Numerical Methods for Financial Mathematics **Exercise Handout Java course**
Lecture: Prof. Dr. Christian Fries, Exercises: Dr. Andrea Mazzon, Tutorium: Niklas Weber
Sommersemester 2023

Exercise 1

Look at the class

`com.andreamazzon.handout0.ManagingArrayLists`.

As you can also get reading its Javadoc documentation, we can assume to fix a given set of equally-spaced times $T_0 = 0 < T_1 < ... < T_N$. Then the class has the main goal to return the set of all the possible vectors of exercise times $(T_k, T_{k+1}, ..., T_n)$, $n > k$, for a Bermudan option, when for every fixed entry time T_k the maturity $T_n > T_k$ is iteratively moved backward from T_N to T_{k+1}, in an inner for loop, and the entry time T_k is moved forward from 0 to T_{N-1}.

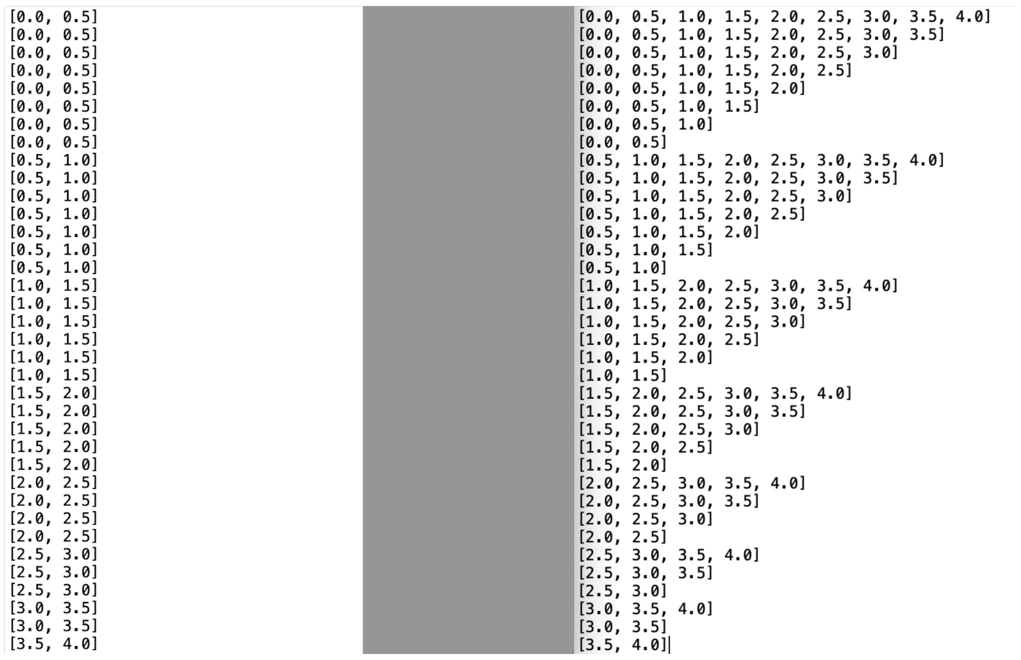
For example, what we want to get when the constant interval between one time to the other is $\Delta = 0.5$ and $T_N = 4$ is showed in the right part of Figure 1. However, the current implementation of the class has a bug, such that what we get running the class

`com.andreamazzon.handout0.TestManagingArrayLists`

in `src/test/java` is what is shown in the left of Figure 1.

Try to see what is wrong and to fix it: it is just about changing one line of

`com.andreamazzon.handout0.ManagingArrayLists`!



```
[0.0, 0.5]
[0.0, 0.5]
[0.0, 0.5]
[0.0, 0.5]
[0.0, 0.5]
[0.0, 0.5]
[0.0, 0.5]
[0.0, 0.5]
[0.5, 1.0]
[0.5, 1.0]
[0.5, 1.0]
[0.5, 1.0]
[0.5, 1.0]
[0.5, 1.0]
[0.5, 1.0]
[1.0, 1.5]
[1.0, 1.5]
[1.0, 1.5]
[1.0, 1.5]
[1.0, 1.5]
[1.0, 1.5]
[1.5, 2.0]
[1.5, 2.0]
[1.5, 2.0]
[1.5, 2.0]
[1.5, 2.0]
[2.0, 2.5]
[2.0, 2.5]
[2.0, 2.5]
[2.0, 2.5]
[2.5, 3.0]
[2.5, 3.0]
[2.5, 3.0]
[3.0, 3.5]
[3.0, 3.5]
[3.5, 4.0]
```

```
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5]
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5]
[0.0, 0.5, 1.0, 1.5, 2.0]
[0.0, 0.5, 1.0, 1.5]
[0.0, 0.5, 1.0]
[0.0, 0.5]
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5]
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
[0.5, 1.0, 1.5, 2.0, 2.5]
[0.5, 1.0, 1.5, 2.0]
[0.5, 1.0, 1.5]
[0.5, 1.0]
[1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
[1.0, 1.5, 2.0, 2.5, 3.0, 3.5]
[1.0, 1.5, 2.0, 2.5, 3.0]
[1.0, 1.5, 2.0, 2.5]
[1.0, 1.5, 2.0]
[1.0, 1.5]
[1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
[1.5, 2.0, 2.5, 3.0, 3.5]
[1.5, 2.0, 2.5, 3.0]
[1.5, 2.0, 2.5]
[1.5, 2.0]
[2.0, 2.5, 3.0, 3.5, 4.0]
[2.0, 2.5, 3.0, 3.5]
[2.0, 2.5, 3.0]
[2.0, 2.5]
[2.5, 3.0, 3.5, 4.0]
[2.5, 3.0, 3.5]
[2.5, 3.0]
[3.0, 3.5, 4.0]
[3.0, 3.5]
[3.5, 4.0]
```

Figure 1: Results we expect on the right, results we get on the left

Exercise 2

In the class `BinomialModelSimulator` of the package

`com.andreamazzon.session4.composition.binomialmodel`

of the Java course project we have seen an example of *composition*: an object of type `LinearCongruentialGenerator` is used to generate the pseudo random numbers on which the simulation of a binomial model is based. We can see that this class, adding few methods, can implement the interface `StochasticProcessSimulatorInterface` that you can find in the package `com.andreamazzon.handout0`

of the exercises project.

This interface has some methods that any class providing the simulation of a stochastic process must implement. However, the implementation of at least some methods does not depend on the specific process into consideration: that is, they can be implemented at a general level, in an abstract parent class that gets then extended by classes simulating the specific processes. These classes have to provide the implementation of the abstract method(s?) that are specific of the simulation of the process into consideration, and that will be called in the methods implemented instead directly in the parent class. Note that the abstract method(s) mentioned above can also not be part of the interface.

Try to do at least some of the following points:

(a) Identify the methods of `StochasticProcessSimulatorInterface` that in your opinion can be implemented in an abstract parent class and write this whole class, also declaring that it implements `StochasticProcessSimulatorInterface`.

(b) Rewrite `BinomialModelSimulator` in such a way that it extends the abstract class above, providing the implementation of its abstract method(s). You can of course use (also doing copy and paste) the code that you find in

`com.andreamazzon.session4.composition.binomialmodel`.

(c) Write a class `TrinomialModelSimulator`, which also extends your abstract class and simulates a trinomial model: the difference is that, at any time t_i, the process can stay the same, i.e., its value at t_{i+1} can be the same as in t_i. Note that in this case the market is not complete: there exist infinitely many equivalent martingale measures, basically identified by the probability that the process does not move at a given time. This probability can then be an argument in the constructor of the class.

(d) Write a class with a `main` method where you do some tests of your implementation.