Pierre VUILLEMIN

ENSAE
2014/2015

Report

# Numerical Methods In Financial Engineering

Subject: Barrier Options (parts 4.17.1 to 4.17.3, pages 152-160)


12 mai 2015

Professeurs: Marian CIUCA
Stéphane CRÉPEY

# Contents

# 1 First Project: Barrier Option Pricing

## 1.1 Introduction to the Barrier Options

Barrier options emerged towards 1967 on the OTC markets. The main idea is that the potentially infinite spot price in the models leads to high prices, especially with lookback options for which the payoff rely on the maximum reached over the remaining maturity. Thus adding a barrier which can deactivate the option by making it worthless, limits this phenomenon and provides lower prices. The principle is then fairly easy to understand. The price of a European barrier option is based on the value of the maximum (or minimum) of the spot price over the remaining maturity but also on the spot price at maturity.

European barrier options are specified by three properties: whether it is a call or put, whether the barrier is above or under the strike (up or down) and whether it activates or deactivates the contract (in or out). Thus there are eight types of contract. Let us present them for the call type, the put case works exactly the same.

1. The down-and-out call: the barrier is under the strike, if the spot price falls below the barrier (*i.e* its minimum is inferior to the barrier) the option becomes worthless. Otherwise, the payoff is the same as a basic call option.

2. The up-and-out call: the barrier is over the strike and the option becomes worthless if the maximum price reached over the period is over the barrier. Otherwise it has the same payoff as a call.

3. The down-and-in call: the barrier is under the strike and the option becomes worthless if the minimum price does not fall below the barrier over the period. In other words the price has to go down in first place in order to have a worthy option.

4. The up-and-in call: the barrier is over the strike and the option remains worthless if the maximum price does not reach the barrier before the maturity. Otherwise it has the same payoff as a call.

The more traded barrier options are the up-and-out call and the down-and-out put. Their behaviour is really close to the genuine calls and put. Although there are really popular and simple to understand, these options are difficult to hedge when they are close to the barrier. Indeed, the price can vary from the one of a frankly in-the-money call to zero really quickly. Thus the sensitivities and of course the delta behaves very badly near the barrier. Their main advantage is their prices, which is much lower than original call and put options.

There also exist other types of barrier options. For example, American barrier options, or barrier options on a basket of asset. But another important case is the double barrier options. These are based on the behaviour of the maximum, the minimum and the spot price at maturity. The most simple instance of it, is the down-and-out up-and-out call. This option becomes worthless if either the maximum reach the upper barrier or the minimum falls below the lowest one. Of course, many combinations exist for the double barrier options, but we will not cite them all here.

## 1.2 Our Pricer

### 1.2.1 Introduction

The aim of the first project was to be able to produce the prices of every option presented in the previous paragraph. To do so, we implemented the closed formed approximations collected by Espen Gaarder Haug in his book: "*The Complete Guide to Option Pricing Formulas*". The assignment was to reproduce the values provided by the book using a DLL C++/VBA Excel. A DLL (Dynamic Link Library) is as its name indicates it, a library of functions. This concept allows for example to compile functions in a fast programming language such as C++ in order to use them in a slower environment like Excel. The interest being to take advantage of a pleasant and user-friendly graphic interface but associated with the speed of a compiled programming language.

### 1.2.2 Users' guide

The computation of such a DLL is pretty straightforward. The first step consists in creating the DLL in the C++ IDE (Integrated Development Environment) such as Visual Studio. The main difference with a common project is the use of a ".def" file. This specific file allow the programmer to declare what functions he is going to export to the Excel spreadsheet. Once the program is compiled, a ".dll" is created instead of a ".exe". The advantage of this technique is its simplicity. However it might be troublesome since the debugger is not available. One can duplicate the project to get a "debbugable" console application or use another method such as the creation of an xll. The second step consists in declaring the functions from the newly created library into Excel. A few code lines are sufficient to do so. An important point about it, is how to give the proper address of the library. The most portable way of doing it, is to use only "libraryName.dll", and to put the corresponding file in the same folder as the ".xlsm" file. Unfortunately, the computer at the ENSAE are working with win32 and it might generate conflicts when the DLL is used on a x64 processor. Nevertheless, recompiling the DLL should allow a good use of it.

### 1.2.3 The results

Our program works really well with standard European barrier options. We also cling to the results of the books when it comes to the double barrier option. The negative point of our work is the third part of the assignment concerning American Options. Unfortunately, we did not manage to reproduce the Bjerksund-Stensland 2002 approximation for the American Call. So we could not reproduce the knock-out American call figures presented page 155 of the book. The code should be enclosed to the present report. If for one reason or another the corrector could not operate it properly, he would find some "proofs" of the results we got in the last three pages of the report.

# 2 The Second Project

The second project was about simulating CIR processes using two different discretization methods: Euler and Milstein. Nevertheless, if the computation of these simulations alone are not that hard, the aim was to really think the architecture of the program, its comments, etc. More than an mathematical assignment, it was more about a programming one. We present here the different elements of the architecture, which uses most of the tools taught in the C++ class of the ENSAE. We will try to stress these considerations throughout the following pages.

## 2.1 The base class: "StochasticProcesses"

The C++ is one of the most famous object oriented programming language, thus the whole project is divided in classes which are used in the main. We committed not to use stand alone functions in the code.

The first brick of the project is the base class called "StochasticProcesses". Here we consider a process defined on an interval $[0; T]$, the final goal being to calculate quantities such as:

$$\mathbb{E}\left[\exp\left(-\int_0^T (x_s + y_s)\mathrm{d}s\right)\right]$$

using Monte-Carlo techniques. From our view, the fundamentals of a stochastic process are as follow: its initial value and its final value. These two parameters are thus defined as protected members. In applied mathematics, a stochastic process is useless if it cannot be simulated. So here we consider that, every stochastic process (which will inherit from this class) must have member functions to allow its simulation. Thus we create four virtual member functions which will constrain the users to implement ways of simulating the processes. The class then become abstract.

The functions Scheme corresponds to the discretization scheme used to simulate the process. Later we will use the Euler explicit Scheme and the Milstein Scheme. But these algorithms require to know more about the process. So they are not implemented at this stage and will remain as pure virtual functions. Here we make the distinction between a function which will be used to simulate the process alone and one for situations when the process has to be simulated jointly with another one. In the first case we just need a starting point and a discretization step, the simulation can be conducted inside the instance. Nonetheless, when it comes to the simulation of two correlated processes, one has to take the brownian motion of the other process into account. Thus the simulation cannot be made inside the instance. This motives the overloading of the function Scheme.

Nevertheless, once we know how to discretize our process it is easy to simulate it or to compute the value of its integral over the interval $[0; T]$. Indeed, the simulation is the iteration for the discretization scheme from the initial value to the final value, in a predefined number of steps. Moreover, the approximation we get is a piecewise affine function. Hence the calculation of the integral is easy using the trapezoidal rule at each subdivision. Knowing

a scheme, these functions consists mainly in doing simulation of normal distribution and iterating the scheme. Thus they can be fully implemented directly in the base class. This combined to the dynamic polymorphism explained in the previous paragraph, makes the things really easy for a user eager to create a new type a process. In effect, the only things the user has to do is to redefine the constructor, give the good parameters and provide the scheme. Then he will be able to directly simulate and integrate his process. This architecture based on the inheritance and dynamic polymorphism appears to be completely relevant.

## 2.2 The individual processes

First and foremost, we start with the CIR class, which oviously inherits from the "StochasticProcesses". The article deals with those processes. Even though, they are not the most basic stochastic processes, we then focused on them. A CIR process can be specified with three members (protected, plus the initial value already declared in the base class). This class brings only a new constructor adapted to these particular processes, it enables us to factorize some code which could have come from different way of discretizing the process.

Then comes the two other classes meant to deal with CIR processes: EulerCIR and MilsteinCIR. Thanks to our previous work, once inherited from the CIR class, the only remaining job is to implement the scheme. This implementation is really straightforward from the article. This shows, the interest of thinking about the architecture. Although the choices we have made surely are not optimal, they allow us to add new processes to our collection really quickly.

## 2.3 Couple of processes

Here the aim was to compute the quantity presented in the previous page. To do so we must be able to simulate jointly two processes. The choice we made is to create a new class independent of the others which is designed to manipulate these couples knowing their correlations. Since we want to be able to handle all kind of stochastic processes we have used two "StochasticProcesses" and a double (the correlation) in our class. Here a problem arises, since the "StochasticProcesses" class is abstract, there cannot exist any instance of it. So either we reduce the scope of the class to instantiable objects such as EulerCIR or we have to find a way to overcome this obstacle. Since we wanted to keep the generality of the program, we found a way to manage this issue. The use of pointers make the code harder to understand and to write but allows us to handle every stochastic processes. Here again the dynamic polymorphism is really helpful. Then, using the above-descripted functions, it is fairly easy to calculate the integral.

## 2.4 Limits of our work

Unfortunately, the program does not produce the expected result. Everything compiles but a problem persists with the seed of the Pseudo Random Number Generator. Here we use the well known Mersenne twister, but the results we get is typically arising from the generator always giving the same sequence. The result is presented in the figure 4 at the end of this

document. Nonetheless different trials, made by setting manually the seed, led to results close to the one expected in the paper.

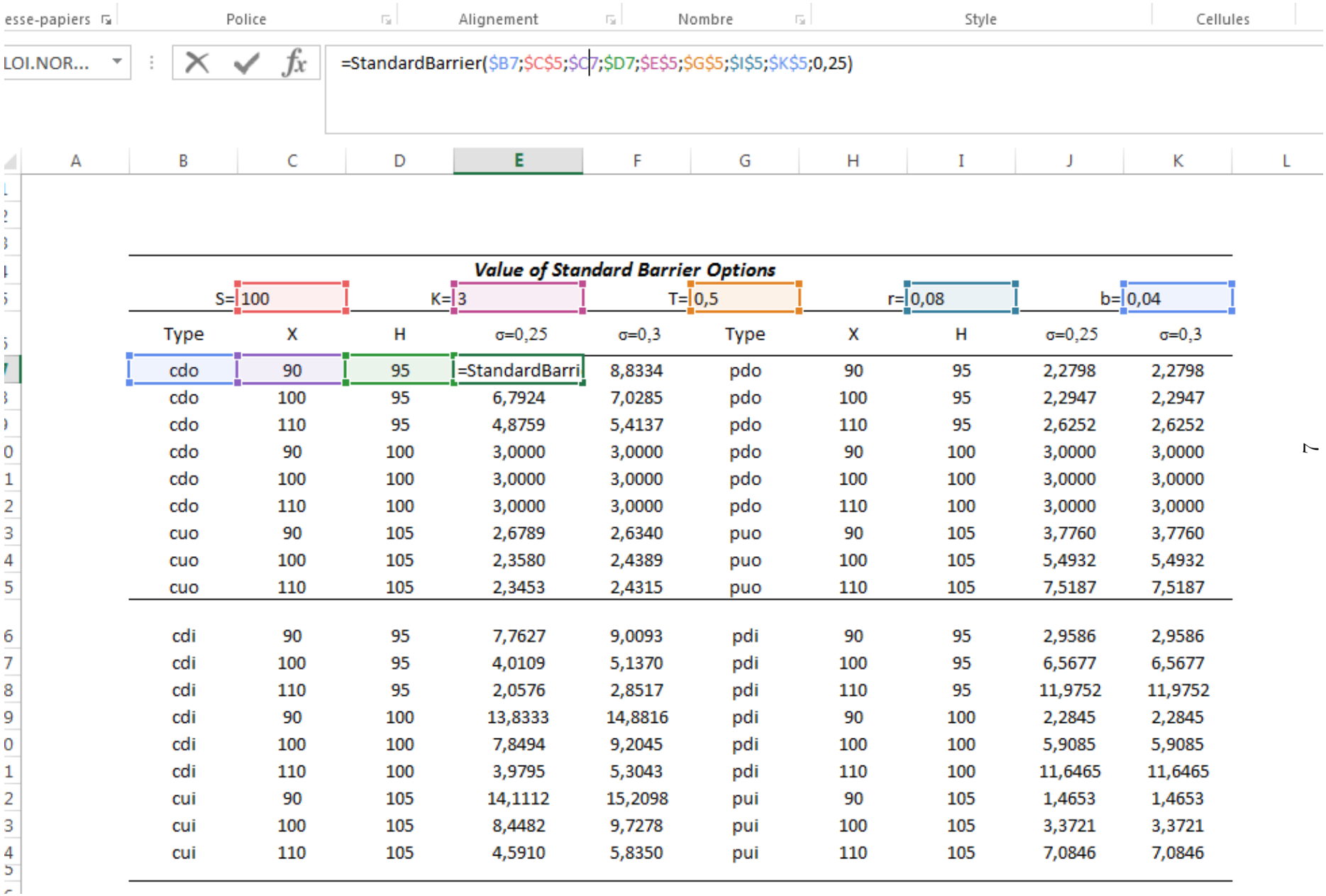


Figure 1: Screenshot of the results obtained for standard barrier options. The formula uses the function from the dll indeed

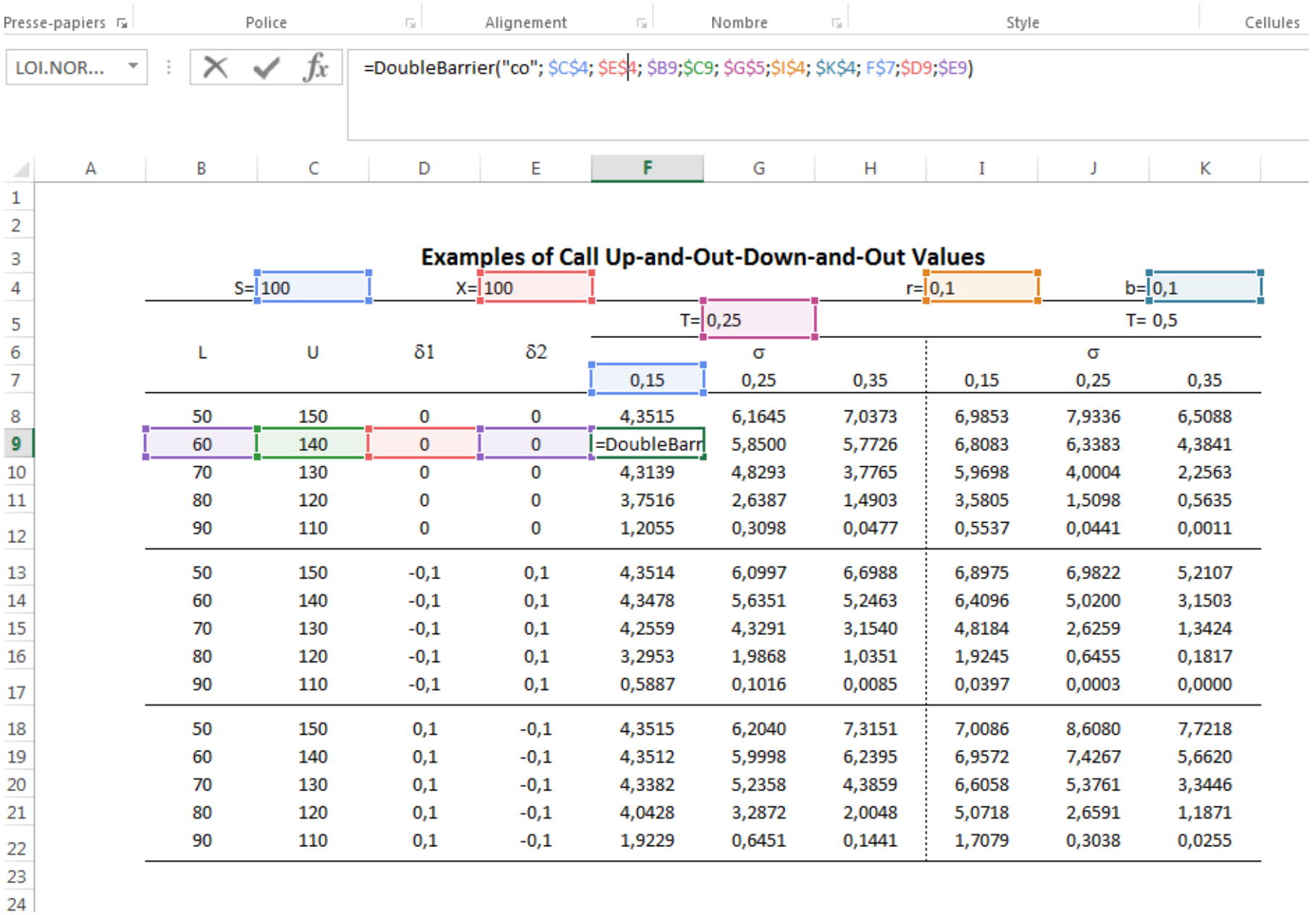


Figure 2: Screenshot of the results obtained for double barrier options. The formula uses the function from the dll indeed

**Examples of Call Up-and-Out-Down-and-Out Values**

| S= 100 | | X= 100 | | | | r= 0,1 | | | b= 0,1 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | T= 0,25 | | | | T= 0,5 | |
| L | U | δ1 | δ2 | σ | | | σ | | |
| | | | | 0,15 | 0,25 | 0,35 | 0,15 | 0,25 | 0,35 |
| 50 | 150 | 0 | 0 | 4,3515 | 6,1645 | 7,0373 | 6,9853 | 7,9336 | 6,5088 |
| 60 | 140 | 0 | 0 | 4,3505 | 5,8500 | 5,7726 | 6,8083 | 6,3383 | 4,3841 |
| 70 | 130 | 0 | 0 | 4,3139 | 4,8293 | 3,7765 | 5,9698 | 4,0004 | 2,2563 |
| 80 | 120 | 0 | 0 | 3,7516 | 2,6387 | 1,4903 | 3,5805 | 1,5098 | 0,5635 |
| 90 | 110 | 0 | 0 | 1,2055 | 0,3098 | 0,0477 | 0,5537 | 0,0441 | 0,0011 |
| 50 | 150 | -0,1 | 0,1 | 4,3514 | 6,0997 | 6,6988 | 6,8975 | 6,9822 | 5,2107 |
| 60 | 140 | -0,1 | 0,1 | 4,3478 | 5,6351 | 5,2463 | 6,4096 | 5,0200 | 3,1503 |
| 70 | 130 | -0,1 | 0,1 | 4,2559 | 4,3291 | 3,1540 | 4,8184 | 2,6259 | 1,3424 |
| 80 | 120 | -0,1 | 0,1 | 3,2953 | 1,9868 | 1,0351 | 1,9245 | 0,6455 | 0,1817 |
| 90 | 110 | -0,1 | 0,1 | 0,5887 | 0,1016 | 0,0085 | 0,0397 | 0,0003 | 0,0000 |
| 50 | 150 | 0,1 | -0,1 | 4,3515 | 6,2040 | 7,3151 | 7,0086 | 8,6080 | 7,7218 |
| 60 | 140 | 0,1 | -0,1 | 4,3512 | 5,9998 | 6,2395 | 6,9572 | 7,4267 | 5,6620 |
| 70 | 130 | 0,1 | -0,1 | 4,3382 | 5,2358 | 4,3859 | 6,6058 | 5,3761 | 3,3446 |
| 80 | 120 | 0,1 | -0,1 | 4,0428 | 3,2872 | 2,0048 | 5,0718 | 2,6591 | 1,1871 |
| 90 | 110 | 0,1 | -0,1 | 1,9229 | 0,6451 | 0,1441 | 1,7079 | 0,3038 | 0,0255 |

Figure 3: Screenshot of the results obtained for double barrier options.

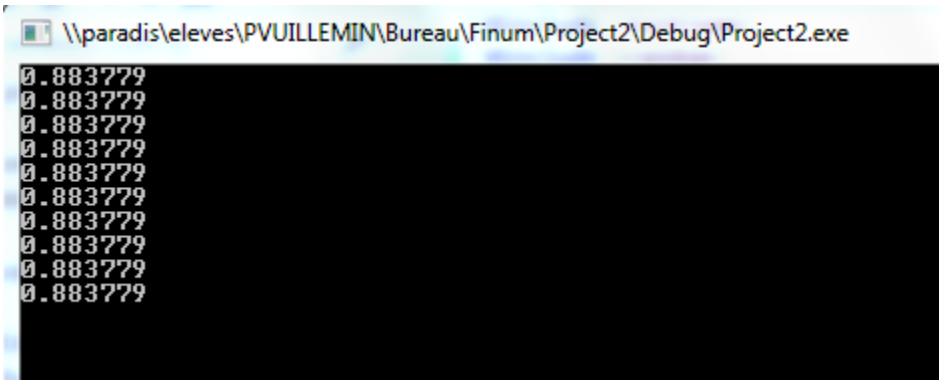


Figure 4: Screenshot of the result obtained with the second project.