Assignment 1

September 16, 2021

Deadline

The assingment should be uploaded on Absalon strictly before 23.59 on Thursday September 30rd. Please remember to follow to the guidelines presented in the separate document "Handin Expectations".

Question 1: The <random> Header

The header <random> can do a lot better than what we have used it for in the lectures. In this exercise you will build your own random number class around one already in C++. Specifically you will use the predefined "Mersenne Twister" (MT). A description can be found here, you will not need to understand vey much about how it works to solve this exercise.

You can get the relevant documentation via mt19937. You might also be interested in Random.

1. Write a code that instantiates the MT object from the <random> header with seed 1. Output a random number using the MT object. What are the minimum and maximum values that the generator is able to produce?

2. Write a class, MTRNG, that holds a MT as field variable. Should it be private or public? Why? Should it be const or non-const? Why? Make a constructor that takes an uint_fast32_t[1] as seed and instantiates the generator. Create an MTRNG object using your class.

3. Implement the ()-operator of MTRNG to return a realization of a $U(0,1)$ random variable. Make sure the seed is 1 and output the value returned by the ()-operator of MTRNG.

4. Give the class a method setSeed that sets the seed of the underlying generator. Demonstrate that your setSeed method can be used to generate the exact same random number twice by outputting to the console.

5. Overload the ()-operator to take in upper and lower bounds for the uniform distribution. Set the seed to 1 and output a realization of a $U(1,2)$ variable.

6. Let F be a continuous, strictly increasing cumulative distribution function. Let $U \sim U(0,1)$. Prove that $F^{-1}(U)$ has distribution function F.

7. In appendix A you see a method for approximating the inverse of the normal CDF. Give the MTRNG class a method genNormal which generates a standard normal. You can use the supplied Gaussian header file. Make sure the seed is 1 and output a standard normal.

[1] Don't worry about it looking weird. It's just a data type like an unsigned integer, in fact you can just pretend it is an unsigned int.

8. Overload your constructor so that it can take a `mt19937` object as input. Demonstrate that it works by instantiating a `mt19937` object to seed 10, instantiate your class using it and output a standard normal.

9. Look up `std::vector` and `size_t`. Add a dimension field variable to your class and an appropriate constructor. Give it a `getDim` method to return the dimension. Overload `genNormal` to fill an inputted vector of the given dimension with standard normal realizations (Note this can be done without using the dimension field variable via range looping. Even if you choose to do it this way you should still add a dimension field variable to your class). You do not need to check that the dimension of the inputted vector is correct. Set the seed to 1 and use the method to fill a vector of dimension 5 with realizations of normal random variables after which you output them to the console.

10. Give your class a copy constructor. Does it need a destructor? Why/Why not? Set the seed of a previous instantiation to 1 and make a copy using your constructor. Output a standard uniform using the copied object.

11. Write a new class that holds a templated random number generator as a field variable. Make a constructor that takes an unsigned int and instantiates the generator with the seed. Give the class a `setSeed` method that sets the seed of the generator. Overload the constructor to also take a random number generator as the input. Give the class a method `genUniform` that generates a single $U(0,1)$ realization. Be sure not to delete any of your old code. Rather copy and adapt it to create your new templated class. Show that your new class works by instantiating an object of your class using the Mersenne Twister generator with seed 1 and outputting a standard uniform using `genUniform`. Use the overloaded constructor to create a new object, set the seed to 1 and output a standard uniform with `genUniform`. Repeat same steps using the newer minimum standard random number generator (see `minstd_rand` in Random) to show that your templated class works on different generators. Note, you will not use the templated class in any of the remaining subquestions.

Question 2: A Simple Bond Portfolio

In this question we will work with time homogeneous one-factor short rate models. We will consider the short rate dynamics directly under the risk-neutral Q-measure. The general short rate dynamics are given as follows

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

Furthermore, we assume that zero coupon bond prices have the form

$$p(t,T) = F(t,r,T). \tag{2}$$

the term structure equation states that in an arbitrage free bond market a T-bond denoted F^T (supressing dependencies) solves the PDE

$$\frac{\partial F^T}{\partial t} + \mu(r)\frac{\partial F^T}{\partial r} + \frac{1}{2}\sigma^2(r)\frac{\partial^2 F^T}{\partial r^2} - rF^T = 0 \tag{3}$$

with initial condition $F(T,r,T) = 1$.

1. A model is said to admit an affine term structure if bond prices have the form
$$F(t,r,T) = e^{A(t,T)+B(t,T)r}. \tag{4}$$
Assume that the model admits an affine term structure and derive the resulting term structure equation.

2. Now, assume that $\mu(r) = \kappa(\theta - r)$ and $\sigma(r) = \Sigma\sqrt{\gamma + \delta r}$. Insert the expressions in the term structure equation and collect the terms dependent on r. The resulting equation must hold for all t, T and r. Use this to derive the following set of ordinary differential equations (ODEs)
$$\frac{\partial B(t,T)}{\partial t} = \kappa B(t,T) - \frac{1}{2}\delta\Sigma^2 B^2(t,T) + 1, \tag{5}$$
$$\frac{\partial A(t,T)}{\partial t} = -\kappa\theta B(t,T) - \frac{1}{2}\gamma\Sigma^2 B^2(t,T). \tag{6}$$
Why must the initial conditions $A(T,T) = 0$ and $B(T,T) = 0$ hold?

3. Let us first consider the Vasicek model with short rate dynamics
$$dr_t = \kappa(\theta - r_t)dt + \Sigma dW_t^Q \tag{7}$$
Write up the Vasicek-specific set of ODEs. Appendix B describes the standard Runge-Kutta method for solving a system of ODEs. Apply

the Runge-Kutta method to create a function that solves the ODEs backwards and returns the price of a T-bond in the Vasicek model. Use the parameter values $r = 0.02$, $\kappa = 0.3$, $\theta = 0.03$, $\Sigma = 0.01$ and $T = 1$ with a step size of $\Delta = 1/100$ and output the resulting bond value.

4. Make a Vasicek model class. it needs to hold r, κ, θ, Σ and the step size Δ. Give it an appropriate constructor. Create get-methods for each of the variables. Make an initilization using the values above and output them to the console via the get methods.

5. Overload the pricing function from problem 3 to work of you Vasicek model object. Output the value of the bond using the overloaded pricing function.

6. Write a zero coupon bond class with appropriate field variables and methods. Overload the function from problem 5 to work of your zero coupon bond class and output the value of the bond.

7. Given a set of parameters we of course can simply solve the discretized ODEs and store the results in a vector of vectors allowing us to price multiple bonds of different maturity without having to re-solve the ODE each time. Give the Vasicek model a method `solveODE`. The method should take in a maturity and solve the system of ODEs for all maturities until that time point. The solution should be stored in a vector of vectors. Create a method `getODE` which takes in a time point and returns the solution of the ODE at the specific maturity. You do not have to check that the discretization admits the time point used when calling `getODE`. Call the `solveODE`-method and output the solution to the ODE for $T = 10$ using `getODE`.

8. Write a Vasicek bond class that holds a Vasicek model object and a zero coupon bond object. Give the class a `getPrice` method that evaluates the bond price using the ODE solution stored in the Vasicek model object. Output the value of the 1 year bond using the `getPrice` method. Repeat for a bond with 3 years to maturity.

9. Write an abstract short rate bond pricing class with a virtual `getPrice` method. Let the Vasicek bond class inherit from this class.

10. If, instead we consider a Cox-Ingersoll-Ross (CIR) short rate process, then the dynamics are given by

$$dr_t = \kappa(\theta - r_t)dt + \Sigma\sqrt{r_t}dW_t^Q. \tag{8}$$

Write up the CIR-specific set of ODEs. Again apply the Runge-kutta method to create a function that returns the price of a T-bond in the CIR-model. Use the parameter values $r = 0.02$, $\kappa = 0.3$, $\theta = 0.03$, $\delta = 0.01$ and $T = 1$ with a step size of $\Delta = 1/100$ and output the resulting bond value.

11. Make a CIR model class and repeat problem 7 for the CIR model class. Create a CIR bond class and, let it inherit from the abstract base class and implement the `getPrice` method. Use the `getPrice` method to output the value of the bond. Repeat for a bond with 3 years to maturity.

12. Write a portfolio class that holds a dimension and two vectors of the given dimension holding short rate bond pricing objects and notional values for each bond. Give the class a constructor. Adapt the rest of your code to accommodate it and implement a `getPrice` method returning the value of the portfolio. Demonstrate that the class works by initializing a portfolio with a 100 notional value Vasicek bond and a 100 notional value CIR bond using the parameters listed above and $T = 1$.

13. Implement a default constructor creating an empty portfolio. output the value of the empty portfolio. Give the portfolio class methods `addBond` and `removeBond` to add and remove short rate bond pricing objects (hint: See `std::vector::push_back` and `std::vector::erase`). Show that your methods work by creating an empty portfolio using the default constructor and outputting the value of the empty portfolio. Add a 100 notional value CIR bond with $T = 1$ and a 100 notional value Vasicek bond with $T = 3$ using the parameters listed above and output the updated portfolio value. Remove the Vasicek bond and output the portfolio value again.

Question 3: The SABR Model

The SABR model ($r = 0$) reads under the risk-neutral measure

$$dS_t = \sigma_t S_t^\beta dW_t^1 \tag{9}$$
$$d\sigma_t = \alpha \sigma_t dW_t^2 \tag{10}$$

where W^1 and W^2 are Wiener processes with correlation ρ and some initial values S_0 and σ_0 are given. Throughout the question we will consider the following set of values $S_0 = 90$, $K = 100$, $\sigma_0 = 1.3$, $\alpha = 0.5$, $\beta = 0.5$, $\rho = -0.5$ and $T = 3$.

1. At Norm CDF multiple methods of approximating the normal CDF are given. Implement the one given by "Zelen and Severo (1964)" in a function. You can use the supplied header `gaussian_header.hpp`. Check that it works properly either by comparing its value in a couple of points with simulated values, some high-level implementation or tabulated values.

2. The SABR model has gained a lot of its popularity due to its analytical approximation for call prices. Write a SABR model class that holds the relevant parameters. give the class a method which implements the approximation found here along with the Black-Scholes formula to evaluate the price of call options (F_{mid} in the link should be chosen as the geometric average). Output the value of the option.

We can simulate the SABR dynamics using the following Euler scheme

$$S_{T_{i+1}} = S_{T_i} + \sigma_{T_i} S_{T_i}^\beta \left(W_{T_{i+1}}^1 - W_{T_i}^1 \right) \tag{11}$$
$$\sigma_{T_{i+1}} = \sigma_{T_i} \exp\left(-\frac{1}{2}\alpha^2 (T_{i+1} - T_i) + \alpha \left(W_{T_{i+1}}^2 - W_{T_i}^2 \right) \right) \tag{12}$$

3. There is an easy way to get two standard normal random variables with correlation $-1 < \rho < 1$ given two independent ones. Derive the method. How can it be applied to simulate increments in correlated Wiener processes?

4. Give the SABR class a method `genPath` that takes a vectors of normals and a maturity, simulates a path using the SABR model and returns the price of the underlying at maturity. Use `genNormal` with dimension 200 and seed equal to 1 to generate the vector of normals and output the result of a single path with 100 steps (i.e. $\Delta = 1/100$ since you need 2 normals per step).

5. Create a call option class which holds the strike and maturity. Give the class a constructor, appropriate get-methods and a method to evaluate the payoff of a call option.

6. Write a Monte Carlo pricing function `MC_SABR` that takes a SABR model object, a call option object, a random number generator object from question 1, a number of steps and a number of paths. Set the seed to 1 and output the result of a simulation with 10000 paths each with 100 steps.

7. Create a function that determines the amount of paths required for the relative error between the analytical approximation and the Monte Carlo pricing to be less than a given value ε. the function should take in a model object, a call object, a number of steps and ε. It has to output the number of paths required to obtain a relative error less than ε. Set the seed to 1, use 100 steps per path and output the result of the function with $\varepsilon = 0.001$.

Appendix A: The Inverse Normal

Approximating the Inverse Normal

Applying the inverse transform method to the normal distribution entails evaluation of Φ^{-1}. At first sight, this may seem infeasible. However, there is really no reason to consider Φ^{-1} any less tractable than, e.g., a logarithm. Neither can be computed exactly in general, but both can be approximated with sufficient accuracy for applications. We discuss some specific methods for evaluating Φ^{-1}.

Because of the symmetry of the normal distribution,

$$\Phi^{-1}(1-u) = -\Phi^{-1}(u), \quad 0 < u < 1;$$

it therefore suffices to approximate Φ^{-1} on the interval $[0.5, 1)$ (or the interval $(0, 0.5]$) and then to use the symmetry property to extend the approximation to the rest of the unit interval. Beasley and Springer [43] provide a rational approximation

$$\Phi^{-1}(u) \approx \frac{\sum_{n=0}^{3} a_n (u - \frac{1}{2})^{2n+1}}{1 + \sum_{n=0}^{3} b_n (u - \frac{1}{2})^{2n}}, \tag{2.27}$$

for $0.5 \le u \le 0.92$, with constants a_n, b_n given in Figure 2.12; for $u > 0.92$ they use a rational function of $\sqrt{\log(1-u)}$. Moro [271] reports greater accuracy in the tails by replacing the second part of the Beasley-Springer approximation with a Chebyshev approximation

$$\Phi^{-1}(u) \approx g(u) = \sum_{n=0}^{8} c_n [\log(-\log(1-u))]^n, \quad 0.92 \le u < 1, \tag{2.28}$$

with constants c_n again given in Figure 2.12. Using the symmetry rule, this gives

$$\Phi^{-1}(u) \approx -g(1-u) \quad 0 < u \le .08.$$

With this modification, Moro [271] finds a maximum absolute error of 3×10^{-9} out to seven standard deviations (i.e., over the range $\Phi(-7) \le u \le \Phi(7)$). The combined algorithm from Moro [271] is given in Figure 2.13.

$a_0 =$	2.50662823884	$b_0 =$	-8.47351093090
$a_1 =$	-18.61500062529	$b_1 =$	23.08336743743
$a_2 =$	41.39119773534	$b_2 =$	-21.06224101826
$a_3 =$	-25.44106049637	$b_3 =$	3.13082909833

$c_0 = 0.3374754822726147$	$c_5 = 0.0003951896511919$
$c_1 = 0.9761690190917186$	$c_6 = 0.0000321767881768$
$c_2 = 0.1607979714918209$	$c_7 = 0.0000002888167364$
$c_3 = 0.0276438810333863$	$c_8 = 0.0000003960315187$
$c_4 = 0.0038405729373609$	

Fig. 2.12. Constants for approximations to inverse normal.

Input: u between 0 and 1
Output: x, approximation to $\Phi^{-1}(u)$.
$y \leftarrow u - 0.5$
if $|y| < 0.42$
  $r \leftarrow y * y$
  $x \leftarrow y * (((a_3 * r + a_2) * r + a_1) * r + a_0)/$
    $((((b_3 * r + b_2) * r + b_1) * r + b_0) * r + 1)$
else
  $r \leftarrow u$;
  if $(y > 0)$ $r \leftarrow 1 - u$
  $r \leftarrow \log(-\log(r))$
  $x \leftarrow c_0 + r * (c_1 + r * (c_2 + r * (c_3 + r * (c_4 +$
    $r * (c_5 + r * (c_6 + r * (c_7 + r * c_8)))))))$
  if $(y < 0)$ $x \leftarrow -x$
return x

Fig. 2.13. Beasley-Springer-Moro algorithm for approximating the inverse normal.

Appendix B: The Runge-Kutta Method

Consider the set of ODEs represented by

$$\begin{aligned}\frac{\partial x_1(t)}{\partial t} &= f_1(t, x_1(t), ..., x_n(t)) \\ &\vdots \\ \frac{\partial x_n(t)}{\partial t} &= f_n(t, x_1(t), ..., x_n(t))\end{aligned}$$

Define $X(t) = (x_1(t), ..., x_n(t))'$ and $f = (f_1, ..., f_n)'$ and write the system as

$$\frac{\partial X(t)}{\partial t} = f(t, X(t))$$

The Runge-Kutta method then requires that we calculate the following values

$$\begin{aligned}k_1 &= hf\left(t, X(t_n)\right) \\ k_2 &= hf\left(t + \frac{1}{2}h, X(t_n) + \frac{1}{2}k_1\right) \\ k_3 &= hf\left(t + \frac{1}{2}h, X(t_n) + \frac{1}{2}k_2\right) \\ k_4 &= hf\left(t + h, X(t_n) + k_3\right)\end{aligned}$$

where $h = t_{n+1} - t_n$ is the step size of the discretized process. The Runga-Kutta approximation to the discretized process then satisfies

$$X(t_{n+1}) = X(t_n) + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right).$$