Computational Finance - Hand-in 1

Clara E. Tørsløv (cnp777)

30. september 2021

Question 1

Problem 1

From the the <random> header we instantiate the MT object by std::mt19937 using seed 1. A random number using the MT object is generated to the console. The minimum and maximum values the random number generator is able to produce is found by the functions .min() and .max(). These are respectively 0 and 4294967295.

Problem 2

The class MTRNG can be found in 'MTRNG.h' and 'MTRNG.cpp'. This class holds a MT variable as field variable. This field variable is private as we don't want the object to be accessed nor changed outside the class. Further the field variable is made to be non-const as we wish to use it in later problems, fx changing the seed of the generator. An object from the class is instantiated. Note that nothing is outputted in this problem.

Problem 3

The ()-operator, which generates one $U(0,1)$ variable, is implemented in 'MTRNG.h' and MTRNG.cpp'. With seed 1 a realization of a $U(1,0)$ is generated to the console.

Problem 4

The setSeed method is implemented in 'MTRNG.h' and MTRNG.cpp'. The method is used to generate the exact same random number twice. The results are generated to the console.

Problem 5

The overloaded ()-operator is implemented in 'MTRNG.h' and MTRNG.cpp'.

To overload the ()-operator to take in upper and lower bounds for the uniform distribution we need to make the following note.

$$a + (b - a) \cdot X \sim U(a, b), \quad X \sim U(0, 1) \text{ and } a, b \in \mathbb{R}$$

A realization of a $U(1, 2)$ variable with seed 1 is generated to the console.

Problem 6

Let F be a continuous, strictly increasing cumulative distribution function and let $U \sim U(0, 1)$.

Suppose X has CDF F, meaning $F(x) = \mathbb{P}(X \leq x)$. As F is strictly increasing and continuous, the inverse, F^{-1}, exists and will be strictly increasing and continuous.

We now show that $F^{-1}(U)$ has distribution function F.

$$\mathbb{P}(F^{-1}(U) \leq x) = \mathbb{P}(F(F^{-1}(U)) \leq F(x)) = \mathbb{P}(U \leq F(x)) = F(x)$$

Note that F^{-1} is well-defined on $(0, 1)$ and therefore can take in U. Further F can be used on both sides of the inequality as it is a strictly increasing, continuous and positive function. The last equation holds as $\mathbb{P}(U \leq u) = u$ by the definition of the uniform distribution.

Problem 7

The method genNormal is implemented in 'MTRNG.h' and MTRNG.cpp'. The method uses the invNormCdf method (defined in 'gaussian_header.hpp') on realizations of $U(0, 1)$ generated by the ()-operator. The invNormCdf() is defined by the Beasley-Springer-Moro algorithm (Fig. 2.13. in Appendix A).

The seed is set to 1 and a realization of a $\mathcal{N}(0, 1)$ is generated to the console.

Problem 8

The overloaded constructor, which takes a mt19937 object as input, can be found in 'MTRNG.h' and MTRNG.cpp'. An mt19937 object with seed 10 is used to instantiate the class and a standard normal realization is generated to the console.

Problem 9

See 'MTRNG.h' and MTRNG.cpp' for the overloaded constructor, which takes a vector as input, the getDim method, which returns the dimension and the overloaded genNormal, which fills an inputted vector with standard normal realizations.

The seed is set to 1 and the genNormal method is used to fill a vector of dimension 5 with realizations of normal random variables after which the realizations are outputted.

Problem 10

The copy constructor is implemented in 'MTRNG.h' and MTRNG.cpp'.

The seed of a previous instantiation is set to 1 and a copy is made using the copy constructor. A standard uniform is generated to the console using the copied object.

As we have implemented a copy constructor one might think that by the Rule of Three we would also need a destructor. However this is not the case as we do not work with dynamically allocated memory. In our case we can simply rely on the default setting.

Problem 11

The templated class can be seen in 'MTRNG.h'. This class contains a templated random number generator as a field variable, a constructor that takes an unsigned int and instantiates the generator with the seed, a setSeed method that sets the seed of the generator, a constructor which takes a random number generator as the input and a method genUniform that generates a single $U(0,1)$ realization.

A realization of an U(0,1) random variable is generated to the console first by using the Mersenne Twister generator and second by using the newer minimum standard random number generator. Each time first by instantiating a tempClass object with seed 1 and second by instantiating a random number generator object with seed 1, applying the constructor thus instantiating a tempClass object and generating the realization to the console using the genUniform method.

Question 2

The general short rate dynamics under the $\mathbb{Q}$-measure are given as follows

$$dr_t = \mu(r_t)dt + \sigma(r_t)dW_t^{\mathbb{Q}} \tag{1}$$

The term structure equation states that in an arbitrage free bond market a T-bond denoted F^T (supressing dependencies) solves the PDE (with initial condition $F(T,r,T) = 1$).

$$F_t^T + \mu(r)F_r^T + \frac{1}{2}\sigma^2(r)F_{rr}^T - rF^T = 0 \tag{2}$$

Problem 1

We assume that the model admits an affine term structure meaning that bond prices have the form

$$F(t,r,T) = e^{A(t,T)+B(t,T)\cdot r} \tag{3}$$

We wish to derive the resulting term structure equation. Note the following

$$\begin{aligned}
F_t^T &= \frac{\partial F^T}{\partial t} = e^{A(t,T)+B(t,T)\cdot r}(A_t(t,T) + B_t(t,T)\cdot r) \\
F_r^T &= \frac{\partial F^T}{\partial r} = e^{A(t,T)+B(t,T)\cdot r}B(t,T) \\
F_{rr}^T &= \frac{\partial^2 F^T}{\partial r^2} = e^{A(t,T)+B(t,T)\cdot r}B^2(t,T)
\end{aligned}$$

Thus, as F^T must solve the term structure equation (2), we have

$$\begin{aligned}
& F^T(A_t(t,T) + B_t(t,T)\cdot r) + \mu(r)F^T B(t,T) + \frac{1}{2}\sigma^2(r)F^T B^2(t,T) - rF^T = 0 \\
\Leftrightarrow \quad & A_t(t,T) + B_t(t,T)\cdot r + \mu(r)B(t,T) + \frac{1}{2}\sigma^2(r)B^2(t,T) - r = 0 \\
\Leftrightarrow \quad & A_t(t,T) - (1 - B_t(t,T))\cdot r + \mu(r)B(t,T) + \frac{1}{2}\sigma^2(r)B^2(t,T) = 0
\end{aligned}$$

Problem 2

Assume $\mu(r) = \kappa(\theta - r)$ and $\sigma(r) = \Sigma\sqrt{\gamma + \delta r}$. Thus the term structure equation becomes

$$\begin{aligned}
& A_t(t,T) - (1 - B_t(t,T))\cdot r + \kappa(\theta - r)B(t,T) + \frac{1}{2}\Sigma^2(\gamma + \delta r)B^2(t,T) = 0 \\
\Leftrightarrow \quad & A_t(t,T) + \kappa\theta B(t,T) + \frac{1}{2}\Sigma^2\gamma B^2(t,T) - \left(1 - B_t(t,T) + \kappa B(t,T) - \frac{1}{2}\Sigma^2\delta B^2(t,T)\right)\cdot r = 0
\end{aligned}$$

The resulting equation must hold for all t, T and r. Thus for fixed t and T the coefficient of r must be equal to zero. We therefore obtain the equation

$$\begin{aligned} & 1 - B_t(t,T) + \kappa B(t,T) - \frac{1}{2}\Sigma^2 \delta B^2(t,T) = 0 \\ \Leftrightarrow \quad & B_t(t,T) = \kappa B(t,T) - \frac{1}{2}\Sigma^2 \delta B^2(t,T) + 1 \end{aligned}$$

Further, as the r-term is equal to zero, we must have that

$$\begin{aligned} & A_t(t,T) + \kappa\theta B(t,T) + \frac{1}{2}\Sigma^2 \gamma B^2(t,T) = 0 \\ \Leftrightarrow \quad & A_t(t,T) = -\kappa\theta B(t,T) - \frac{1}{2}\Sigma^2 \gamma B^2(t,T) \end{aligned}$$

Thus arriving at the ordinary differential equations (ODEs)

$$\frac{\partial B(t,T)}{\partial t} = \kappa B(t,T) - \frac{1}{2}\Sigma^2 \delta B^2(t,T) + 1 \tag{4}$$

$$\frac{\partial A(t,T)}{\partial t} = -\kappa\theta B(t,T) - \frac{1}{2}\Sigma^2 \gamma B^2(t,T) \tag{5}$$

with initial conditions

$$B(T,T) = 0$$

$$A(T,T) = 0$$

$F(T,r,T) = 1$ entails that $e^{A(T,T)+B(T,T)r} = 1$ thus $A(T,T) + B(T,T)r = 0$. As this should hold for all values of r it must be that $A(T,T) = 0$ and $B(T,T) = 0$.

Problem 3

We consider the Vasicek model with short rate dynamics

$$dr_t = \kappa(\theta - r_t)dt + \Sigma dW_t^{\mathbb{Q}} \tag{6}$$

Meaning that

$$\mu(r) = \kappa(\theta - r) \text{ and } \sigma(r) = \Sigma$$

Following the same argumentation as in the previous subproblem we find that the Vasicek-specific set of ODEs are

$$\frac{\partial B(t,T)}{\partial t} = \kappa B(t,T) + 1 \tag{7}$$
$$\frac{\partial A(t,T)}{\partial t} = -\kappa\theta B(t,T) - \frac{1}{2}\Sigma^2 B^2(t,T) \tag{8}$$

In the files 'VasicekZCB.h' and 'VasicekZCB.cpp' the Runge-Kutta method is applied to create a function that solves the ODEs backwards and returns the price of a T-bond in the Vasicek model. Note that the timesteps are negative as we start at maturity T and iterate backwards towards zero. This is the case due to our ODE conditions ($A(T,T) = 0$ and $B(T,T) = 0$).

Using the given parameter values the bond value is generated to the console.

Problem 4

A Vasicek model class is implemented in the files 'VasicekZCB.h' and 'VasicekZCB.cpp'. It holds parameters r, κ, θ, Σ, h, get-methods for each variable and a constructor.

A Vasicek class object is instantiated using the given values and the values generated to the console using the respective get-methods.

Problem 5

The overloaded pricing function (from 2.3), which takes in a Vasicek model object, can be found in 'VasicekZCB.h' and 'VasicekZCB.cpp'.

The value of the bond is generated to the console using the overloaded pricing function.

Problem 6

A Zero Coupon Bond class is implemented in the files 'VasicekZCB.h' and 'VasicekZCB.cpp'.

The value of the bond is generated to the console using the function from subproblem 2.5, overloaded to also take in a ZCB class object.

Problem 7

See the files 'VasicekZCB.h' and 'VasicekZCB.cpp' to find the mthod solveODE, which takes in a maturity and solves the system of ODEs for all maturities until that time point, and the method getODE, which takes in a time point and returns the solution of the ODE at the specific maturity.

The solveODE method uses the Runge-Kutta method to solve the system of ODEs. The ODE solutions for all timepoints are stored in a private field variable named 'solutions'. This way we only have to run solveODE once and thus reduce runtime. The solutions matrix has the size $2 \times ((T/h) + 2)$ where the solutions to ODE 1 is reached by calling column 1 and solutions to ODE 2 is reached by calling column 2. A specific timepoint is reached by calling a certain row.

The solution to the ODEs for $T = 10$ is generated to the console using first solveODE and next getODE.

Problem 8

A Vasicek Bond class is implemented in the files 'VasicekZCB.h' and 'VasicekZCB.cpp'. It holds a Vasicek model object, a zero coupon bond object and a getPrice method, which evaluates the bond price using the ODE solution stored in the Vasicek model object.

The value of a bond with respectively 1 and 3 years to maturity is generated to the console.

Problem 9

See 'ShortRateBond.h' to see the abstract short rate bond pricing class, which holds a virtual getPrice method. In the file 'VasicekZCB.h' the Vasicek bond class inherits from this class. Note, that nothing is outputted in this problem.

Problem 10

We consider the CIR model with short rate dynamics

$$dr_t = \kappa(\theta - r_t)dt + \Sigma\sqrt{r}dW_t^{\mathbb{Q}} \tag{9}$$

Meaning

$$\mu(r) = \kappa(\theta - r) \text{ and } \sigma(r) = \Sigma\sqrt{r}$$

Thus, using the same procedure, the CIR-specific set of ODEs are

$$\frac{\partial B(t,T)}{\partial t} = \kappa B(t,T) + \frac{1}{2}\Sigma^2 B^2(t,T) + 1 \tag{10}$$

$$\frac{\partial A(t,T)}{\partial t} = -\kappa\theta B(t,T) \tag{11}$$

In the files 'CIR.h' and 'CIR.cpp' the Runge-Kutta method is again applied to create a function that solves the ODEs backwards and returns the price of a T-bond in the CIR model.

Using the given parameter values the bond value is generated to the console.

Problem 11

See the files 'CIR.h' and 'CIR.cpp' to find the mthod solveODE, which takes in a maturity and solves the system of ODEs for all maturities until that time point, and the method getODE, which takes in a time point and returns the solution of the ODE at the specific maturity.

The solution to the ODEs for $T = 10$ is generated to the console using first solveODE and next getODE.

Further a CIR bond class, which inherits from the abstract base class and holds a getPrice method, is implemented in the files 'CIR.h' and 'CIR.cpp'.

The value of a CIR bond with respectively 1 and 3 years to maturity is generated to the console.

Problem 12

See 'Portfolio.h' and 'Portfolio.cpp' for the portfolio class, which holds a dimension, two vectors of the given dimension holding short rate bond pricing objects and notional values for each bond, a constructor and a getPrice method, which returns the value of the portfolio. By adding const after '*' we make sure that the pointer can't be reassigned to point to something else.

The price of the portfolio holding a 100 notional value Vasicek bond and a 100 notional value CIR bond, both with $T = 1$ is generated to the console, using the getPrice method.

Problem 13

In the files 'Portfolio.h' and 'Portfolio.cpp' a constructor creating an empty portfolio and the methods addBond and removeBond is added to the portfolio class.

The addBond method uses std::vector::push_back to add a given bond to the end of the portfolio vector. The method removeBond takes in an integer n and uses std::vector::erase to remove the n'th element of the portfolio vector.

We test the newly implemented methods by creating an empty portfolio and generating the value of the portfolio to the console. Then adding two bonds (100 notional value CIR bond with $T = 1$ and a 100 notional value Vasicek bond with $T = 3$) to the empty portfolio and yet again generating the value to the console. Lastly we remove the Vasicek Bond and generate the value of the portfolio to the console.

Question 3

Problem 1

In the file 'gaussian_header.hpp' the NormCDF method is implemented using the Zelen and Severo (1964) approximation to the normal CDF.

We check that our implementation works by comparing with theoretical values. We know by heart the value of the normal CDF in the points 0 and ± 1.96. We compare in the console output and conclude that the approximation is good.

Problem 2

In the files 'SABR.h' and 'SABR.cpp' a SABR model class is implemented. This holds parameters, a method which returns the implied volatility[1], and a method BS_CallPrice which evaluates the Black-Scholes formula using the approximation of the implied volatility and returns the price of the call option.

The value of the call option in the SABR model is then generated to the console by the BS function.

Problem 3

Assume we have two independent standard normal random variables, N_1 and N_2. Frem these we wish to create two standard normal random variables, Z_1 and Z_2, where $\text{Corr}(Z_1, Z_2) = \rho$. Define

$$
\begin{aligned}
Z_1 &= N_1 \sim \mathcal{N}(0,1) \\
Z_2 &= \rho N_1 + \sqrt{1-\rho^2} N_2 \sim \mathcal{N}(\rho \underbrace{\mathbb{E}[N_1]}_{=0} + \sqrt{1-\rho^2} \underbrace{\mathbb{E}[N_2]}_{=0}, \rho^2 \underbrace{\mathbb{V}[N_1]}_{=1} + \left(\sqrt{1-\rho^2}\right)^2 \underbrace{\mathbb{V}[N_2]}_{=1}) = \mathcal{N}(0,1)
\end{aligned}
$$

Then

$$
\begin{aligned}
\text{Corr}(Z_1, Z_2) &= \frac{\text{Cov}(Z_1, Z_2)}{\underbrace{\sqrt{\mathbb{V}[Z_1]}}_{=1} \underbrace{\sqrt{\mathbb{V}[Z_2]}}_{=1}} = \text{Cov}(Z_1, Z_2) = \text{Cov}(N_1, \rho N_1 + \sqrt{1-\rho^2} N_2) \\
&= \rho \underbrace{\text{Cov}(N_1, N_1)}_{=\mathbb{V}[N_1]=1} + \sqrt{1-\rho^2} \underbrace{\text{Cov}(N_1, N_2)}_{=0} = \rho
\end{aligned}
$$

Thus we now have two standard normal random variables, Z_1 and Z_2, with correlation ρ, given by the two independent standard normal random variables, N_1 and N_2. To extend this result to increments of

[1] As approximated by the method here: https://en.wikipedia.org/wiki/SABR_volatility_model#Asymptotic_solution

correlated Wiener processes we need to make the following notes

$$W_t \sim \mathcal{N}(0,t) \Rightarrow W_t \overset{d}{=} \sqrt{t} \cdot N_1$$
$$W_{t+\Delta} - W_t \sim \mathcal{N}(0,\Delta) \Rightarrow W_{t+\Delta} - W_t \overset{d}{=} \sqrt{\Delta} \cdot N_1$$

Then, from two independent Wiener processes W_t^1 and W_t^2, we construct the Wiener process $W_t^3 = \rho W_t^1 + \sqrt{1-\rho^2} W_t^2$ and note

$$\text{Corr}(W_t^1, W_t^3) = \frac{\text{Cov}(W_t^1, W_t^3)}{\underbrace{\sqrt{\mathbb{V}[W_t^1]}}_{=\sqrt{t}} \underbrace{\sqrt{\mathbb{V}[W_t^3]}}_{=\sqrt{t}}} = \frac{\rho \text{Cov}(W_t^1, W_t^1) + \sqrt{1-\rho^2}\text{Cov}(W_t^1, W_t^2)}{\sqrt{t}\sqrt{t}} = \frac{t\rho}{t} = \rho$$

Thus to simulate increments in correlated Wiener processes, one will only have to simulate two standard normal variables, N_1 and N_2, and use the above scheme.

Problem 4

In the files 'SABR.h' and 'SABR.cpp' the method genPath is implemented in the SABR class. This method takes in a vector of normals and a maturity and returns the price of the underlying at maturity. Within the genPath method we simulate the SABR model dynamics using the Euler scheme

$$S_{T_{i+1}} = S_{T_i} + \sigma_{T_i} S_{T_i}^{\beta} \left(W_{T_{i+1}}^1 - W_{T_i}^1 \right)$$
$$\sigma_{T_{i+1}} = \sigma_{T_i} \exp\left(-\frac{1}{2}\alpha^2 (T_{i+1} - T_i) + \alpha \left(W_{T_{i+1}}^2 - W_{T_i}^2 \right) \right)$$

where the increments in the correlated Wiener processes, $(W_t^1)_{t=0,\dots,T}$ and $(W_t^2)_{t=0,\dots,T}$, are simulated using the scheme from problem 3.3. The discretized SDE may obtain negative values if the increments of the Wiener processes become negative. By definition of Wiener processes we know that these increments are normally distributed and thus obtain negative values with a probability greater than 0. We cannot simply ignore this problem as we have been given the value $\beta = \frac{1}{2}$, meaning the discretized SDE includes a square root and thus computation might be impossible. We fix it by using the full-truncation scheme[2], i.e.

$$S_{T_{i+1}} = S_{T_i}^+ + \sigma_{T_i} (S_{T_i}^+)^{\beta} \left(W_{T_{i+1}}^1 - W_{T_i}^1 \right)$$

Further, within the genPath method, we need a method to seperate the content of the vector of iid standard normal variables. We do so by using even and uneven indices. The length of the normals vector,

[2] In the course Continous Time Finance 2 we wrote a paper which analyzed different discretization schemes to handle this problem. Here the full-truncation scheme performed best in regards to precision and runtime.

and the maturity T, determines the amount of timesteps.

A vector of normals is filled using seed 1 and the genNormal method from 'MTRNG.h' and 'MTRNG.cpp'. This vector and genPath is then used to generate the result of a single path with 100 steps to the console.

Problem 5

In the files 'CallOption.h' and 'CallOption.cpp' a call option class is implemented. This holds the strike and maturity as well as a constructor, get-methods and a method to evaluate the payoff of a call option. Note that const is added to the get-methods as we do not wish for the methods to modify class member variables. Further note that nothing is outputted in this problem.

Problem 6

In the files 'SABR.h' and 'SABR.cpp' the Monte Carlo pricing function, MC_SABR, is implemented. This function takes in a SABR model object (instantiated in 3.2), a call option object (instantiated in 3.5), a random number generator (MTRNG) object (instantiated in 1.2), a number of steps and a number of paths.

The Monte Carlo function uses the MTRNG class object to fill a vector, with the length equal to number of steps, with iid normals. This is done inside the Monte Carlo loop. The method genPath from the SABR class object is used to generate the result of a single path with the given number of steps. The call option payoff is evaluated using the CallPayoff method from the CallOption class object using the generated path. This is done until the MCprice vector, with length equal to the given number of paths, is full. The method std::accumulate, which sums all the entries of a vector, is used and this result is divided by number of paths to return the Monte Carlo estimator.

The seed is set to 1 and the payoff of a call option using MC_SABR with 10.000 paths each with 100 steps is generated to the console.

Problem 7

In the files 'SABR.h' and 'SABR.cpp' a method PathsRequired is implemented. The method takes in a SABR model object, a call object, a random number generator, a number of steps and an error term ϵ and returns the number of paths required for the relative error between the analytical approximation and the Monte Carlo price to be less than a given value ϵ.

The seed is set to 1 and the number of paths required, using 100 steps per path and an error term $\epsilon = 0.001$, is generated to the console.

C++ console output

```
Question 1
---------------------------

---- Q1.P1 ----
Outputs using instantiation of MT object with seed 1
Random 1791095845

---- Q1.P2 ----

---- Q1.P3 ----
Realization of a U(0,1) random variable: 0.417022

---- Q1.P4 ----
Realization of U(0,1) with seed 1: 0.417022
Realization of U(0,1) after resetting seed to 1: 0.417022

---- Q1.P5 ----
Realization of U(1,2) with seed 1: 1.41702

---- Q1.P6 ----

---- Q1.P7 ----
Realization of N(0,1) with seed 1: -0.209518

---- Q1.P8 ----
Realization of N(0,1) with seed 10: 0.743203

---- Q1.P9 ----
Vector of dimension 5 holding realizations of N(0,1)
-0.209518
2.76856
0.583806
1.49511
-3.68493
```

```
—— Q1.P10 ——
Realization of U(0,1) using copied object: 0.417022

—— Q1.P11 ——
Realization of U(0,1) (using templated class and MT) 0.417022
Realization of U(0,1) (using overloaded constructor and MT) 0.417022
Realization of U(0,1) (using templated class and minstd) 2.24779e-05
Realization of U(0,1) (using overloaded constructor and minstd) 2.24779e-05
```

Question 2

```
—— Q2.P1 ——

—— Q2.P2 ——

—— Q2.P3 ——
Bond price using Runge-Kutta function = 0.978917

—— Q2.P4 ——
kappa = 0.3
r = 0.02
theta = 0.03
Sigma = 0.01
h = 0.01

—— Q2.P5 ——
Bond price using Runge-Kutta function and Vasicek object = 0.978917

—— Q2.P6 ——
Bond price using Runge-Kutta function, Vasicek object and ZCB object = 0.978917

—— Q2.P7 ——
```

```
Solution to the ODE for T = 10 using getODE() from Vasicek class
B = -3.16738 A = -0.201888

---- Q2.P8 ----
Value of a Vasicek bond 1. 1 year to maturity using the getPrice method 0.978917
Value of a Vasicek bond w. 3 years to maturity using the getPrice method 0.932496

---- Q2.P9 ----

---- Q2.P10 ----
CIR bond value using Runge-Kutta method = 0.978904

---- Q2.P11 ----
Solution to the ODE for T = 10 using getODE() from CIR class
B = -3.16608 A = -0.204795

Value of a CIR bond w. 1 year to maturity using the getPrice method 0.978904
Value of a CIR bond w. 3 years to maturity using the getPrice method 0.932276

---- Q2.P12 ----
The value of the portfolio holding 1 Vasicek Bond and 1 CIR Bond,
both with 1 year to maturity and notional value 100, is 195.782

---- Q2.P13 ----
The value of the empty portfolio is 0

The value of the portfolio holding 1 Vasicek Bond w. T=3 and 1 CIR Bond w. T=1,
both with notional value 100, is 191.14

The value of the portfolio holding 1 CIR Bond w. T=1 and notional value 100 is 97.8904
```

Question 3

—— Q3.P1 ——

The approximation of the normal CDF, using the normalCdf method:

x	normCdf(x)	theoretical
−1.96	0.0249978	0.025
0	0.5	0.5
1.96	0.975002	0.975

—— Q3.P2 ——

The value of the call option in the SABR model is 4.21699

—— Q3.P3 ——

—— Q3.P4 ——

The value at maturity of a single path with 100 steps using seed 1 is 70.0941

—— Q3.P5 ——

—— Q3.P6 ——

The payoff of a call option, using seed 1 and MC_SABR with 100 steps and 10.000 paths, is 4.3037

—— Q3.P7 ——

Number of paths required, using seed = 1, 100 steps per path, for relative error < 0.001 is 2841