Simulation study involving geometric brownian motion

Mihnea S. Andrei[1], Sam Bauer.[2]

Abstract

Computing probabilities for stopping times.

1. Introduction

There are 2 approached:

1 Simply multiply the probability of each character obtained from the output of the NN transformer. This actually is truly Bayesian.

2 Use Bayesian statistics

In the following sections, let us assume that c_i is some character from the output language.

2. Methodology for simulating GBM

2.1 Simulating S_t the Geometric Brownian Motion

$$P_t = P_0 e^{\mu t + \sigma W_t}$$

Here, W_t is the standard brownian motion. In simulation, it is easier to divide by S_0 and take log:

$$\log\left(\frac{P_t}{P_0}\right) = \mu t + \sigma W_t$$

Now, you can choose any step size Δt and by replacing t with $t + \Delta t$ in the above equation, you obtain:

$$\log\left(\frac{P_{t+\Delta t}}{P_0}\right) = \mu(t + \Delta t) + \sigma W_{t+\Delta_t}$$

For simulation purposes, at step t I know P_t and I want to find $P_{t+\Delta t}$. So I subtract the last to equations:

$$\log\left(\frac{P_{t+\Delta t}}{P_0}\right) - \log\left(\frac{P_t}{P_0}\right) = \mu\Delta t + \sigma\left(W_{t+\Delta t} - W_t\right)$$

The left hand side is by magic: $\log\left(\frac{\frac{P_{t+\Delta t}}{P_0}}{\frac{P_t}{P_0}}\right) = \log\left(\frac{P_{t+\Delta t}}{P_t}\right)$

$$\log\left(\frac{P_{t+\Delta t}}{P_t}\right) = \mu\Delta t + \sigma\left(W_{t+\Delta t} - W_t\right)$$

So, by taking exponential, I obtain the iterative way of simulating a GBM. At time $t + \Delta t$ the stock will be (remember that this is iterative so S_t is know

$$P_{t+\Delta t} = P_t e^{\mu\Delta t + \sigma(W_{t+\Delta t} - W_t)}$$

Now, as you mentioned in the very detailed explanation, there will be many values of μ,σ. Either way fr each pair this stock price can be simulated. Also, Δt is chosen small enough and acording to Einstein's definition of standard brownian motions, I just need to simulate $W_{t+\Delta t} - W_t \sim \mathcal{N}(0, \Delta t)$. Which is the same as simulating $Z \sim \mathcal{N}(0, 1)$ and doing $W_{t+\Delta t} - W_t = \sqrt{\Delta t} \cdot Z$.

3. Methodology for probabilities

Now, for each combination of μ and σ from the previous section, I can compute many paths for the prices P_t up until some time that is very far into the future (call it T).

- Let τ_u be the first time when the the stock $S_t > u$ (I use lower case u since those are not random, they are some inputs that you will choose). Let the maximum of the stock between times $(0, t)$ be $M_t, \forall t$. Therefore, one formalization for your question is:

$$\tau_u = \min_s \{0 \leq s \leq T | M_s > u\}$$

- Let M_t be the running maximum of the price in the interval $[0, t]$. Formally, this means that

$$M_t = max_{0 \leq s \leq t} P_s$$

  Of course P_s follows the GBM defined in the previous section.

- The trialing loss is triggered as soon as the price drops below the running maximum (M_t) minus a value r (lower case since R is not random, it is just something inputted by the user. Formally this means that we need to define another stopping time when this trailing loss is triggered:

$$\tau_r = min_s \{0 \leq s \leq t | P_s \leq M_s - r\} \tag{1}$$

The objective of this study is to compute $P(\tau_u < \tau_r)$

4. Implementation for probabilities

The objective formalized in the previouss ection is expressed in the example that Mr.Bauer provided in the proposal:

"Intermediate Result that is required for my project: One result reported in the output file is an estimate the probability stock price reaches objective U before a "Trailing stop-loss" is triggered by the price retracing a fixed amount R from its current maximum. If the maximum price is achieved at time t*, then the probability I am looking for is the probability that the stopping time of the maximum being above U is smaller than the stopping time of the minimum (defined on time interval t* s t being above (maximum - R)."

The code is written both in python and R. The one in R has the same speed for one combination of parameters as the one in python (R has as compiler directly g++, which is a C++ compiler). It is actually a bit faster than python for the same combination of parameters. Furthermore, we decided to use R because it is far much easier to truly parallelize the code. This means that we are simulating 10000 GBM paths on each core of a CPU and afterwards aggregating the results. TO my knwoledge, not even to this day python can't truly parallelise. It can do asynchronous tasks, giving the impression of parallel computing, but it is not true parallelism. Python basically switches between different tasks very fast, it is not running the task at once, like R can. It is insanely easy to do this in R also, using the doSNOW package. The hardest part with this package is aggregating the results from different cores usually.

Now, let me delve into the implementation of this parallelized R code.

The first part of the code contains some libraries, which have functions. Some of those are needed in this code, others were used in the printing code:

```
1  ##############
2  #Libraries
3  ##############
4  library(progress)
5  library(foreach)
6  library(parallel)
7  library(doSNOW)
8  set.seed(123456789)
```

Figure 1. Libraries and seed

The set.seed() functions sets a seed so that the results are reproducible on different runs of the code.

The next line sets the working directory to the folder where the script is situated. This is done so that all the printed information and output of the code is done in this folder:

```
###########
#Change the following path to the path where this code is located in your laptop.
###########
setwd("~/Work/BM_simulation/R_codes")
```

Figure 2. Working directory

In the next lines of code, I simply set the parameters, according to Mr. Bauer's specifications.

Remark 1. Please note that r_range and r in the code are the powers of wealth used in the CRRA utility function, which I will introduce a little bit later. This is not to be confused with r from this paper, which is the retracement for the trailing stop loss from definition (1).

In the next figure, you can see all the parameters of the model, as defined by Mr. Bauer:

```
###############
#Initialize the parameters
#################
M<-5.867
G<-5.34
P0<-100
delta_t<-1/2530
L<-630/2530
L_range<-seq(from=0,to=L,by=delta_t)
N<-10^(4)

r_range<-1:5
len_r_range<-length(r_range)

mu_range<-seq(from=0,to=0.25,by=0.01)
#mu_range<-seq(from=0,to=0.25,length.out=2)
len_mu_range<-length(mu_range)

#sigma_range<-seq(from=0.05,to=0.3,by=0.01)
sigma_range<-seq(from=0.1,to=0.1,length.out=1)
len_sigma_range<-length(sigma_range)

#u_range<-seq(from=110,to=300,by=10)
u_range<-seq(from=115,to=115,length.out=1)
#u_range<-P0+(1:7)*M
len_u_range<-length(u_range)

alpha_range<-seq(from=0.05,to=1,by=0.05)
#alpha_range<-seq(from=0.05,to=1,length.out=2)
#alpha_range<-c(0.05,0.1,0.15)
len_alpha_range<-length(alpha_range)
```

Figure 3. List of all parameters

The next line counts the total number of combinations of parameters and creates a counter. They are used to print in a txt file the completion percentage:

```
n_iters<-len_mu_range*len_sigma_range*len_u_range*len_alpha_range
count_iters<-1
```

Figure 4. Counter and nr of combinations

Afterwards, I specify the number of cores. I usually set it to 2 less the total number of cores on the PC or laptop. This is because if I use all the cores on a pc or laptop, it will freeze up. ON the server, all the cores available can be used. I also create a txt file that outputs information about completion and the parameters that were used so far.

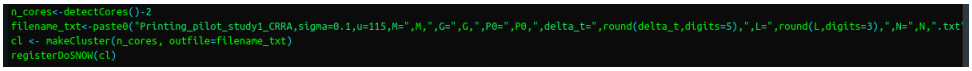

Figure 5. Nr of cores

In order to run in parallel different combinations of μ, σ, u and α, I just have to use foreach instead of for. Also, I have to specify how to combine the results from each core. Each row will contain:

- the μ used
- the σ used
- the u used
- the α used
- the r used
- the probability of reaching objective
- the probability of stop loss
- the probability of neither
- the expected return
- the standard deviation of return
- the CRRA utility (the formula for which I will present later)

However, in this code, r is not parallelized since r is just the power in the following CRRA formula:

$$CRRA = \frac{w^{1-r}}{1-r}, \text{ where}$$
$$w = \left(1 - \frac{\alpha}{M}(Pv' - Pv)\right) e^{\frac{\alpha}{M}(Pv - Pv')} - 1 \tag{2}$$

Here the following notation was introduced:

- v is the time when the trading ends: either the objective u is reached or the trailing stop loss is triggered.
- Pv is the price at which trading stops (either the objective u is reached or the trailing stop loss is triggered.
- Pv' is the maximum price reached in the interval of time $[0, v]$

Now, since r only shows up in this formula and nowhere else, it would be inefficient to run it in parallel. Hence, at the very end of the

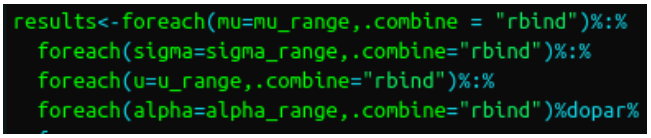

Figure 6. Foreach

I initialize parameters for each combination of μ, σ, u and α:

```
Returns<-matrix(0,nrow = N,ncol = 1)
crra<-matrix(0,nrow = N,ncol = len_r_range)
R<-(M/alpha)-G
count_stop_loss<-0
count_objective<-0
count_end<-0
```

Figure 7. Initialize variables for each combination of parameters

Each one has the following role:

- Returns will be a vector of size N that will compute the return for each path. The formula for returns is:

$$\text{Return} = e^{\frac{\alpha}{M}(Pv'-P0)}\left(1+\frac{\alpha}{M}(Pv-Pv')\right)-1$$

- CRRRA is a utility defined in (2).
- R is the retracement for the trailing stop loss as defined in definition (1).

Next is a for loop which generates each one of the N paths.

```
70      for(i in 1:N)
71      {
```

Figure 8. For loop creating for each path

Inside this for loop I have the following:

- I initialize a vector P which will contain all prices and the first price to be the inputted value by the user P_0. I also intialize the time to be 0 and I create a flag for objective being hit first and a flag for trailing stop loss being hit first. This will help me a bit later in the code figure out which one of the 2 occurred first.

```
P<-c()
P<-c(P,P0)
t<-0
flag_obj<-1
flag_trail<-1
```

Figure 9. Initialize parameters for each path

- Inside the for loop, I keep on generating new prices according to the discretization of the GBM:

```
while(t<=L)
{
  z<-rnorm(n=1,mean=0,sd=1)
  P<-c(P,P[length(P)]*exp(mu*delta_t+sigma*sqrt(delta_t)*z))
  len_P<-length(P)
```

Figure 10. Generate new price using the GBM discretization formula

- If the last price generated is below the trailing stop loss, I memorize the time, this last price and the maximum of the price until this point and I break from the while loop (which means that I stop generating new prices according to the GBM discretization):

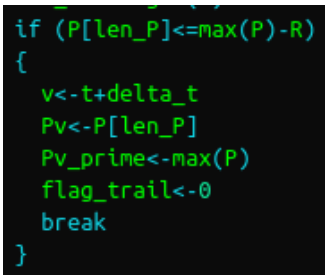

Figure 11. Trailing loss reached

- If the objective is reached, I memorize the time, this last price, the maximum of the price until this point and I break from this while loop (which means that I stop generating new prices according to the GBM discretization):

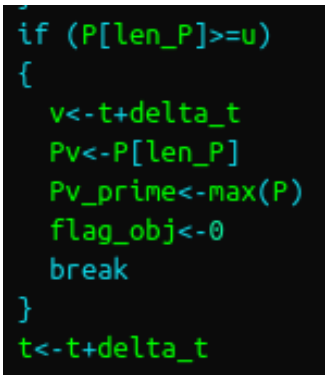

Figure 12. Objective is reached

As you probably noticed, I have some flags for objective and trailing stop loss. Their use comes into play now. Once exiting the while loop that generates the GBM path, I have 3 scenarios:

- If both flags are 1 it means that neither the objective nor the trailing stop loss were reached. So I am at the end of the time for simulating this GBM and $v = L$, Pv is the last price and Pv' is the maximum of the price over the whole time interval $[0, L]$:

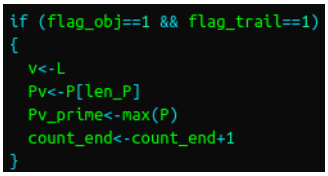

Figure 13. Neither objective or trailing loss is reached

Next, I compute the return according to the formula provided by Mr. Bauer:

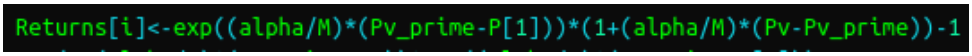

Figure 14. Return formula

$$\text{Return} = e^{\frac{\alpha}{M}(Pv' - P_0)}\left(1 + \frac{\alpha}{M}(Pv - Pv')\right)$$

I also compute the wealth according to the formula provided by Mr. Bauer:

```
w<-(1-(alpha/M)*(Pv_prime-Pv))*exp((alpha/M)*(Pv_prime-P[1]))
```

Figure 15. Wealth formula

Here the formula for wealth is:

$$\text{wealth} = \left(1 - \frac{\alpha}{M}(Pv' - Pv)\right)e^{\frac{\alpha}{M}(Pv' - P_0)}$$

For each value of r, I compute the CRRA utility, which is based on the wealth formula presented previously:

```
for(j in 1:len_r_range)
{
  if(r_range[j]==1)
  {
    crra[i,j]<-log(w)
  }
  else
  {
    crra[i,j]<-(1/(1-r_range[j]))*(w^(1-r_range[j]))
  }
```

Figure 16. CRRA utility

Compute the mean and standard deviation of the returns:

```
mean_return<-mean(Returns)
sd_return<-sd(Returns)
```

Figure 17. Mean and standard deviation of returns

Print some information about the iterations that were completed:

Figure 18. Print information to txt

Now, the very last line of the code is what the foreach function concatenates according to the rules supplied in the .combine argument. Since I supplied the "rbind" argument, it means that the following results will be stacked on top of each other:

```
cbind(rep(mu,times=len_r_range),rep(sigma,times=len_r_range),rep(u,times=len_r_range),rep(alpha,times=len_r_range),r_range,rep(mean_return,times=len_r_range),
      rep(sd_return,times=len_r_range),apply(crra,2,mean),rep(count_objective/N,times=len_r_range),rep(count_stop_loss/N,times=len_r_range),
      rep(count_end/N,times=len_r_range))
```

Figure 19. Combining results

Please notice how I repeat all the results for different values of r. This is because the only place where r shows up is in the formula (2) for CRRA.

The last part of the code just labels the columns with names, prints it to a csv file and also prints some relevant information:

```
colnames(results)<-c("mu","sigma","u","alpha","r","mean return","sd return","mean crra","prob obj","prob trail","prob neither")
(results<-as.data.frame(results))
print(paste0("The maximum expected CRRA is ",max(results$`mean crra`)))
print(paste0("The parameters that correspond to this maximum expected CRRA are: mu=",results$mu[which.max(results$`mean crra`)],
             ",sigma=",results$sigma[which.max(results$`mean crra`)],",u=",results$u[which.max(results$`mean crra`)],
             ",alpha=",results$alpha[which.max(results$`mean crra`)],",r=",results$r[which.max(results$`mean crra`)]))
print(paste0("For this maximum expected CRRA mean return=",results$`mean return`[which.max(results$`mean crra`)],
             ",sd return=",results$`sd return`[which.max(results$`mean crra`)],",prob obj=",results$`prob obj`[which.max(results$`mean crra`)],
             ",prob trail=",results$`prob trail`[which.max(results$`mean crra`)],",prob neither=",results$`prob neither`[which.max(results$`mean crra`)]))


filename_csv<-paste0("Results_pilot_study1_CRRA,sigma=0.1,u=115,M=",M,",G=",G,",P0=",P0,",delta_t=",round(delta_t,digits=5),",L=",round(L,digits=3),",N=",N,".csv")
write.csv(results,filename_csv)
```

Figure 20. Combining results

The very last line of the code closes up the connection to the cores on the PC. If you do not run it, you will see in the system monitor some significant RAM memory and less than 1% for each core used:

```
#stop the cluster of cores assigned for this task
stopCluster(cl)
```

Figure 21. Combining results

Formula for wealth including comissions:

$$w = \left(\frac{M}{M + C\alpha} + \frac{\alpha}{M + C\alpha}\right)(Pv - Pv')\, e^{\frac{\alpha}{M+C\alpha}(Pv - P_0)}$$

Remark 2. (Possible issue with the wealth formula). I think that there are scenarios in which the formula for wealth w from above gives 0, and, therefore when computing the utility as $\log(w)$, I obtain $-\infty$. In the case when the price is increasing all the time until it reached objective u, $Pv = Pv'$ and therefore the formula yields $w = 0$.

I ran the code with the suggested parameters:

```
#Initialize the parameters
##################
C<-1            #commission and slippage
M<-5.867
G<-5.34
P0<-100
delta_t<-1/2530
#L<-630/2530
L<-60
L_range<-seq(from=0,to=L,by=delta_t)
N<-10^(4)

r_range<-1:5
len_r_range<-length(r_range)

#mu_range<-seq(from=0,to=0.25,by=0.01)
mu_range<-seq(from=0.1,to=0.1,length.out=1)
len_mu_range<-length(mu_range)

#sigma_range<-seq(from=0.05,to=0.3,by=0.01)
sigma_range<-seq(from=0.2,to=0.2,length.out=1)
len_sigma_range<-length(sigma_range)

#u_range<-seq(from=110,to=300,by=10)
#u_range<-seq(from=115,to=115,length.out=1)
#u_range<-P0+(1:7)*M
u_range<-c(3*M)
len_u_range<-length(u_range)

#alpha_range<-seq(from=0.05,to=1,by=0.05)
alpha_range<-seq(from=0.05,to=0.05,length.out=1)
#alpha_range<-c(0.05,0.1,0.15)
len_alpha_range<-length(alpha_range)
```

Figure 22. Parameters for new wealth

But I obtain many NA's:

```
> (results<-as.data.frame(results))
  mu sigma      u alpha r   mean return   sd return mean crra wealth prob obj prob trail prob neither
1 0.1   0.2 17.601  0.05 1 -9.743318e-06 0.003378509        NA     NA        1          0           0
2 0.1   0.2 17.601  0.05 2 -9.743318e-06 0.003378509        NA     NA        1          0           0
3 0.1   0.2 17.601  0.05 3 -9.743318e-06 0.003378509        NA     NA        1          0           0
4 0.1   0.2 17.601  0.05 4 -9.743318e-06 0.003378509        NA     NA        1          0           0
5 0.1   0.2 17.601  0.05 5 -9.743318e-06 0.003378509        NA     NA        1          0           0
>
```

Figure 23. Results for new wealth

If I run the same code, but with the old formula, there are no NAs:

```
> (results<-as.data.frame(results))
  mu sigma      u alpha r  mean return   sd return     mean crra   wealth prob obj prob trail prob neither
1 0.1   0.2 17.601  0.05 1 7.434504e-05 0.003430264  6.846043e-05 1.006525        1          0           0
2 0.1   0.2 17.601  0.05 2 7.434504e-05 0.003430264 -9.999374e-01 1.006525        1          0           0
3 0.1   0.2 17.601  0.05 3 7.434504e-05 0.003430264 -4.999433e-01 1.006525        1          0           0
4 0.1   0.2 17.601  0.05 4 7.434504e-05 0.003430264 -3.332825e-01 1.006525        1          0           0
5 0.1   0.2 17.601  0.05 5 7.434504e-05 0.003430264 -2.499551e-01 1.006525        1          0           0
```

Figure 24. Results for old wealth formula