Numerical Methods for Financial Mathematics. **Exercise Handout 3**
Lecture: Prof. Dr. Christian Fries, Exercises: Dr. Andrea Mazzon, Tutorium: Roland Bachl
Sommersemester 2020

Exercise 1

Consider the class

`com.andreamazzon.session3.encapsulation.lazyinitialization.LinearCongruentialGenerator`

of the Java course project. We saw that due to overflows, the number

```
long congruence = (a * randomNumbers[indexOfInteger] + c) % modulus
```

can be negative, and to fix this problem we increment it by `modulus`. However, even if this still produce a (pseudo) random natural number smaller than `modulus`, this is not mathematically completely correct. To better understand the problem, consider 4 bits, giving 16 integers from -8 to $+7$. Take the modulus equal to 3. Here $8 \,\%\, 3 = 2$. But 8 corresponds to -8 due to the overflow, $(-8) \,\%\, 3 = -(8 \,\%\, 3) = -2$ and $-2 + 3 = 1$. So our method does not give what we would expect. Indeed, the operations *overflow* and % do not commute, and one has first to fix the overflow and then to apply the operation %.
Think about a good way to do this in the case when

$$\texttt{Long.MAX\_VALUE} < a{\cdot}\texttt{modulus} < 2{\cdot}\ \texttt{Long.MAX\_VALUE},$$

and correct the method `generate()` written in `LinearCongruentialGenerator` accordingly, changing the values of `modulus` and `a` so that the above condition holds. You can first copy and paste the class `LinearCongruentialGenerator` in your exercise project, and test it as in

`com.andreamazzon.session3.encapsulation.lazyinitialization.PseudoRandomNumbersTesting`.

Check, for example by Mathematica, that the natural number you get in the presence of an overflow (i.e., in the case when you would get a negative number if the overflow was not handled at all) is indeed

```
(a * previousRandomNumber + c) % modulus
```

where `previousRandomNumber` is the previous number in the list.
Note: possible values are `a = 6553590L`, `modulus = 2814749767110L`. Choosing `seed = 2814749763100L` in the test class, the value of the first random natural number must be `2788469871221`.

Exercise 2

Have a look at the interfaces

`MonteCarloEvaluationsInterface`

and

`MonteCarloEvaluationsWithExactResultInterface`.

you can find in the package

`com.andreamazzon.exercise3.montecarloevaluations`.

The first interface provides methods to experiment with the Monte-Carlo implementation of a possibly general class of problems, from pricing to the computation of an integral. The second interface extends the first one. For this second interface, we suppose that we already know the exact value of the quantity we approximate by Monte-Carlo, so that we can compute the absolute errors of our approximations.

Write two classes `MonteCarloIntegrationPowerFunction` and `MonteCarloPi` implementing this second interface and computing the integral $\int_0^1 x^\alpha dx$, $\alpha \in (0,\infty)$, in the case of `MonteCarloIntegrationPowerFunction`, and π in the way you have seen in the lecture in the case of `MonteCarloPi`.

Test then your results by running the classes

`com.andreamazzon.exercise3.montecarlopi.MonteCarloPi`

and

`com.andreamazzon.exercise3.montecarlointegration.MonteCarloIntegrationCheck`.

Hints:

- Here you have total freedom about the choice of your design. A suggestion is: you can note that the implementation of almost all the methods of the interfaces does not depend on the specific Monte-Carlo computation, as they might use the vectors hosting the computed values, just like the method `getPrice(StochasticProcessSimulatorInterface underlyingProcess)` in `DigitalOption`. You can think to implement these methods in some abstract class / abstract classes implementing the correspondent interfaces, and then let `MonteCarloIntegrationPowerFunction` and `MonteCarloPi` extend this class / one of these classes. In this way, they would automatically implement the interface.

- Consider that, with respect to the methods of the interface

  `com.andreamazzon.exercise2.MonteCarloExperiments`,

  the methods of the present interfaces do not take the number of Monte-Carlo computations as argument. So this should be a field of your classes. You can decide which access modifier to assign it according to the design of your solution (i.e., if you have or not the parent abstract classes, for example).

- You can find some methods to perform your experiments (for example, computing the standard deviation, or the average error, of the Monte-Carlo computations) in the class

  `com.andreamazzon.usefulmethodsmatricesandvectors.UsefulMethodsMatricesAndVectors`

  in the exercise project (so you don't have to pull the Java project again).