Financial Engineering HW2

Group 2: Natalie Frantsits, Lara Hofmann, Eldar Kodzov, Chiara Lesa, Vedad Sehanovic

March 2022

Contents

Ex 1

a

Suppose we have process X following geometric Brownian motion dynamics with drift

$$\mu = 0.1,$$

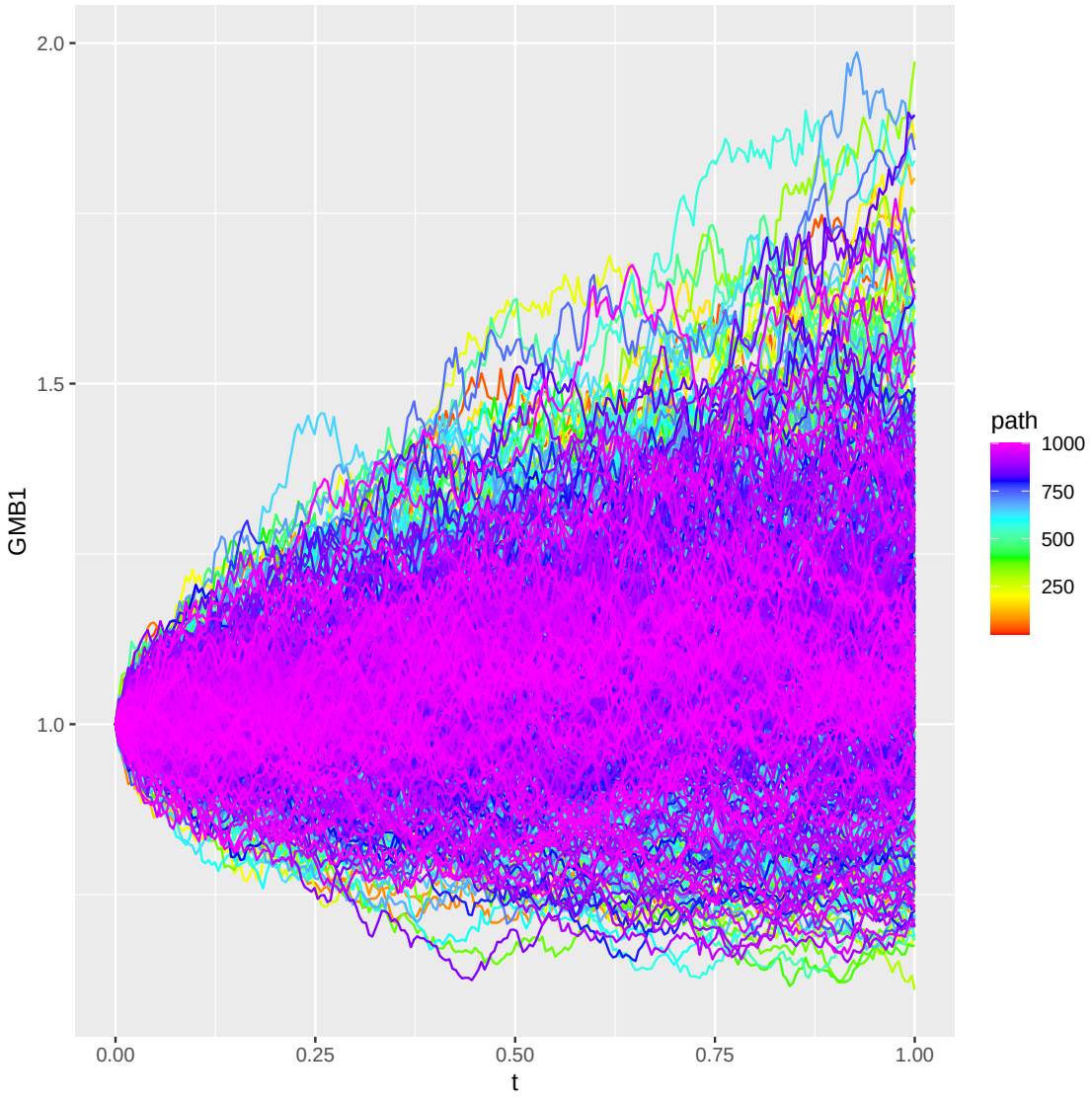
$$\sigma = 0.2.$$

By using the random walk approximation as well as the Cholesky decomposition to generate Brownian motion, simulate 1000 paths for the time interval $[0, 1]$. In your computations use $n = 25$ equidistant time discritization points with $t_0 = 0$ and $t_n = T = 1$. Compare the computation times corresponding to the two method. Repeat this exercise for $\mu = 0.4$, $\sigma = 0.5$ (keep the seed fixed in order to see the impact of different parameters). For all cases provide a plot with the simulated paths.

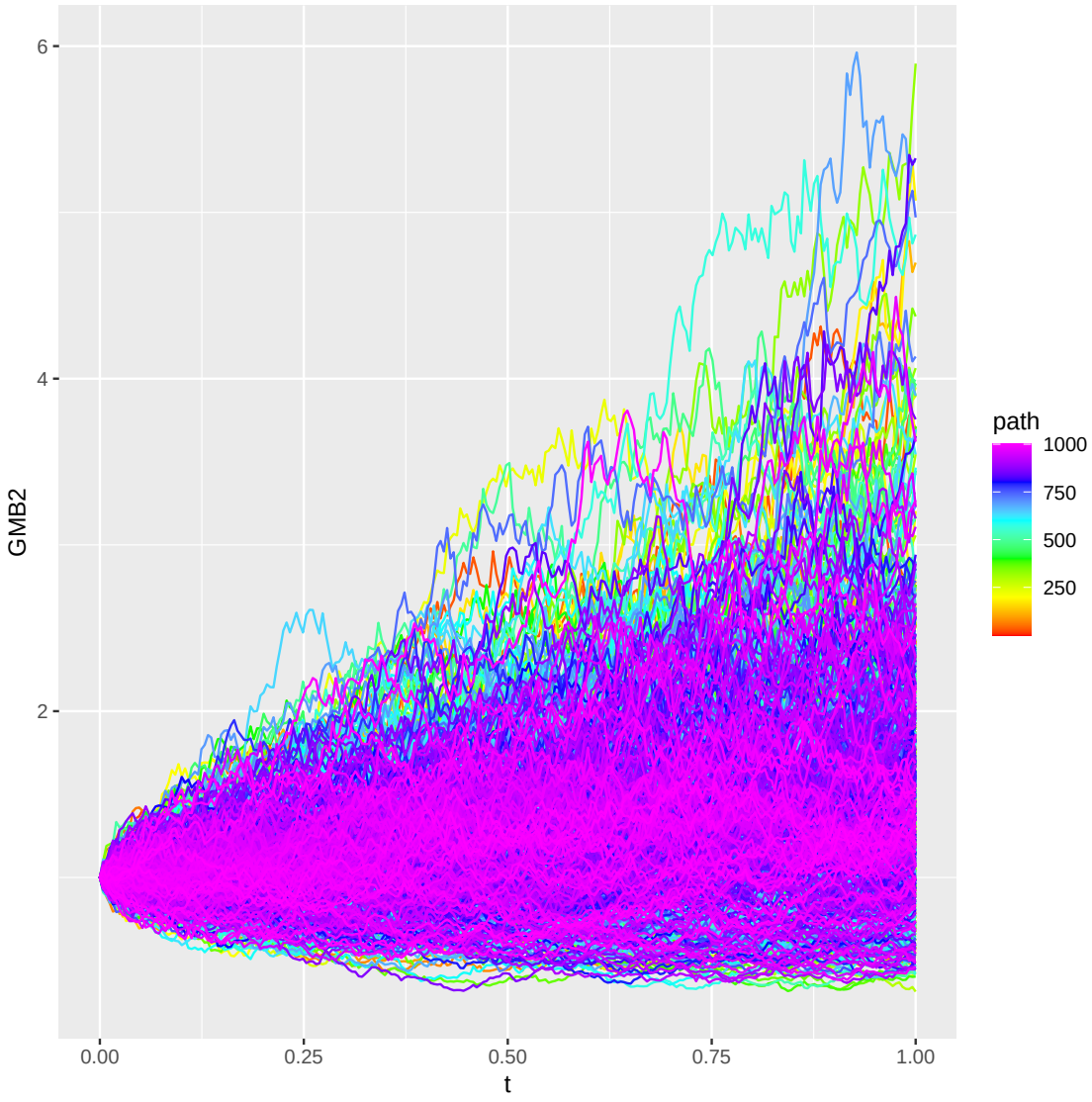
Random Walk Method

```
library(ggplot2)
#mu =0.1, mu = 0.4
#sigma = 0.2, sigma = 0.5
#1. random walk approximation
#2. cholesky decomposition
#simulate 1000 paths in [0,1], n = 250 equidistant time discritization points with t0 = 0, tn=T=1
#w_t+1 = w_t + sqrt(t_1+1 - t_i)*Z_i+1
set.seed(123)

t = seq(from = 0, to = 1, length.out = 250) #create time discretization vector

W_0 <- 0

time_step <- data.frame()

start_time <- Sys.time()#to count how long does it take

n.paths <- 1000
start_time <- Sys.time()

for ( i in 1:n.paths){#simulate 1000 paths

  W <-  c(W_0,numeric(250-1)) #250 - 1 because the first 1 is W_0 = 0 in t=0

  for(j in 2:length(t)){ #first entry of t is t0

    W[j] = W[j-1] + sqrt(t[j] - t[j-1])*rnorm(1)

    }

  time_step<- rbind(time_step, cbind(t,W))

}
end_time<- Sys.time()
comput_time <- end_time - start_time
```

```
#Time difference of 8.84635 secs

time_step$path <-rep(1:1000, each=250)
head(time_step)

##             t           W path
## 1 0.000000000  0.00000000    1
## 2 0.004016064 -0.03551870    1
## 3 0.008032129 -0.05010561    1
## 4 0.012048193  0.04867352    1
## 5 0.016064257  0.05314181    1
## 6 0.020080321  0.06133508    1

#generate the two GMB processes
time_step$GMB1 <- exp((0.1 - 0.2^2/2) * t + 0.2* time_step$W)
time_step$GMB2 <- exp((0.4 - 0.5^2/2) * t + 0.5* time_step$W)

p1<- ggplot(data=
              time_step[1:(nrow(time_step)),])+
              geom_line(aes(x = t,
                            y = GMB1,
                            group = path,
                            col = path))+
              scale_color_gradientn(colours = rainbow(6))

p2 <- ggplot(data=time_step[1:nrow(time_step),]) +
  geom_line(aes(x = t,
                y = GMB2,
                group = path,
                col = path))+
  scale_color_gradientn(colours = rainbow(6))
p1
```



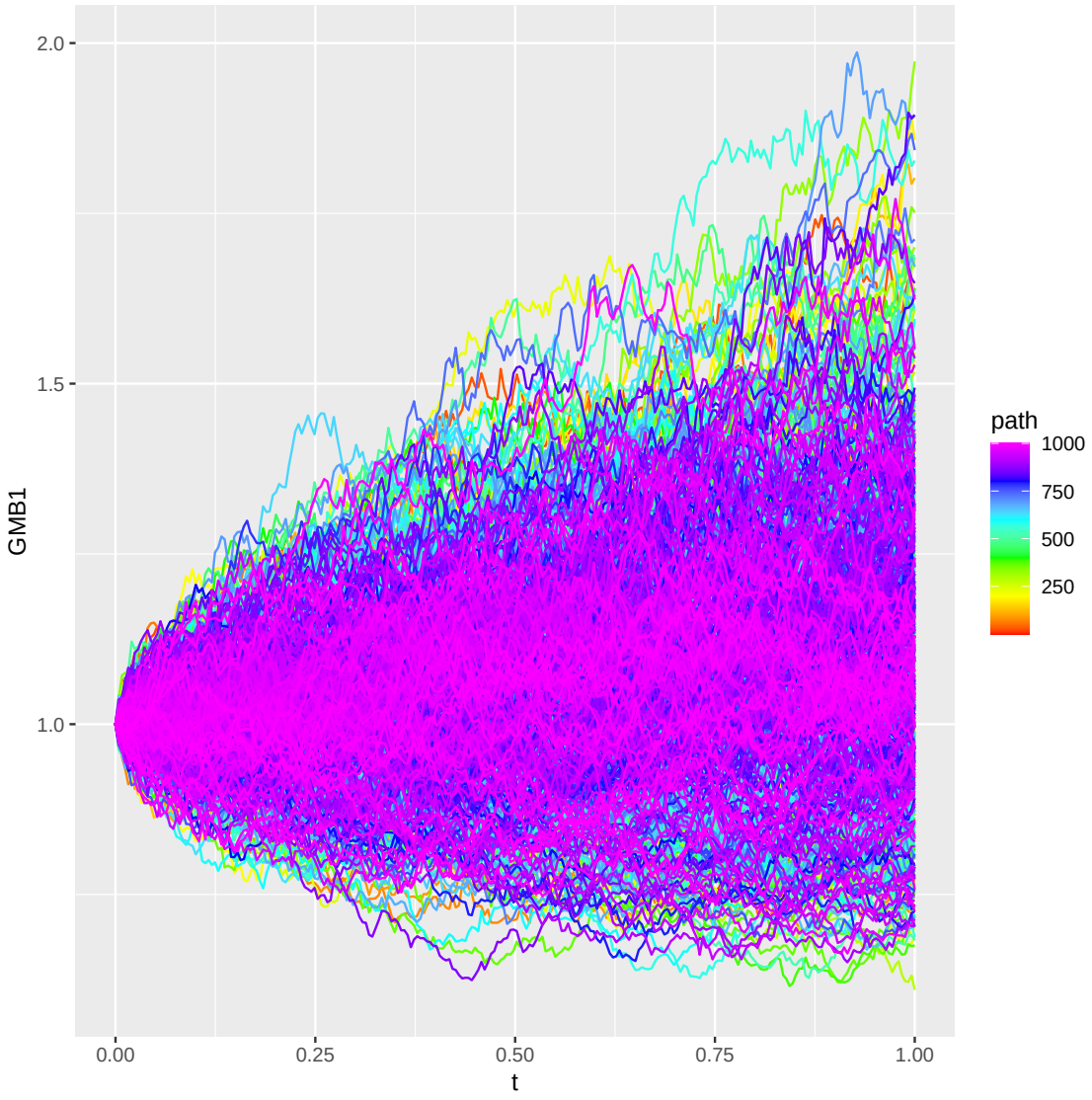

```
p2
```



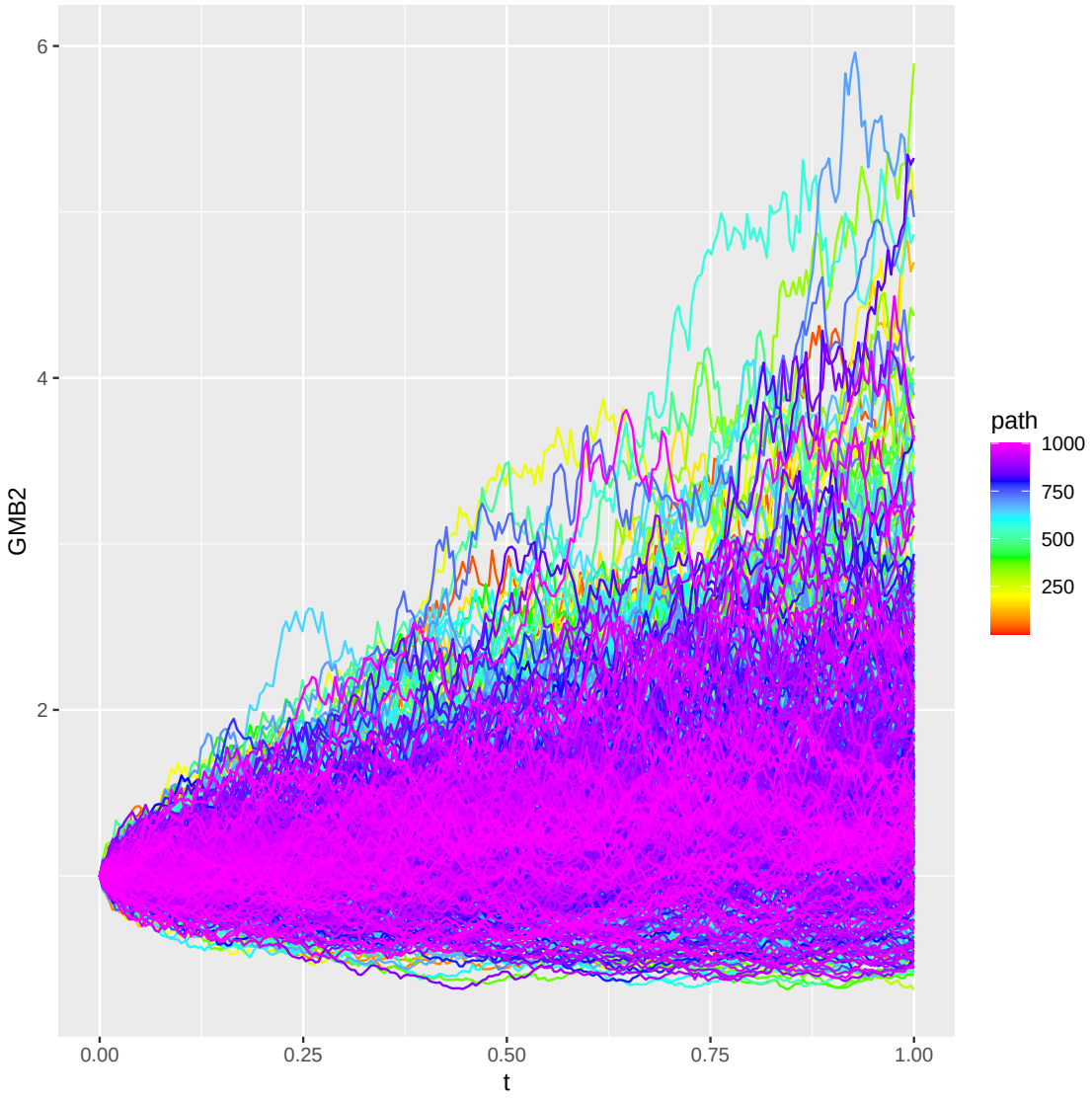
Cholesky Decomposition Method

```
C <- outer(t[2:250],t[2:250],FUN="pmin")#returns the parallel maxima and minima of the input values.
A <- rbind(0,cbind(0,t(chol(C))))
dim(A)# 250 250

## [1] 250 250

time_step2 <- data.frame()
set.seed(123)


start_time <- Sys.time()
for ( i in 1:1000){#simulate 1000 paths

  z <-  rnorm(250) #250 because we need Z = (Z_1, ... ,Z_n)
```

```
  AZ <- A %*% z

  time_step2<- rbind(time_step2, cbind(t,AZ))

}
end_time<- Sys.time()
comput_time <- end_time - start_time
#Time difference of 7.552732 secs
time_step2$path <-  rep(1:1000, each=250)
colnames(time_step2) <- c("t", "W", "path")
time_step2$GMB1 = exp((0.1 - 0.2^2/2) * t + 0.2* time_step$W)
time_step2$GMB2 = exp((0.4 - 0.5^2/2) * t + 0.5* time_step$W)

pchol1<- ggplot(data=
              time_step2[1:(nrow(time_step2)),])+
  geom_line(aes(x = t,
                y = GMB1,
                group = path,
                col = path))+
  scale_color_gradientn(colours = rainbow(6))

pchol2 <- ggplot(data=time_step2[1:nrow(time_step2),]) +
  geom_line(aes(x = t,
                y = GMB2,
                group = path,
                col = path))+
  scale_color_gradientn(colours = rainbow(6))
pchol1
```




```
pchol2
```



b

```
set.seed(100)

poisson_sim <- function(run=1, tt, lambda){
  # this is a function for only one path
  # "df" is shaped in accordance with the need of plotting the process paths:
  # "I" is implemented as in the slides
  # "run" and "N" are auxiliary columns that will be used in plotting

  ta <- 0
  I <- 0
  U <- runif(1, 0, 1)
  ta <- ta - log(U)/lambda
  S <- c()
```

```
  while (ta < tt){
    I <- I+1
    S <- append(S,ta)
    ta <- ta-log(runif(1))/lambda
  }

  df <- rbind(c(0), data.frame(S))
  df$I <- I
  df$N <- ave(df$S, FUN = seq_along)
  df$run = rep(run, length(S)+1)

  return(df)
}
df1 <- poisson_sim(1, 5, 1)
df1$N <- df1$N-1
df1
```

```
##          S I N run
## 1 0.000000 3 0   1
## 2 1.178415 3 1   1
## 3 2.534481 3 2   1
## 4 3.128104 3 3   1
```

```
df2 <- poisson_sim(1, 5, 0.02) # no jumps is not surprising - refer to text below
df2
```

```
##   X0 I run
## 1  0 0   1
```

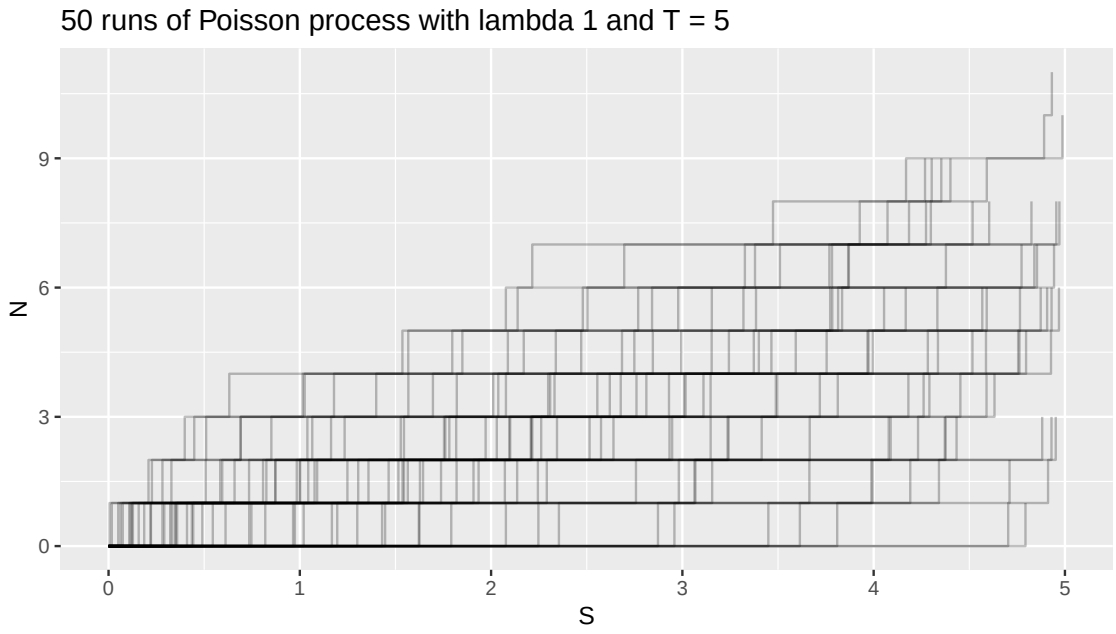
In case when lambda is small and the time T short, it is more likely that no jumps will occur in time $\leq T$. Hence the simulation yields only a few jump events in case of the second set of the chosen parameters. This is also reflected in the plot of process paths below.

```
# simulate on 50 runs

simulate <- function(runs, tmax, lambda)
{
  aa <- lapply(seq(runs), poisson_sim, tmax, lambda)
  df <- do.call(rbind, aa[which(lapply(aa, length) > 3)])
  df$N <- df$N-1
  df

}

# Given the lengthy output on 50 simulations, only head() will be shown.
# For full output, remove comment and run the line below:
# simulate(runs = 50, tmax = 5, lambda = 1)

head(simulate(runs = 50, tmax = 5, lambda = 1), 30)
```

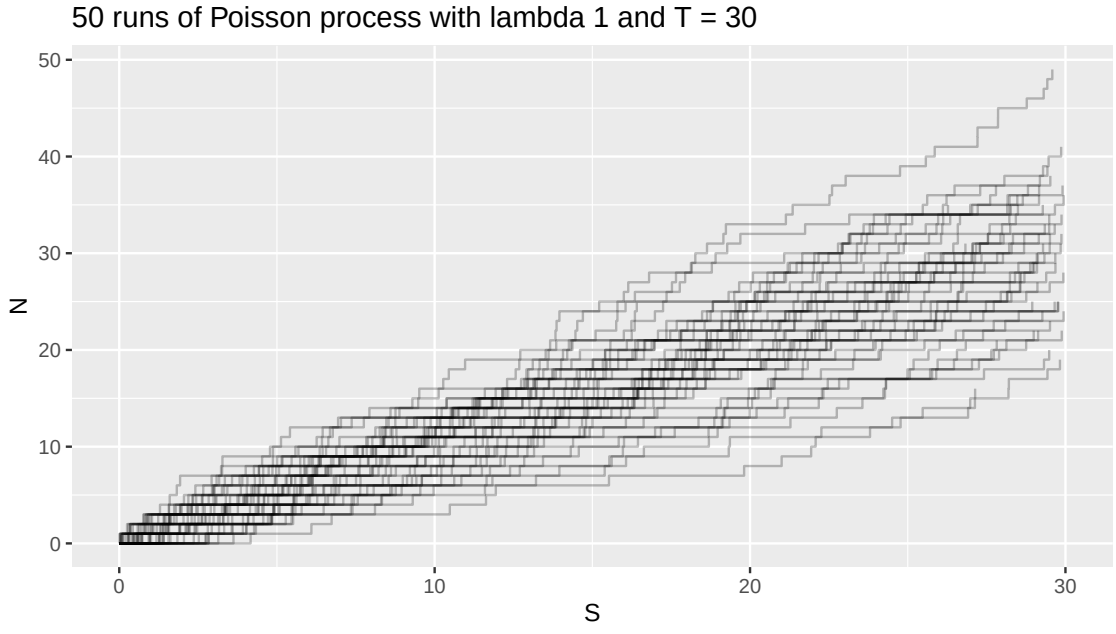
```
##            S I N run
## 1  0.0000000 7 0   1
## 2  0.7261442 7 1   1
## 3  0.9339034 7 2   1
```

```
## 4  1.9272897 7 3   1
## 5  2.5314035 7 4   1
## 6  4.3018200 7 5   1
## 7  4.7718293 7 6   1
## 8  4.8972049 7 7   1
## 9  0.0000000 5 0   2
## 10 0.9200781 5 1   2
## 11 1.1911639 5 2   2
## 12 1.5931027 5 3   2
## 13 3.1797417 5 4   2
## 14 4.2082921 5 5   2
## 15 0.0000000 7 0   3
## 16 0.3706427 7 1   3
## 17 0.9946162 7 2   3
## 18 1.3359750 7 3   3
## 19 1.9552238 7 4   3
## 20 2.2442772 7 5   3
## 21 3.1115362 7 6   3
## 22 4.8751736 7 7   3
## 23 0.0000000 9 0   4
## 24 0.1256158 9 1   4
## 25 0.7250965 9 2   4
## 26 2.0062249 9 3   4
## 27 2.7230379 9 4   4
## 28 2.7972173 9 5   4
## 29 3.8507837 9 6   4
## 30 3.8977100 9 7   4

# Simulate will only return those runs which do contain a jump.
# For lambda = 0.02, there will be only a few:
simulate(runs = 50, tmax = 5, lambda = 0.02)

##            S I N run
## 1  0.0000000 1 0   1
## 2  0.3613422 1 1   1
## 3  0.0000000 1 0   2
## 4  0.1998361 1 1   2
## 5  0.0000000 1 0   5
## 6  4.6256432 1 1   5
## 7  0.0000000 1 0  22
## 8  2.1400194 1 1  22
## 9  0.0000000 1 0  32
## 10 0.7585508 1 1  32
## 11 0.0000000 1 0  42
## 12 1.9112467 1 1  42

# Plot

plot_poisson <- function(runs, tmax, lambda)
{
  aa <- lapply(seq(runs), poisson_sim, tmax, lambda)
  df <- do.call(rbind, aa[which(lapply(aa, length) > 3)])
  df$N <- df$N-1
```

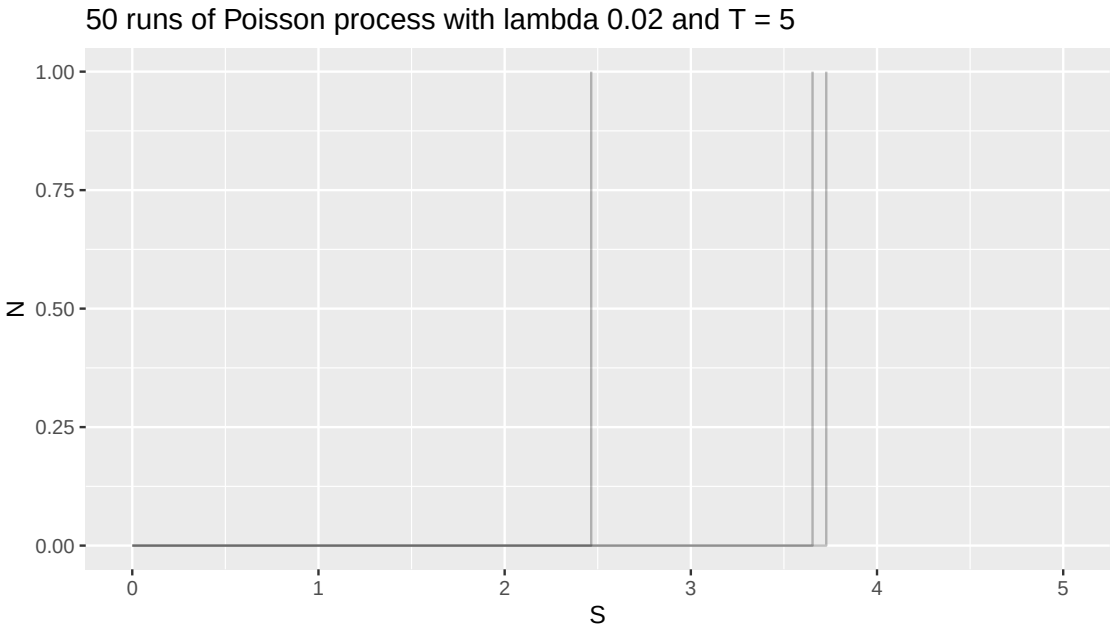
```
  ggplot2::ggplot(df, aes(S, N, group = run)) +
    geom_step(alpha = 0.25) +
    labs( title = paste(runs, "runs of Poisson process with lambda", lambda, "and T =", tmax)) +
    theme(legend.position = "none") +
    coord_cartesian(xlim = c(0, tmax))
}

plot_poisson(runs = 50, tmax = 5, lambda = 1)
```




```
#This plot is run to show paths on a longer time interval:
plot_poisson(runs = 50, tmax = 30, lambda = 1) #stylized view on bigger T
```



```r
plot_poisson(runs = 50, tmax = 5, lambda = 0.02) #shows only cases when there is a jump
```



Ex 2

Assume a Black-Scholes setting and suppose we want to price a European call option written on a stock with initial value $S_0 = 90$, $\sigma = 0.25$, $\mu = 0.3$. The maturity of the option is in $T = 1$ year and the strike price is $K = 100$. Assume that the risk-free interest rate is $r = 0.5\%$. Price the option analytically (Black-Scholes fromula) and numerically by using naive Monte Carlo (simulate $n = 10000$ paths). Compute the corresponding Monte Carlo standard error and confidence interval for $\alpha = 0.05$.
The price of a call option analytically computed is:

$$C = S_0\Phi(d1) - e^{-rT}K\Phi(d2)$$

with

$$d_1 = \frac{log(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

and

$$d_2 = d_1 - \sqrt{T}$$

.
The algorithm to compute the Monte Carlo estimate of a call option is:

1: Given inputs S_0, r,σ, K, T, n.
2: **for** $i = 1 : n$ **do**
3: Generate $Z_i \sim \mathcal{N}(0, 1)$
4: $S_i = S_0 e^{(r-\sigma^2/2)T+\sigma\sqrt{T}Z_i}$
5: $C = e^{-rT}(S_i - K)^+$
6: **end for**
7: $\hat{C_n} = \frac{1}{n}\sum_{i=1}^{N} C[i]$
8: Return $\hat{C_n}$.

The Variance of the Monte Carlo Error, since we know the true analythical value of the Call, is:

$$\mathbb{V}(MCE_n) = \mathbb{V}(\hat{\Theta}_n - \Theta) = \sigma^2/n$$

Taking the square root of σ^2/n we obtain the standard error $\sigma/\sqrt{n}$. To compute the confidence interval for Θ we proceed by taking the $(1 - \alpha/2)$ percentile of the standard normal distribution.

$$\Theta \in \hat{\Theta_n} \pm \frac{z_\alpha\sigma}{\sqrt{n}}$$

.

```
set.seed(1000)
S0 = 90
sigma = 0.25
mu= 0.3
Tmax = 1
K = 100
r = 0.5/100
n = 10000
call <-function(S0, sigma, mu, Tmax, r){
    d1<- (log(S0/K) + (r + sigma^2/2)*Tmax)/(sigma*sqrt(Tmax))
    d2<- d1 - (sigma*sqrt(Tmax))
    #analytical price for call option
    C <- S0*pnorm(d1) + (-exp(-r*Tmax)*pnorm(d2)*K)
    return(C)
}
```

```r
call_value <- call(S0, sigma, mu, Tmax, r)
# [1] 5.419602
#now with MC
set.seed(123)
MC_call <- function(S0, sigma, mu, Tmax, r, n, alpha, true_call_val){

  sim_call <- function(S0, sigma, mu, Tmax, r){
  z  <- rnorm(1,0,1)
  St <- S0 * exp((r - sigma^2/2)*Tmax + sigma*sqrt(Tmax)*z)
  C  <- exp(-r*Tmax) * max(St - K,0)
  C
  }

  Call_MC <- mean(replicate(n, sim_call(S0, sigma, mu, Tmax, r)))
  simulations <-replicate(n, sim_call(S0, sigma, mu, Tmax, r))
  standard_error_MCE <- sd(simulations - true_call_val)/sqrt(n) #we know the call true value

  CI <- c(Call_MC - qnorm(1-alpha/2) * standard_error_MCE,
          Call_MC + qnorm(1-alpha/2) * standard_error_MCE)
  width_CI <-2*qnorm(1-alpha/2)*standard_error_MCE
  list_results <- list("Call Price" = true_call_val
                       , "MC Estimate Call Price" = Call_MC
                       , "Standard Error MCE" = standard_error_MCE
                       , "Confidence interval" = CI
                       , "Width Confidence Interval" = width_CI)
  return(list_results)
}

results <- MC_call(S0, sigma, mu, Tmax, r, n,
                   alpha = 0.05,
                   true_call_val= call_value)
results

## $`Call Price`
## [1] 5.419602
##
## $`MC Estimate Call Price`
## [1] 5.39059
##
## $`Standard Error MCE`
## [1] 0.1224315
##
## $`Confidence interval`
## [1] 5.150629 5.630551
##
## $`Width Confidence Interval`
## [1] 0.4799227
```

We can see that with the naive Monte Carlo approach, we already obtain a close approximation to the true call value given by the Black and Scholes formula. The standard error with a value of 0.1224 is also quite low, but could be further improved. For the 95% intervals we have a width of 0.4788, which can also be tightened by employing variance reduction methods as used in Ex 3.

Ex 3

Assume a Black-Scholes setting and suppose we want to price a European call option C written on a stock S with initial value $S_0 = 80$, $\sigma = 0.25$, $\mu = 0.3$. The maturity of the option is in $T = 1$ year and the strike price is $K = 80$. Assume that the risk-free interest rate is $r = 0.05\%$. Price the option with (Normal) antithetic variates (simulate $n = 10000$ paths). Estimate the reduction in the variance relative to naive Monte Carlo. Now take the strike value $K = 40$ (option is deep-in-the-money) and redo your computations (keep the seed fixed). How is the reduction in variance affected?

Due to the assumption of a Black-Scholes setting, we know that the log prices of the stock S are normally distributed, in particular:

$$\log(S_T) \sim \mathcal{N}\left(\log(S_0) + (\mu - \sigma^2/2)T, \sigma^2 T\right).$$

The price of the European call option at $t = 0$ is given by

$$C_0 = \mathbb{E}^{\mathcal{Q}}\left[e^{-rT}(S_T - K)^+\right],$$

where $\mathcal{Q}$ denotes the risk-neutral measure. Under $\mathcal{Q}$, the discounted stock price is a martingale and the drift of the stock price therefore changes to the risk-free rate r:

$$\log(S_T) \sim \mathcal{N}\left(\log(S_0) + (r - \sigma^2/2)T, \sigma^2 T\right).$$

Now, we use antithetic variates and naive Monte Carlo to compute the price C_0. In order to reduce the variance of the Monte Carlo estimate by using normal antithetic variates, the function h where $C_0 = \mathbb{E}(Y)$ and $Y = h(X_1, ..., X_n)$ for normal distributed X_i has to be monotonic. In this case, h is defined as follows:

$$h(x) = h_2(h_1(x)) \quad \text{where} \quad h_2(x) = e^{-rT}(x - K)^+ \quad \text{and} \quad h_1(x) = \exp(x).$$

Since both functions h_1 and h_2 are monotonic, also the function h is monotonic and we can apply the antithetic variates method.
To obtain two negatively correlated variables that are identically normally distributed, we use:

$$X \sim \mathcal{N}(\bar{\mu}, \sigma^2 T) \quad \text{and} \quad \bar{X} = 2\bar{\mu} - X \sim \mathcal{N}(\bar{\mu}, \sigma^2 T) \quad \text{where} \quad \bar{\mu} = \log(S_0) + (r - \sigma^2/2)T.$$

The algorithm to estimate C_0 works as follows:

1: Let $C = (0, ..., 0) \in \mathbb{R}^N$.
2: **for** $i = 1, .., N$ **do**
3: Generate $X \sim \mathcal{N}(\bar{\mu}, \sigma^2 T)$
4: $\bar{X} = 2\bar{\mu} - X$
5: $Z = e^{-rT}(\exp(X) - K)^+$
6: $\bar{Z} = e^{-rT}\left(\exp(\bar{X}) - K\right)^+$
7: $C[i] = 1/2(Z + \bar{Z})$
8: **end for**
9: $\hat{C}_0 = \frac{1}{N}\sum_{i=1}^{N} C[i]$
10: Return $\hat{C}_0 \in \mathbb{R}$.

We implement this algorithm in `R` and compare the reduction in the variance relative to the naive Monte Carlo. Note that if we use $n = 10000$ paths for the antithetic case, in order to obtain a reasonable comparison, we have to use $2n = 20000$ paths when using the naive Monte Carlo.

```r
BS_antithetic <- function(S0,sigma,mu,T,K,r,n){
  # n = number of simulation paths

  C <- rep(0,times=n)
  mu_bar <- log(S0) + (r-sigma^2/2)*T
  std_dev <- sigma*sqrt(T)
  for(i in 1:n){
    X <- rnorm(1, mu_bar, std_dev)
    X_bar <- 2*mu_bar - X
    Z <- exp(-r*T)*max(exp(X)-K,0)
    Z_bar <- exp(-r*T)*max(exp(X_bar)-K,0)
    C[i] <- 1/2*(Z + Z_bar)
  }
  C_hat <- mean(C)
  output <- list(C,C_hat)
  return(output)
}

BS_MC <- function(S0,sigma,mu,T,K,r,n){
  C <- rep(0,times=n)
  mu_bar <- log(S0) + (r-sigma^2/2)*T
  std_dev <- sigma*sqrt(T)
  for(i in 1:n){
    X <- rnorm(1, mu_bar, std_dev)
    C[i] <- exp(-r*T)*max(exp(X)-K,0)
  }
  C_hat <- mean(C)
  output <- list(C,C_hat)
  return(output)
}

S0 <- 80
sigma <- 0.25
mu <- 0.3
T <- 1
K <- 80
r <- 0.0005
n <- 10000

#now test variance
set.seed(1234)
result_antithetic <- BS_antithetic(S0,sigma,mu,T,K,r,n)
set.seed(1234)
result_MC <- BS_MC(S0,sigma,mu,T,K,r,2*n)

value_antithetic <- result_antithetic[[2]]
value_MC <- result_MC[[2]]
sd_antithetic <- sd(result_antithetic[[1]])/sqrt(n)
sd_MC <- sd(result_MC[[1]])/sqrt(2*n)

#95 percent confidence interval
alpha <- 0.05
z <- qnorm(1-alpha/2)
```

```
lowerbound_antithetic <- value_antithetic - z*sd_antithetic
upperbound_antithetic <- value_antithetic + z*sd_antithetic
width_antithetic <- 2*z*sd_antithetic

lowerbound_MC <- value_MC - z*sd_MC
upperbound_MC <- value_MC + z*sd_MC
width_MC <- 2*z*sd_MC

result_final1 <- matrix(c(value_antithetic, lowerbound_antithetic,
                          upperbound_antithetic, width_antithetic,
                          sd_antithetic, value_MC, lowerbound_MC,
                          upperbound_MC, width_MC, sd_MC),
                        byrow = FALSE, ncol = 2, nrow = 5)
colnames(result_final1) <- c("Antithetic K=80","MC K=80")
row.names(result_final1) <- c("Value","Lower Bound", "Upper Bound",
                              "Width", "Std. Dev.")
result_final1

##              Antithetic K=80    MC K=80
## Value             7.83002713 7.94324377
## Lower Bound       7.67989366 7.75634721
## Upper Bound       7.98016061 8.13014033
## Width             0.30026695 0.37379312
## Std. Dev.         0.07660012 0.09535714

abs(sd_antithetic-sd_MC) #variance reduction

## [1] 0.01875702
```

Comparing the results, we see that using the antithetic variables one obtains a smaller confidence interval for the estimate $\hat{C}_0$ and also a smaller standard deviation than with the naive Monte Carlo. In particular, the standard deviation was reduced by approximately 0.02.

Now, we repeat the procedure keeping the seed fixed but changing only the strike price to $K = 40$ (option is deep-in-the-money).

```
## new strike price K=40
set.seed(1234)
result_antithetic <- BS_antithetic(S0,sigma,mu,T,K=40,r,n)
set.seed(1234)
result_MC <- BS_MC(S0,sigma,mu,T,K=40,r,2*n)

value_antithetic <- result_antithetic[[2]]
value_MC <- result_MC[[2]]
sd_antithetic <- sd(result_antithetic[[1]])/sqrt(n)
sd_MC <- sd(result_MC[[1]])/sqrt(2*n)

#95 percent confidence interval
alpha <- 0.05
z <- qnorm(1-alpha/2)
lowerbound_antithetic <- value_antithetic - z*sd_antithetic
upperbound_antithetic <- value_antithetic + z*sd_antithetic
width_antithetic <- 2*z*sd_antithetic
```

```r
lowerbound_MC <- value_MC - z*sd_MC
upperbound_MC <- value_MC + z*sd_MC
width_MC <- 2*z*sd_MC

result_final2 <- matrix(c(value_antithetic, lowerbound_antithetic,
                          upperbound_antithetic, width_antithetic,
                          sd_antithetic, value_MC, lowerbound_MC,
                          upperbound_MC, width_MC, sd_MC),
                        byrow = FALSE, ncol = 2, nrow = 5)
colnames(result_final2) <- c("Antithetic K=40","MC K=40")
row.names(result_final2) <- c("Value","Lower Bound", "Upper Bound",
                              "Width", "Std. Dev.")
result_final2
```

```
##             Antithetic K=40    MC K=40
## Value           39.96893390 39.9985096
## Lower Bound     39.89979188 39.7182524
## Upper Bound     40.03807592 40.2787667
## Width            0.13828404  0.5605143
## Std. Dev.        0.03527719  0.1429910
```

```r
abs(sd_antithetic-sd_MC) #variance reduction
```

```
## [1] 0.1077138
```

```r
cbind(result_final1,result_final2)
```

```
##             Antithetic K=80    MC K=80 Antithetic K=40    MC K=40
## Value            7.83002713 7.94324377     39.96893390 39.9985096
## Lower Bound      7.67989366 7.75634721     39.89979188 39.7182524
## Upper Bound      7.98016061 8.13014033     40.03807592 40.2787667
## Width            0.30026695 0.37379312      0.13828404  0.5605143
## Std. Dev.        0.07660012 0.09535714      0.03527719  0.1429910
```

Now, the variance is reduced even further and the confidence interval has a smaller width when using antithetic variates than in the case before. However, for the naive Monte Carlo method the change in the strike has the contrary effect: the standard deviation is higher and the width of the 95% confidence interval is even larger. Therefore, the absolute difference of the standard deviations is much higher than in the case before: approximately 0.11.
This could be explained by the fact that a higher strike price leads to a payoff of 0 more often than it would be for the smaller strike price. Therefore, the antithetic negatively correlated variables have a greater influence on the estimated result and the variance is therefore reduced even further.