4)
```

```
##      Price         SE      Lower      Upper
## 11.1219161  0.3475531 10.7743631 11.4694692
```

```r
#Antithetic Variates
sim_european_call_av(n=10^4)
```

```
##      Price         SE      Lower      Upper
## 10.7899287  0.2688067 10.5211220 11.0587354
```

```r
#BS Value
black_scholes_eopt(s0=100,K=100,r=0.02,T_in_days=252,sig=0.25,callOrPut="call")
```

```
## [1] 10.87056
```

Control Variates

Control variates exploit the inclusion of an additional factor that is correlated with the actual process. Here is how a process $Z = X$ can be re-written as $Z = X + \theta(E[Y] - Y)$ where Y is a control variate and θ is a number. You can see the expectation estimate is unbiased and variance is reduced depending on θ.

$$E[Z] = E[X] + \theta(E[Y] - E[Y]) = E[X]$$

$$V(Z_\theta) = V(X - \theta Y) = V(X) - 2\theta Cov(X, Y) + \theta^2 V(Y)$$

If we take the derivative with respect to θ to find the optimal value (θ^*) to minimize the variance the result is as follows.

$$\theta^* = \frac{Cov(X, Y)}{V(Y)}$$

We can use internal CV or external CV. In internal CV we use part of the simulated process and in external CV we use another similar process that we know the result of.

For the European Call we can use the final stock price estimate S_T as a CV.

```
sim_european_call_cv<-function(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,n=10^4,with_naive=TRUE){
    #Simulate S_T values
    S_T_est <-S_0*exp((r-0.5*vol^2)*T_years + vol*rnorm(n)*sqrt(T))
    #Simulate the payoffs with both processes
    sim_payoff<-exp(-r*T)*pmax(S_T_est-K,0)
    #Calculate theta_star
    theta_star<-cov(S_T_est,sim_payoff)/var(S_T_est)
    #Calculate CV effect
    payoff_cv <- sim_payoff - theta_star*(S_T_est - S_0*exp(r*T_years))
    #Calculate the output of naive simulation as well
    if(with_naive){
        Price<-mean(sim_payoff)
        SE<-1.96*sd(sim_payoff)/sqrt(n)
        LowerB <- Price - SE
        UpperB <- Price + SE
        print(c(Price_naive=Price,SE_naive=SE,Lower=LowerB,Upper=UpperB))
    }
    Price<-mean(payoff_cv)
    SE<-1.96*sd(payoff_cv)/sqrt(n)
    LowerB <- Price - SE
    UpperB <- Price + SE
    return(c(Price_CV=Price,SE_CV=SE,Lower=LowerB,Upper=UpperB))
}
```

Common Random Numbers

Common Random Numbers variance reduction technique is used in finding differences between two different processes by employing positively correlated random variates in those processes. Using the same random variate sequence for both processes has the ultimate correlation.

Suppose we want to compare the differences between two option prices with $r = 0.01$ and $r = 0.06$.

```
#Same EC function that returns payoff vector
payoff_EC<-function(S_0=100,K=100,vol=0.25,T_years=1,r=0.02,z_val){
    return(pmax(S_0*exp((r-0.5*vol^2)*T_years + vol*z_val*sqrt(T))-K,0)*exp(-r*T_years))
}
```

```r
n<-10^4 #Number of instances
#Naive method.
payoff_1<-payoff_EC(r=0.01,z_val=rnorm(n))
payoff_2<-payoff_EC(r=0.06,z_val=rnorm(n))
diff_vec<-payoff_1 - payoff_2
mu_diff<-mean(diff_vec)
SE=1.96*sd(diff_vec)/sqrt(n)
print(c(Diff=mu_diff,SE=SE,Lower=mu_diff-SE,Upper=mu_diff+SE))
```

```
##       Diff         SE      Lower      Upper
## -2.6566057  0.5087156 -3.1653213 -2.1478901
```

```r
#CRN Method
z_val<-rnorm(n)
payoff_1<-payoff_EC(r=0.01,z_val=z_val)
payoff_2<-payoff_EC(r=0.06,z_val=z_val)
diff_vec<-payoff_1 - payoff_2
mu_diff<-mean(diff_vec)
SE=1.96*sd(diff_vec)/sqrt(n)
print(c(Diff=mu_diff,SE=SE,Lower=mu_diff-SE,Upper=mu_diff+SE))
```

```
##        Diff          SE       Lower       Upper
## -2.47149545  0.04597101 -2.51746646 -2.42552445
```

Modeling Credit Risk

Lender institutions (i.e. Banks) should calculate the risk of default by their obligors (i.e. borrower). A basic representation of this loss can be formulated as follows.

$$Loss = (EAD)x(LGD)x1_{\{D\}}$$

$$1_{\{D\}} = Bernoulli(PD)$$

where EAD is the exposure at default, LGD is loss given default as a fraction of EAD and PD is the probability of default.

Total portfolio loss can be calculated as follows.

$$Loss_n = \sum_{i=1}^{n}(EAD_i)x(LGD_i)x1_{\{D_i\}} = \sum_{i=1}^{n} Loss_i x1_{\{D_i\}}$$

Also

- Expected loss (EL) is defined as $EL := E[Loss_n]$
- Unexpected loss (UL) is defined as standard deviation around EL; $UL := \sqrt{V[Loss_n]}$
- Value-at-Risk (VaR_α) is defined as the minimal exposure to loss given an $1-\alpha$ probability. It is calculated as the quantile value of the given distribution at α; $VaR_\alpha := F^{-1}(\alpha)$.
- Expected Shortfall (ES_α) is the expected loss given the α threshold is exceeded; $ES := E[Loss_n | Loss_n \geq VaR_\alpha]$.
- Economic Capital (EC_α) is defined as $EC_\alpha := VaR_\alpha - EL$

It is highly possible that default probabilities (PD) are correlated in some form, so total portfolio loss is not straightforward to estimate. For instance suppose X_i is a standard normal variate and C_i is a default threshold point. Obligor defaults if $X_i < C_i$.

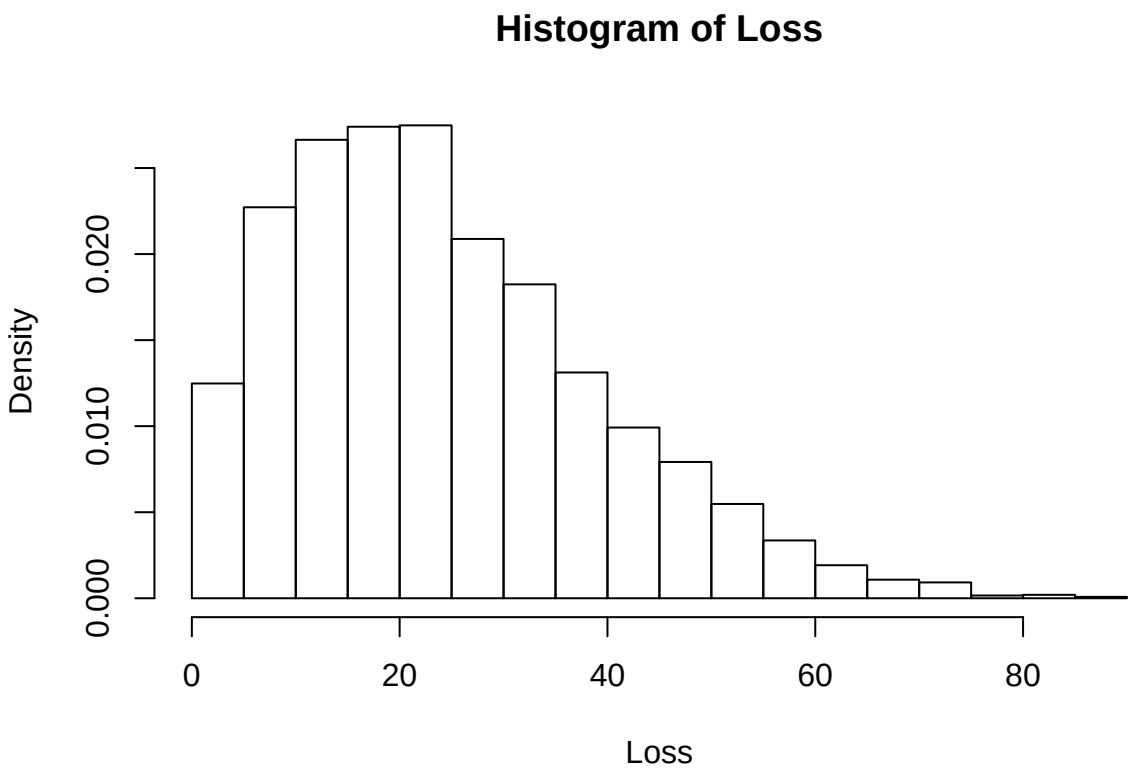
$$PD_i = P\{X_i < C_i\} = \Phi(C_i)$$
$$C_i = \Phi^{-1}(PD_i)$$

We can consider X_i in two parts. General economic factors (Z) affecting obligor's state and obligor's individual state (Y_i).

$$X_i = \sqrt{\rho} * Z + \sqrt{1-\rho^2} Y_i$$

```
lossrealization<-function (n,PD,EAD,LGD,rho){
    # Keep track of the loss in this portfolio .
    totalloss <- 0
    # Draw a normal random variable for the # systematic factor .
    sf <- rnorm(1)
    # Loop through all obligors to see if they # go into default .
    for(obligor in 1:n){
        # Draw specific factor .
        of <- rnorm(1)
        # Asset value for this obligor .
        x <- sqrt(rho)*sf + sqrt(1-rho)*of
        # Critical threshold for this obligor .
        c <- qnorm(PD[obligor])
        # check for default .
        if(x < c){
            totalloss <- totalloss + EAD[obligor] * LGD[obligor];
        }
    }
    return(totalloss)
}

n <- 100 # The number of obligors in the portfolio.
runs <- 5000 # Number of realizations.
# Run a number of realizations .
EAD <- rep (1,n)
LGD <- rep (1,n)
PD <- rep (0.25,n)
rho <- 0.2
losses <- c()
for(run in 1:runs){
    # Add a new realiza/on of the loss variable .
    losses <- c(losses,lossrealization(n,PD,EAD,LGD,rho))
}
# Output : normalised histogram .
hist(losses, freq =FALSE , main =" Histogram of Loss ", xlab =" Loss ", ylab =" Density ");
```




```
alpha <- 0.95 # Alpha level
# Sort losses and select right value
losses <- sort(losses)
j <- floor(alpha*runs)
var_value <- losses[j]
# Select the losses that are larger than VaR
largelosses <- losses[losses >= var_value]
# Output TCE
ES <- mean (largelosses)
```