Object Oriented Programming with Applications: Big Project.

Alejandro Medina Castro

January 2019

Introduction

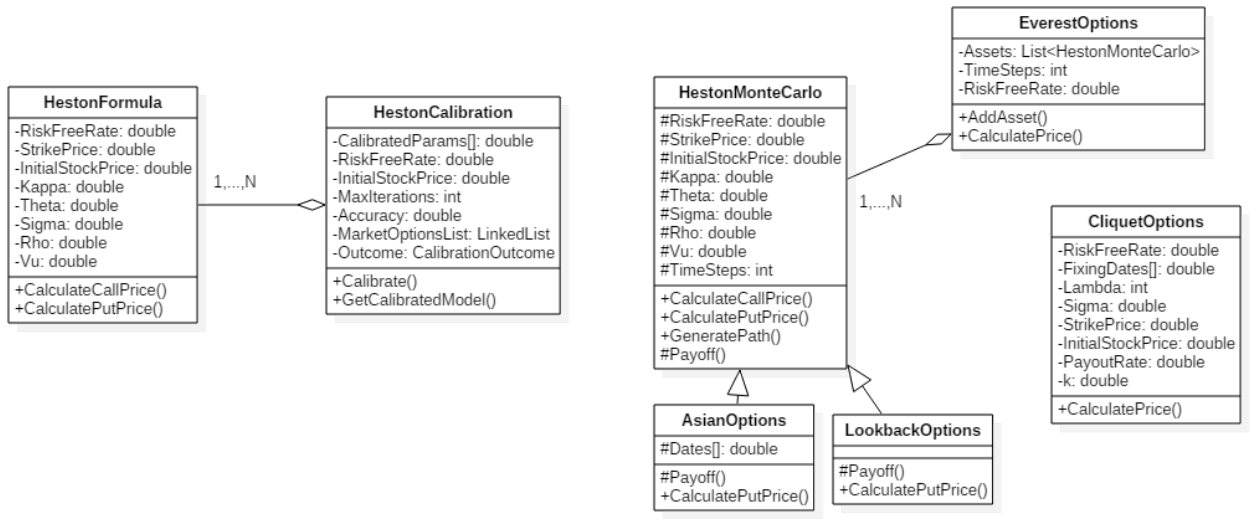
The purpose of the following report is to explain the structure and functionality of my implementation of the project [2], as well as to present the results obtained for the data provided.

Code Structure

Besides the projects already provided in the template, my implementation contains four other projects. Each has its own purpose:

- **FinalProject** is a class library that contains the classes that answer most of the questions of [2]. These include the HestonFormula, HestonMonteCarlo, HestonCalibration, AsianOptions and LookbackOptions.
- **FinalProjectTests** is a class library specific for testing. Here all the testing functionality is implemented. It only contains two classes: Tests and ConvergenceHestonMonteCarlo.
- **HestonCmlLine** is a console application whose only purpose is to showcase the functionality of FinalProject using the console.
- **OptionGUI** is a Windows Application that aims to provide a friendlier graphic interface for the user to interact with the classes of FinalProjects.

I chose to separate my classes by functionality to make the code easier to scale and understand. Let us now turn our attention to FinalProject because there is where most of the functionality is implemented. The internal structure of FinalProject is illustrated by the following class diagram:



We see that, in order to reduce code repetition, AsianOptions and LookbackOptions are sub-classes of HestonMonteCarlo. This makes sense since the only mayor changes for both are: the Payoff and CalculatePutPrice functions, we therefore make these methods virtual. This is very advantageous since now, given an option payoff function, we can price it without having to write much extra code..

For EverestOptions the situation is different. Given that we are pricing an option for a basket of assets, as oppose to a single asset, we create instances of HestonMonteCarlo for each asset. Finally, HestonCalibration uses instances of HestonFormula so therefore the association.

Results

Task 2.2 : As mentioned before, the entirety of the Heston Formula calculation is carried out in the HestonFormula class. The functions are written as in [1] to make understanding easier. Let us now present the results for task 2.2 by filling in Table 1:

Strike K	Option Exercise	Price
100	1	7.274
100	2	11.737
100	3	15.479
100	4	18.774
100	15	43.170

Table 1: Heston Formula

Task 2.3 : This task is entirely self-contained in the HestonMonteCarlo class, where the formulae from [1] is implemented. To generete the paths the MathNet.Numerics.Distributions library is used. Table 2 is filled using 10^5 different paths and 1000 time steps per year:

Strike K	Option Exercise	Price
100	1	13.6
100	2	22.6
100	3	29.9
100	4	36.6
100	15	75.7

Table 2: Monte Carlo

Task 2.4 : In order to check whether the Monte Carlo method was converging to the correct result, I attached the HestonFormula prices to Table 2 in order to compare:

Strike K	Option Exercise	Monte Carlo Price	Formula Price
100	1	13.6	13.630
100	2	22.6	22.452
100	3	29.9	29.996
100	4	36.6	36.655
100	15	75.7	78.416

Table 3: Monte Carlo vs. Formula

Then, as suggested in [2], I decided to plot the difference between both methods, aka the error, against the number of paths and time steps to see whether the error converged to zero as the others converged to infinity. These are the results:

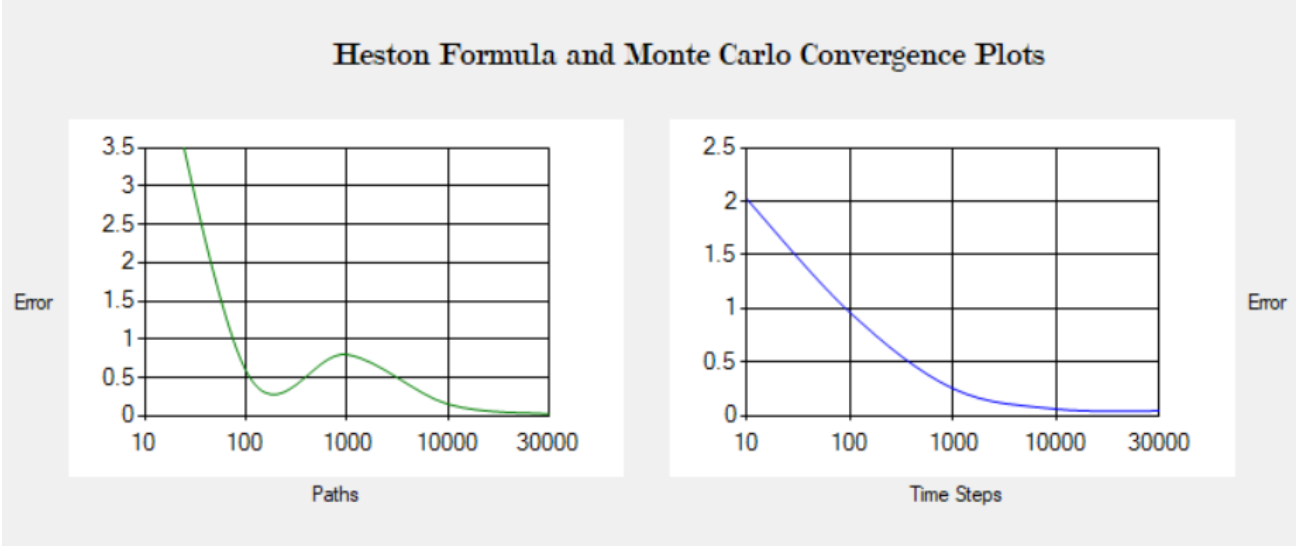


Figure 1: Paths vs. Error, Time Steps vs. Error

So the form of the curves do suggest convergence. This plots were made within the application and are available in the GUI.

Task 2.5 : The calibrator is implemented in the class HestonCalibration. It uses instances of the class HestonFormula. The class allows to set a guess parameter, if none is given then it has a default guess. Accuracy and the maximum number of iterations are also inputs. After running the calibrator with accuracy set to 10^{-3}, maximum iterations to 1000, the market data from Table 3 of [2] and the initial guess of: $\kappa = 0.5,\ \theta = 0.01,\ \sigma = 0.2,\ \rho = 0.2$ and $\nu = 0.4$; we get:

Error	Iterations	κ	θ	σ	ρ	ν
0.3264	16	0.5101	0.09241	0.2155	0.0463	0.1056

Table 4: Calibrated parameters.

Task 2.6 : In order to see whether the calibrator works, one must take the calibrated model and calculate the prices for the observations. The resulted prices should be roughly the same as the one in the observations. For the data provided the resulted prices are:

Strike K	Option Exercise	Price
80	1	25.44
90	1	18.89
80	2	30.13
100	2	19.55
100	1.5	16.86

Table 5: Prices with the calibrated parameters.

The main variables that determine whether the method converges are:

- Initial guess: A poor initial guess can make the method either fail badly, take a very long time to converge or go to a local minimum. Therefore we must choose a starting point that is not a special case and that is near the solution. A few examples of this are in the Table 6. They were calculated using the CheckingCalibration class.
- Step Size: If the differentiation step size is too large it will result in large truncation errors, while if it is too small it will result in large numerical errors.

Initial Guess	κ	θ	σ	ρ	ν	Result
	0	0	0	0	0	DivideByZeroException
	10	10	10	10	10	Error = 7786
	-1	-1	1	-1	-1	Error = 860, Iterations = 2

Table 6: Examples of the calibration not converging because of poor initial guess.

Step Size	Iterations	Error
10^{-1}	6	13.42
10^{-6}	16	0.3264
10^{-15}	10	8.38
10^{-18}	0	386.75

Table 7: Iterations and error for every step size.

Task 2.7 : This task is implemented in the class AsianOptions. It inherits almost all its functionality from HestonMonteCarlo. The only three added elements are: the array Dates and the overridden functions Payoff and CalculatePutPrice. The latter is because Put-Call parity does not hold in this case. The results for 10^5 paths, 365 time steps per year and the given parameters are:

Strike K	Option Exercise	$T_1, ..., T_M$	MC Price
100	1	0.75,1.00	11.9
100	2	0.25,0.50,0.75,1.00,1.25,1.50,1.75	11.2
100	3	1.00,2.00,3.00	19.5

Table 8: Asian call option prices

Task 2.8 : This task is implemented using the class LookbackOptions which inherits almost all its functionality from HestonMonteCarlo. Both CalculatePrice and CalculatePutPrice return the same price. The results for Table 5 of [2] using 10^5 paths 365 time steps per year and the given parameters are:

Option exercise T	MC Price
1	19.0
3	37.5
5	50.2
7	59.9
9	67.4

Table 9: Lookback option prices

The feller condition is taken care of in HestonMonteCarlo and therefore in all its subclases.

Exploration

Cliquet Options : First, I decided to look into tho complex problem of pricing Cliquet options without using Monte Carlo. A method for this is derived in [3]. I numerically implement Proposition 1 using the CliquetOptions class. In order to test the result, I checked whether the prices are consistent. The following table has parameters: K = 90, S = 100, $\sigma = 40\%$, r = 10% and Fixing Dates = [1 ,2 ,3].

Payout Rate δ	Call Price	Put Price
0	20.96	5.84
0.05	17.03	7.22
0.1	13.44	8.47
0.15	10.15	9.61
0.2	7.13	10.64

Table 10: Cliquet Option price evolution.

As you can see, when the payout is small, the call option has a significant greater price that the put option. That is consistent because the call option is the one that is in the money. The fact that the situation is reversed as the payout grows is also consistent, because a higher payout means more money coming out. The magnitude of the prices also makes sense since they are around 10 which is the payout of a European Option at maturity.

Everest Options : I decided to take on the challenge of pricing Everest Options using the Heston-MonteCarlo class that I had already implemented. The difficulty was in the fact that the underlying is a basket of asset. The way I did it was I created a list of HestonMonteCarlo in which each element represents an asset. For the payoff I used the following function:

$$\min_{i=1,...,n}(\frac{S_i^T}{S_i^0}) \tag{1}$$

Where $S_1, ..., S_n$ are n assets in a basket. So, in each iteration I create a path for each asset, then calculate the payoff for that set of paths and in the end average the results. Table 11 contains the data used in the test.

Asset	r	K	κ	θ	σ	ρ	ν	S
1	0.1	100	2	0.6	0.4	0.7	0.7	98
2	0.1	110	2.5	0.6	0.3	0.12	0.2	100
3	0.1	120	3.5	0.7	0.02	0.394	0.09	115

Table 11: Data used for the Everest Option test.

The Time Steps are 365 and the the number of paths is 10^4.

Exercise Date T	Everest Price
0.2	0.83872
0.5	0.69088
1	0.51849
5	0.11734
15	0.00690

Table 12: Everest Option price evolution with 3 assets.

We see the price goes down the higher the exercise date. This is because of the fact that the price deviates more from the initial price the longer the time passes. So, it just takes one of the assets to perform badly to render the entire option worthless. The amount of assets is also inversely proportional to the price because it increases the probability that one of the assets will perform badly. Our choice of the parameters obviously plays a role on S and therefore the price of the option. Let us now see the prices for a portfolio with six assets, where the parameters for each asset of Table 11 are represented twice in the portfolio:

Exercise Date T	Everest Price
0.2	0.77492
0.5	0.57820
1	0.38990
5	0.04687
15	0.00102

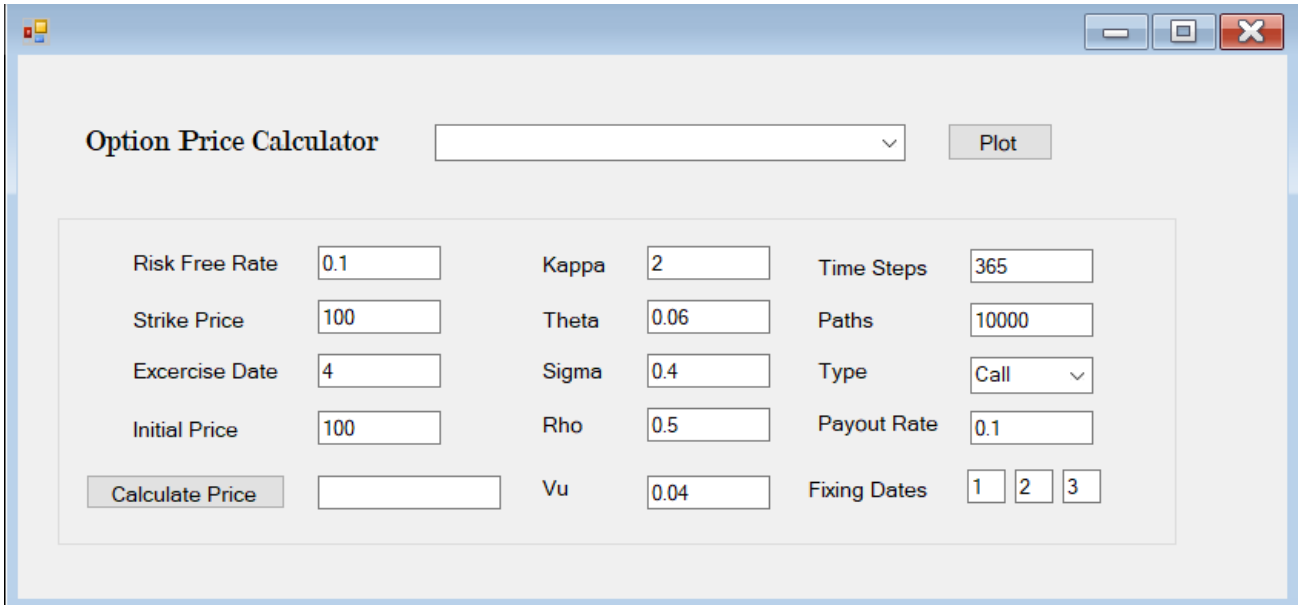
Table 13: Everest Option price evolution with 6 assets.

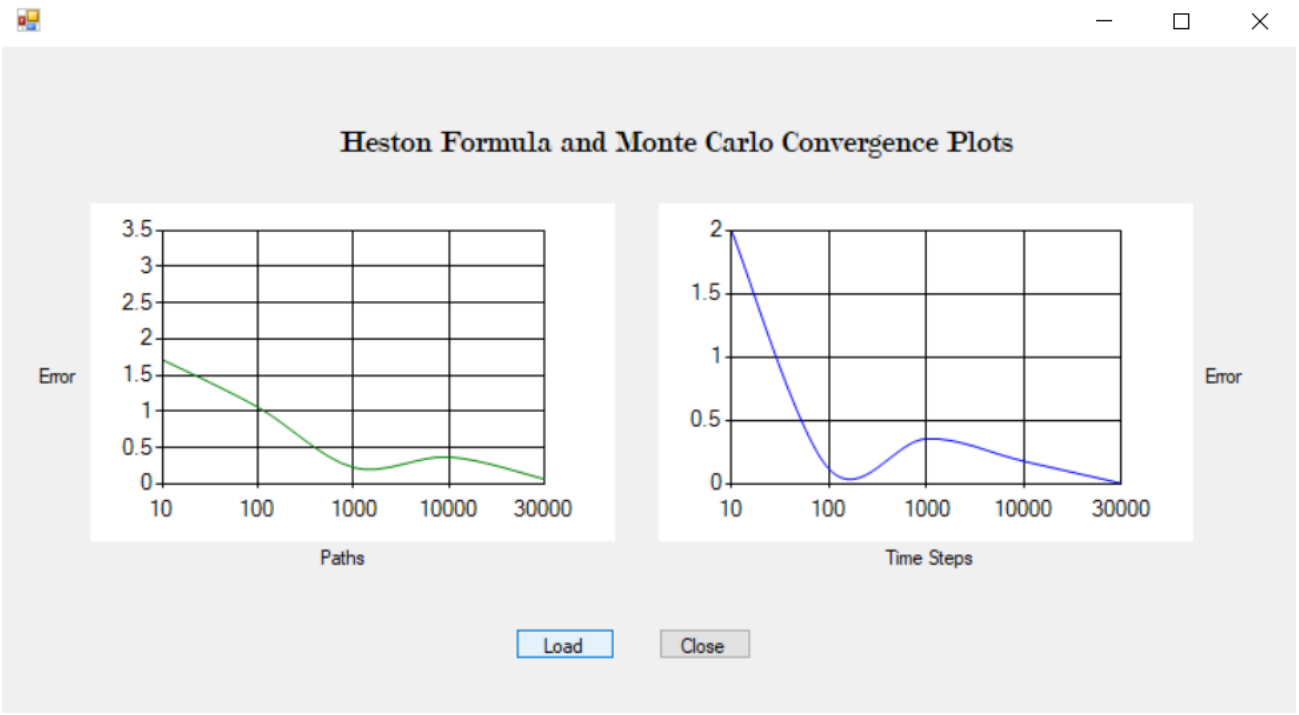
Parallel implementation of the Monte Carlo algorithm : In order to increase the efficiency of the Monte Carlo algorithm I decided to implement Multithreading for the path generation. It is safe to do so because Normal.Samples() uses a cryptographic random number generator. The point is that each thread creates a different path, calculates the payoff for that path and adds it to the accumulator. Interlocked makes sure that there is no data loss or a situation when 2 threads are locked waiting for the other to finish. The increase in efficiency for the first and third value of task 2.3 is shown in Table 14.

Linear Execution	Multithreading
18.19 s	5.35 s
52.08 s	12.62 s

Table 14: Efficiency improvement with multithreading

GraphicalUserInterface : I implemented a simple GUI utilizing the tools provided in Microsoft Visual Studio. This enables the user to calculate the price of European, Lookback and Cliquet options. For Europeans it can be done either by Formula or Monte Carlo. It also shows the result of task 2.4. The forms are shown below.





References

[1] Šiška D. *A note on the Heston model.* OOPA course website.

[2] Šiška D. *Object Oriented Programming with Applications Big Project.* OOPA course website.

[3] Tristan Guillaume. *A few insights into cliquet options.* International Journal of Business, 2012, 17 (2), pp.163-180. `https://hal.archives-ouvertes.fr/file/index/docid/924287/filename/A_few_insights_into_cliquet_options_TGuillaume.pdf`

Appendix

Attached code begins in the following page.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using MathNet.Numerics.Distributions;
7  using FinalProject;
8
9  namespace HestonCmdLine
10 {
11     /// <summary>
12     /// This class was created to experiment with the HestonCalibration Class to
          answer task 2.6.
13     /// It is in the Console Application so that the output can be seen.
14     /// </summary>
15     class CheckingCalibration
16     {
17         /// <summary>
18         /// This is the only method of the class. It initializes the data, the
            calibrator, calls the calibration function
19         /// and prints out all the results. This include the calibration
            outcome, error, Kappa, Theta, Sigma, Rho, Nu
20         /// and the approximated prices.
21         /// </summary>
22         public static void CheckCalibration()
23         {
24             double Error = 0;
25
26             double RiskFreeRate = 0.025;
27             double InitialStockPrice = 100;
28             double Kappa = 0.5;
29             double Theta = 0.01;
30             double Sigma = 0.2;
31             double Rho = 0.1;
32             double Nu = 0.4;
33             double Accuracy = 0.001;
34             int MaxIterations = 1000;
35             double[] StrikePrices = new double[] { 80, 90, 80, 100, 100 };
36             double[] OptionExerciseTimes = new double[] { 1, 1, 2, 2, 1.5 };
37             double[] Prices = new double[] { 25.72, 18.93, 30.49, 19.36,
                16.58 };
38
39             Console.WriteLine("RiskFreeRate: {0} -- InitialStockPrice: {1} --
                Kappa: {2} -- Theta {3} -- Sigma {4} -- Rho {5} -- Nu {6}"
40              , RiskFreeRate, InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu);
41
42             HestonCalibration Calibrator = new HestonCalibration(RiskFreeRate,
                InitialStockPrice, Accuracy, MaxIterations);
43
```

```
44              Calibrator.SetGuessParameters(Kappa, Theta, Sigma, Rho, Nu);
45
46              for (int i = 0; i < Prices.Length; ++i)
47              {
48                  Calibrator.AddObservedOption(OptionExerciseTimes[i],
                      StrikePrices[i], Prices[i]);
49              }
50              Calibrator.Calibrate();
51              CalibrationOutcome Outcome = CalibrationOutcome.NotStarted;
52              Calibrator.GetCalibrationStatus( ref Outcome, ref Error);
53
54              HestonFormula CalibratedModel = Calibrator.GetCalibratedModel();
55              double[] Param = CalibratedModel.ConvertCalibrationParamsToArray();
56              double Price;
57
58              Console.WriteLine("RiskFreeRate: {0} -- InitialStockPrice: {1} --
                  Kappa: {2} -- Theta {3} -- Sigma {4} -- Rho {5} -- Nu {6}"
59                  , RiskFreeRate, InitialStockPrice, Param[0], Param[1], Param[2],
                       Param[3], Param[4]);
60              for (int i = 0; i < Prices.Length; i++)
61              {
62                  Price = CalibratedModel.CalculateCallPrice(StrikePrices[i],
                      OptionExerciseTimes[i]);
63                  Console.WriteLine(" {3} -- Strike {0} -- Exercise {1} -- Price
                      {2}", StrikePrices[i], OptionExerciseTimes[i], Price, i);
64              }
65              Console.WriteLine("Calibration outcome: {0} and error: {1}",
                  Outcome, Error);
66
67
68
69
70
71              Console.ReadKey();
72          }
73
74
75
76      }
77  }
78
```

```
1  using System;
2  using MathNet.Numerics.Integration;
3  using MathNet.Numerics;
4
5  namespace FinalProject
6  {
7
8      /// <summary>
9      /// Class for pricing Cliquet Options using the Guillaume Model.
10     /// <see href="https://hal.archives-ouvertes.fr/file/index/docid/924287/
         filename/A_few_insights_into_cliquet_options_TGuillaume.pdf">Reference</
         see>
11     /// </summary>
12     /// <remarks>
13     /// This result is not in the Heston model framework.
14     /// </remarks>
15     public class CliquetOptions
16     {
17         private double RiskFreeRate { get; set; }
18         private double[] FixingDates { get; set; }
19         private int Lambda { get; set; }
20         private double Sigma { get; set; }
21         private double StrikePrice { get; set; }
22         private double InitialStockPrice { get; set; }
23         private double PayoutRate { get; set; }
24         private readonly double k;
25
26
27         /// <summary>
28         /// Constructor for the Cliquet class.
29         /// </summary>
30         /// <param name="RiskFreeRate"></param>
31         /// <param name="FixingDates">The dates that are used for calculating
             the price.</param>
32         /// <param name="Lambda"> Lambda represents whether the option is put
             or call.</param>
33         /// <param name="Sigma">Volatility.</param>
34         /// <param name="StrikePrice"></param>
35         /// <param name="InitialStockPrice"></param>
36         /// <param name="PayoutRate">The rate by which shareholders are paid.</
             param>
37         /// <exception cref="ArgumentException">Thrown when one of the
38         /// arguments provided to a method is not valid.</exception>
39         public CliquetOptions(
40         double RiskFreeRate,
41         double[] FixingDates,
42         int Lambda,
43         double Sigma,
44         double StrikePrice,
```

```
45          double InitialStockPrice,
46          double PayoutRate
47              )
48          {
49              this.RiskFreeRate = RiskFreeRate;
50              this.FixingDates = FixingDates;
51              this.Lambda = Lambda;
52              this.Sigma = Sigma;
53              this.StrikePrice = StrikePrice;
54              this.InitialStockPrice = InitialStockPrice;
55              this.PayoutRate = PayoutRate;
56
57              for (int i=0; i<FixingDates.Length; i++)
58              {
59                  if (FixingDates[i] <= 0) throw new ArgumentException("---
                      Error: Dates must be positive. ---");
60
61              }
62
63              if (InitialStockPrice == 0) throw new ArgumentException("--- Error:
                    Initial Stock Price con not be 0. ---");
64
65              k = Math.Log(StrikePrice / InitialStockPrice);
66          }
67
68          private double Mu(int i)
69          {
70              return (RiskFreeRate - Sigma * Sigma / 2) * FixingDates[i];
71          }
72
73          private double MuHat(int i)
74          {
75              return (RiskFreeRate + Sigma * Sigma / 2) * FixingDates[i];
76          }
77
78          private double Mu(int i,int j)
79          {
80              return (RiskFreeRate - Sigma * Sigma / 2) * (FixingDates[j] -
                  FixingDates[i]);
81          }
82
83          private double MuHat(int i,int j)
84          {
85              return (RiskFreeRate + Sigma * Sigma / 2) * (FixingDates[j] -
                  FixingDates[i]);
86          }
87
88          private double SigmaFunction(int i)
89          {
```

```
90              return Sigma * Math.Sqrt(FixingDates[i]);
91          }
92
93          private double SigmaFunction(int i, int j)
94          {
95              return Sigma * Math.Sqrt(FixingDates[j] - FixingDates[i]);
96          }
97
98          private double Beta(int i, int j)
99          {
100             return Math.Sqrt(FixingDates[i] / FixingDates[j]);
101         }
102
103         private double Rho(int i, int j)
104         {
105             return Math.Sqrt(1 - FixingDates[i] / FixingDates[j]);
106         }
107
108         private double Rho(int i, int j, int m, int n)
109         {
110             return Math.Sqrt((FixingDates[j] - FixingDates[i]) / (FixingDates
                  [n] - FixingDates[m]));
111         }
112
113         private double N(double x)
114         {
115             return Math.Exp(-x * x / 2) / Math.Sqrt(2 * Math.PI);
116         }
117
118         private double Phi1(double a)
119         {
120             Func<double, double> TheFunction = t => Math.Exp(-t * t / 2);
121
122             double Integral = SimpsonRule.IntegrateComposite(TheFunction, -100,
                    a, 100);
123
124             return Integral / Math.Sqrt(2 * Math.PI);
125         }
126
127         private double Phi2(double a1, double a2, double Theta)
128         {
129             Func<double, double, double> TheFunction = (x, y) =>
130             Math.Exp(-(x * x + y * y - 2 * Theta * x * y) / (2 * (1 - Theta *
                  Theta))) /
131             (2 * Math.PI * Math.Sqrt(1 - Theta * Theta));
132
133             double Integral = Integrate.OnRectangle(TheFunction, -100, a1,
                  -100, a2);
134             return Integral;
```

```
135         }
136
137         private double Phi3(double b1, double b2, double b3, double Theta1,
              double Theta2)
138         {
139
140             Func<double, double> TheFunction = x =>
141             Math.Exp(-x * x / 2) / Math.Sqrt(2 * Math.PI) *
142             N((b1 - Theta1 * x) / Math.Sqrt(1 - Theta1 * Theta1)) *
143             N((b3 - Theta2 * x) / Math.Sqrt(1 - Theta2 * Theta2));
144
145             double Integral = SimpsonRule.IntegrateComposite(TheFunction, -100,
                b2, 100);
146             return Integral;
147         }
148         /// <summary>
149         /// This is where all the functions come together to produce the final
              price of the option.
150         /// </summary>
151         /// <returns>
152         /// The price of the option as a double precision number.
153         /// </returns>
154         /// <remarks>
155         /// The double integrals are achieved using
              MathNet.Numerics.Integration.OnRectagle.
156         /// </remarks>
157         public double CalculatePrice()
158         {
159             double Aux1 = Lambda * Math.Exp(-RiskFreeRate * (FixingDates[2] -
                FixingDates[0]) - PayoutRate * FixingDates[0]);
160
161             double Aux2 =  InitialStockPrice * Phi1(Lambda * ( -k + MuHat(0)) /
                SigmaFunction(0));
162
163             double Aux3 = Phi2(-Lambda * Mu(0, 1) / SigmaFunction(0, 1), -
                Lambda * Mu(0, 2) / SigmaFunction(0, 2), Rho(0, 1, 0, 2));
164
165             double Aux4 = -Lambda * Math.Exp(-RiskFreeRate * FixingDates[2]) *
                StrikePrice * Phi1(Lambda * (-k + Mu(0)) / SigmaFunction(0));
166
167             double Aux5 = Phi2(Lambda * -Mu(0, 1) / SigmaFunction(0, 1), Lambda
                 * -Mu(0, 2) / SigmaFunction(0, 2), Rho(0, 1, 0, 2));
168
169             double Aux6 = Lambda * Math.Exp(-RiskFreeRate * (FixingDates[2] -
                FixingDates[1]) - PayoutRate * FixingDates[1]);
170
171             double Aux7 = InitialStockPrice * Phi1(Lambda *  -Mu(1, 2) /
                SigmaFunction(1, 2));
172
```

```
173             double Aux8 = Phi2(Lambda * (k - MuHat(0)) / SigmaFunction(0),
                  Lambda * (-k + MuHat(1)) / SigmaFunction(1), -Beta(0, 1));
174
175             double Aux9 = Phi1(Lambda * (-k + MuHat(0)) / SigmaFunction(0)) *
                  Phi1(Lambda * MuHat(0, 1) / SigmaFunction(0, 1));
176
177             double Aux10 = -Lambda * Math.Exp(-RiskFreeRate * FixingDates[2]) *
                   StrikePrice * Phi1(Lambda * -Mu(1, 2) / SigmaFunction(1, 2));
178
179             double Aux11 = Phi2(Lambda * (k - Mu(0)) / SigmaFunction(0), Lambda
                   * (-k + Mu(1)) / SigmaFunction(1), -Beta(0, 1));
180
181             double Aux12 = Phi1(Lambda * (-k + Mu(0)) / SigmaFunction(0)) *
                  Phi1(Lambda * Mu(0, 1) / SigmaFunction(0, 1));
182
183             double Aux13 = Lambda * Math.Exp(-PayoutRate * FixingDates[2]) *
                  InitialStockPrice;
184
185             double Aux14 = Phi3(Lambda * (-k + MuHat(2)) / SigmaFunction(2),
                  Lambda * MuHat(0, 2) / SigmaFunction(0, 2), Lambda * MuHat(1,
                  2) / SigmaFunction(1, 2), Rho(0, 2), Rho(1, 2, 0, 2));
186
187             double Aux15 = -Lambda * Math.Exp(-RiskFreeRate * FixingDates[2]) *
                   StrikePrice;
188
189             double Aux16 = Phi3(Lambda * (-k + Mu(2)) / SigmaFunction(2),
                  Lambda * Mu(0, 2) / SigmaFunction(0, 2), Lambda * Mu(1, 2) /
                  SigmaFunction(1, 2), Rho(0, 2), Rho(1, 2, 0, 2));
190
191             return Aux1 * Aux2 * Aux3 + Aux4 * Aux5 + Aux6 * Aux7 * (Aux8 +
                  Aux9) + Aux10 * (Aux11 + Aux12) + Aux13 * Aux14 + Aux15 * Aux16;
192         }
193
194     }
195 }
196
```

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using FinalProject;
7
8  namespace FinalProjectTests
9  {
10     /// <summary>
11     /// The purpose of this class is to provide the functions that will be used
          to plot the error between the prices generated
12     /// by the Monte Carlo and the Heston formula algorithms.
13     /// </summary>
14     public class ConvergenceHestonMonteCarlo
15     {
16         private readonly double RiskFreeRate;
17         private readonly double StrikePrice;
18         private readonly double OptionExercise;
19         private readonly double Kappa;
20         private readonly double Theta;
21         private readonly double Sigma;
22         private readonly double Rho;
23         private readonly double InitialStockPrice;
24         private readonly double Nu;
25
26         /// <summary>
27         /// Constructor for the class. All values can be set by default.
28         /// </summary>
29         /// <param name="RiskFreeRate"></param>
30         /// <param name="StrikePrice"></param>
31         /// <param name="OptionExercise"></param>
32         /// <param name="Kappa"></param>
33         /// <param name="Theta"></param>
34         /// <param name="Sigma"></param>
35         /// <param name="Rho"></param>
36         /// <param name="InitialStockPrice"></param>
37         /// <param name="Nu"></param>
38         /// <remarks>We used the same data provided in task 2.3.</remarks>
39         public ConvergenceHestonMonteCarlo(
40             double RiskFreeRate = 0.1,
41             double StrikePrice = 100,
42             double OptionExercise = 1,
43             double Kappa = 2,
44             double Theta = 0.06,
45             double Sigma = 0.4,
46             double Rho = 0.5,
47             double InitialStockPrice = 100,
48             double Nu = 0.04)
```

```
49          {
50              this.RiskFreeRate = RiskFreeRate;
51              this.StrikePrice = StrikePrice;
52              this.OptionExercise = OptionExercise;
53              this.Kappa = Kappa;
54              this.Theta = Theta;
55              this.Sigma = Sigma;
56              this.Rho = Rho;
57              this.InitialStockPrice = InitialStockPrice;
58              this.Nu = Nu;
59
60          }
61          /// <summary>
62          /// Given an array of paths it calculates the error for each element of
                the array.
63          /// </summary>
64          /// <param name="X">The array of paths.</param>
65          /// <param name="TimeSteps"></param>
66          /// <returns>An array that represents the error between the
              HestonMonteCarlo price and the HestonFormula price.</returns>
67          public double[] DataPointsPath(int[] X, int TimeSteps = 356)
68          {
69              double[] Y = new double[X.Length];
70
71              for (int i=1; i<X.Length; i++)
72              {
73                  Y[i] = CalculateError(X[i], TimeSteps);
74              }
75              return Y;
76
77          }
78          /// <summary>
79          /// Given an array of time steps it calculates the error for each
              element of the array.
80          /// </summary>
81          /// <param name="X">the array of time steps</param>
82          /// <param name="Paths"></param>
83          /// <returns>An array that represents the error between the
              HestonMonteCarlo price and the HestonFormula price.</returns>
84          public double[] DataPointsTimeSteps(int[] X, int Paths = 10000)
85          {
86              double[] Y = new double[X.Length];
87
88              for (int i = 1; i < X.Length; i++)
89              {
90                  Y[i] = CalculateError(Paths, X[i]);
91              }
92              return Y;
93
```

```
94          }
95
96          /// <summary>
97          /// Calculates the error for a single instance of the HestonMonteCarlo
              and the HestonFormula.
98          /// </summary>
99          /// <param name="Paths"></param>
100         /// <param name="TimeSteps"></param>
101         /// <returns>The error for the price of the HestonMonteCarlo and the
              HestonFormula.</returns>
102         public double CalculateError(int Paths, int TimeSteps)
103         {
104             HestonFormula Formula = new HestonFormula(RiskFreeRate,
                  InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu);
105             HestonMonteCarlo MonteCarlo = new HestonMonteCarlo(RiskFreeRate,
                  StrikePrice, InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu,
                  TimeSteps);
106             double PriceHestonFormula = Formula.CalculateCallPrice(StrikePrice,
                   OptionExercise);
107             double PriceMonteCarlo = MonteCarlo.CalculatePrice(OptionExercise,
                  Paths);
108             return Math.Abs(PriceHestonFormula - PriceMonteCarlo);
109         }
110
111
112     }
113 }
114
```

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using HestonModel;
7  using HestonModel.Interfaces;
8  using HestonModel.TheClasses;
9
10 namespace HestonCmdLine
11 {
12     /// <summary>
13     /// This class exists to test whether the set up was done correctly in the
         HestonModel project. It calls all its functions
14     /// and prints out the results.
15     /// </summary>
16     public class CorrectSetUp
17     {
18         /// <summary>
19         /// This function tests the first function of the Heston class and
             prints the results.
20         /// </summary>
21         public static void CalibrationSetUp()
22         {
23             double RiskFreeRate = 0.025;
24             double InitialStockPrice = 100;
25             double Kappa = 0.5;
26             double Theta = 0.01;
27             double Sigma = 0.2;
28             double Rho = 0.1;
29             double Nu = 0.4;
30             double Accuracy = 0.001;
31             int MaxIterations = 1000;
32
33
34             double[] StrikePrices = new double[] { 80, 90, 80, 100, 100 };
35             double[] OptionExerciseTimes = new double[] { 1, 1, 2, 2, 1.5 };
36             double[] Prices = new double[] { 25.72, 18.93, 30.49, 19.36,
                 16.58 };
37
38             //These are implementation of interfaces that are necessary to
                 interact with the HestonModel project.
39             EuropeanOption TheOption;
40             OptionMarketData Data;
41             HestonEnumerable2 Enum_ = new HestonEnumerable2();
42
43
44             for (int i = 0; i < StrikePrices.Length; i++)
45             {
```

```
46                  TheOption = new EuropeanOption(StrikePrices[i],
                      OptionExerciseTimes[i], HestonModel.PayoffType.Call);
47                  Data = new OptionMarketData(TheOption, Prices[i]);
48                  Enum_[i] = Data;
49
50              }
51
52              VarianceProcessParameters VProcessParameters = new
                  VarianceProcessParameters(Kappa, Theta, Sigma, Nu, Rho);
53              HestonModelParameters guessModelParameters = new
                  HestonModelParameters(InitialStockPrice, RiskFreeRate,
                  VProcessParameters);
54              CalibrationSettings Settings = new CalibrationSettings(Accuracy,
                  MaxIterations);
55
56              HestonCalibrationResult Result = Heston.CalibrateHestonParameters
                  (guessModelParameters,Enum_, Settings);
57
58              Console.WriteLine(Result.Parameters.VarianceParameters.Kappa);
59              Console.WriteLine(Result.Parameters.VarianceParameters.Rho);
60              Console.WriteLine(Result.Parameters.VarianceParameters.Sigma);
61              Console.WriteLine(Result.Parameters.VarianceParameters.Theta);
62              Console.WriteLine(Result.Parameters.VarianceParameters.V0);
63              Console.WriteLine(Result.PricingError);
64              Console.ReadKey();
65          }
66          /// <summary>
67          /// This functions tests the second function of the class Heston and
                prints out the results.
68          /// </summary>
69          public static void FormulaSetUp()
70          {
71              double RiskFreeRate = 0.025;
72              double StrikePrice = 100;
73              double[] OptionExercise = { 1, 2, 3, 4, 15 };
74              double Kappa = 1.5768;
75              double Theta = 0.0398;
76              double Sigma = 0.5751;
77              double Rho = -0.5711;
78              double InitialStockPrice = 100;
79              double Nu = 0.0175;
80              VarianceProcessParameters VProcessParameters = new
                  VarianceProcessParameters(Kappa, Theta, Sigma, Nu, Rho);
81              EuropeanOption europeanOption = new EuropeanOption(StrikePrice,
                  OptionExercise[4], HestonModel.PayoffType.Call);
82              HestonModelParameters parameters = new HestonModelParameters
                  (InitialStockPrice, RiskFreeRate, VProcessParameters);
83              double Price = Heston.HestonEuropeanOptionPrice(parameters,
                  europeanOption);
```

```
84              Console.WriteLine(Price);
85              Console.ReadKey();
86          }
87          /// <summary>
88          /// This function tests the third function in the Heston class and
              prints out the results.
89          /// </summary>
90          public static void MonteCarloSetUp()
91          {
92              double RiskFreeRate = 0.1;
93              double StrikePrice = 100;
94              double[] OptionExercise = { 1, 2, 3, 4, 15 };
95              double Kappa = 2;
96              double Theta = 0.06;
97              double Sigma = 0.4;
98              double Rho = 0.5;
99              double InitialStockPrice = 100;
100             double Nu = 0.04;
101             int TimeSteps = 1000;
102             int Paths = 10000;
103             double Price= 0;
104             for(int i = 0; i < OptionExercise.Length; i++)
105             {
106                 VarianceProcessParameters VProcessParameters = new
                      VarianceProcessParameters(Kappa, Theta, Sigma, Nu, Rho);
107                 HestonModelParameters parameters = new HestonModelParameters
                      (InitialStockPrice, RiskFreeRate, VProcessParameters);
108                 EuropeanOption europeanOption = new EuropeanOption(StrikePrice,
                       OptionExercise[i], HestonModel.PayoffType.Call);
109                 MonteCarloSettings settings = new MonteCarloSettings(Paths,
                      TimeSteps);
110                 Price = Heston.HestonEuropeanOptionPriceMC(parameters,
                      europeanOption, settings);
111                 Console.WriteLine(Price);
112             }
113
114             Console.ReadKey();
115         }
116         /// <summary>
117         /// This function tests the fourth function in the Heston class and
              prints out the results.
118         /// </summary>
119         public static void AsianOptionsSetUp()
120         {
121             double RiskFreeRate = 0.1;
122             double StrikePrice = 100;
123             double[] OptionExercise = { 1, 2, 3 };
124             double[] Aux1 = { 0.75, 1 };
125             double[] Aux2 = { 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75 };
```

```
126             double[] Aux3 = { 1, 2, 3 };
127             double[][] Dates = { Aux1, Aux2, Aux3 };
128             double Kappa = 2;
129             double Theta = 0.06;
130             double Sigma = 0.4;
131             double Rho = 0.5;
132             double InitialStockPrice = 100;
133             double Nu = 0.04;
134             int TimeSteps = 365;
135             int Paths = 10000;
136
137
138
139             for (int i = 0; i < OptionExercise.Length; i++)
140             {
141                 HestonEnumerable Enume = new HestonEnumerable(Dates[i]);
142                 VarianceProcessParameters VProcessParameters = new
                      VarianceProcessParameters(Kappa, Theta, Sigma, Nu, Rho);
143                 HestonModelParameters parameters = new HestonModelParameters
                      (InitialStockPrice, RiskFreeRate, VProcessParameters);
144                 AsianOption Option = new AsianOption(StrikePrice,
                      OptionExercise[i], Enume, PayoffType.Call);
145                 MonteCarloSettings settings = new MonteCarloSettings(Paths,
                      TimeSteps);
146                 double Price = Heston.HestonAsianOptionPriceMC(parameters,
                      Option, settings);
147                 Console.WriteLine(Price);
148             }
149
150             Console.ReadKey();
151
152         }
153         /// <summary>
154         /// This function tests the fifth function in the Heston class and
              prints out the results.
155         /// </summary>
156         public static void LookbackOptionSetUp()
157         {
158             double RiskFreeRate = 0.1;
159             double[] OptionExercise = { 1, 3, 5, 7, 9 };
160             double Kappa = 2;
161             double Theta = 0.06;
162             double Sigma = 0.4;
163             double Rho = 0.5;
164             double InitialStockPrice = 100;
165             double Nu = 0.04;
166             int TimeSteps = 365;
167             int Paths = 10000;
168             for (int i=0; i<OptionExercise.Length;i++)
```

```
169             {
170                 VarianceProcessParameters VProcessParameters = new
                      VarianceProcessParameters(Kappa, Theta, Sigma, Nu, Rho);
171                 HestonModelParameters parameters = new HestonModelParameters
                      (InitialStockPrice, RiskFreeRate, VProcessParameters);
172                 EuropeanOption Option = new EuropeanOption(0, OptionExercise
                      [i], HestonModel.PayoffType.Call);
173                 MonteCarloSettings settings = new MonteCarloSettings(Paths,
                      TimeSteps);
174                 double Price = Heston.HestonLookbackOptionPriceMC(parameters,
                      Option, settings);
175                 Console.WriteLine(Price);
176             }
177             Console.ReadKey();
178         }
179
180     }
181 }
182
183
184
```

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading;
 6  using System.Threading.Tasks;
 7
 8  namespace FinalProject
 9  {
10      public class NoAssetsAddedException : Exception
11      {
12          /// <summary>
13          /// Exception that will be thrown if one attempts to calculate the
              price and the portfolio has no assets.
14          /// </summary>
15          public NoAssetsAddedException(string message)
16              : base(message)
17          {
18          }
19      }
20
21      /// <summary>
22      /// Class for pricing Everest Options.
23      /// An Everest option is a type of exotic option belonging to a class known
            as mountain range
24      /// options. The value of an Everest option is based on a basket of
          underlying securities.
25      /// </summary>
26      public class EverestOptions
27      {
28          private List<HestonMonteCarlo> Assets = new List<HestonMonteCarlo>();
29          private readonly int TimeSteps;
30          private readonly double RiskFreeRate;
31
32          public EverestOptions(double RiskFreeRate, int TimeSteps =365)
33          {
34              this.RiskFreeRate = RiskFreeRate;
35              this.TimeSteps = TimeSteps;
36          }
37
38          /// <summary>
39          /// Adds an asset to the portfolio that will be used in the option
              price calculation.
40          /// </summary>
41          /// <param name="StrikePrice"></param>
42          /// <param name="InitialStockPrice"></param>
43          /// <param name="Kappa"></param>
44          /// <param name="Theta"></param>
45          /// <param name="Sigma"></param>
```

```
46          /// <param name="Rho"></param>
47          /// <param name="Nu"></param>
48          public void AddAsset(
49              double StrikePrice,
50              double InitialStockPrice,
51              double Kappa,
52              double Theta,
53              double Sigma,
54              double Rho,
55              double Nu)
56          {
57
58              HestonMonteCarlo NewAsset = new HestonMonteCarlo(RiskFreeRate,
                  StrikePrice, InitialStockPrice, Kappa,
59                    Theta, Sigma, Rho, Nu, TimeSteps);
60              Assets.Add(NewAsset);
61          }
62
63          private double AuxFunction(double [] Path)
64          {
65              return Path[Path.Length - 1] / Path[0];
66          }
67          /// <summary>
68          /// Calculate the price of an Everest Option. You must have added
              assets for it to work.
69          /// </summary>
70          /// <param name="OptionExercise"></param>
71          /// <param name="NPaths"></param>
72          /// <returns>The Price.</returns>
73          /// <remarks>
74          /// We use multi-threading just is we did in HestonMonteCarlo.
75          /// </remarks>
76          /// <exception cref="NoAssetsAddedException">
77          /// Thrown we one attempts to calculate the price when the portfolio
              has no assets.
78          /// </exception>
79          public double CalculatePrice(double OptionExercise, int NPaths = 10000)
80          {
81              if (Assets is null)
82                  throw new NoAssetsAddedException("--- You have added no assets
                      to the models. ---");
83
84              int NumSamples = (int)Math.Ceiling(OptionExercise * TimeSteps);
85              double Tau = OptionExercise / NumSamples;
86
87              double Aux = 0;
88              double[] Aux2 = new double[Assets.Count];
89              Parallel.For(0, NPaths, i => {
90
```

```
91                  for(int j=0; j<Assets.Count; j++)
92                  {
93                      double[] Path = Assets[j].GeneratePath(NumSamples, Tau);
94                      Aux2[j] = AuxFunction(Path);
95
96                  }
97
98                  double Payoff = Aux2.Min();
99
100                 Interlocked.Exchange(ref Aux, Payoff + Aux);
101
102             });
103             return Math.Exp(-RiskFreeRate * OptionExercise) * Aux / NPaths;
104
105         }
106
107     }
108 }
109
```

```
using FinalProject;
using System;
using System.Globalization;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace OptionGUI
{
    /// <summary>
    /// This class contains all functionality related with the GUI. Here most of the classes of FinalProjet are called.
    /// </summary>
    public partial class OptionCalculator : Form
    {
        public OptionCalculator()
        {
            InitializeComponent();

        }

        double RiskFreeRate;
        double StrikePrice;
        double OptionExcercise;
        double Sigma;
        double InitialStockPrice;
        double Kappa;
        double Theta;
        double Rho;
        double Nu;
        int TimeSteps;
        int Paths;

        /// <summary>
        /// If the button is clicked the price is calculated based on the information provided in the text boxes and the combo box.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void button1_Click(object sender, EventArgs e)
        {


```

```
48              RiskFreeRate = double.Parse(textBox2.Text);
49              StrikePrice = double.Parse(textBox3.Text);
50              OptionExcercise = double.Parse(textBox4.Text);
51              Sigma = double.Parse(textBox9.Text);
52              InitialStockPrice = double.Parse(textBox6.Text);
53
54
55
56
57              double Price = 0;
58
59              if (comboBox2.Text == "Heston Formula") {
60
61                  Kappa = double.Parse(textBox11.Text);
62                  Theta = double.Parse(textBox10.Text);
63                  Rho = double.Parse(textBox8.Text);
64                  Nu = double.Parse(textBox7.Text);
65
66
67                  HestonFormula Formula = new HestonFormula(RiskFreeRate,
                      InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu);
68
69
70
71                  if (comboBox1.Text == "Call")
72                  {
73                      Price = Formula.CalculateCallPrice(StrikePrice,
                        OptionExcercise);
74                  }
75                  else if (comboBox1.Text == "Put")
76                  {
77                      Price = Formula.CalculatePutPrice(StrikePrice,
                        OptionExcercise);
78                  }
79
80
81
82              }else if (comboBox2.Text == "Heston Monte Carlo")
83              {
84                  Kappa = double.Parse(textBox11.Text);
85                  Theta = double.Parse(textBox10.Text);
86                  Rho = double.Parse(textBox8.Text);
87                  Nu = double.Parse(textBox7.Text);
88                  TimeSteps = int.Parse(textBox16.Text);
89                  Paths = int.Parse(textBox15.Text);
90
91
92
93                  HestonMonteCarlo MonteCarlo = new HestonMonteCarlo
```

```
                    (RiskFreeRate, StrikePrice, InitialStockPrice,
 94                   Kappa, Theta, Sigma, Rho, Nu, TimeSteps);
 95
 96                 if (comboBox1.Text == "Call")
 97                 {
 98                     Price = MonteCarlo.CalculatePrice(OptionExcercise, Paths);
 99                 }
100                 else if (comboBox1.Text == "Put")
101                 {
102                     Price = MonteCarlo.CalculatePutPrice(OptionExcercise,
                        Paths);
103                 }
104
105
106
107
108             }
109             else if (comboBox2.Text == "Lookback")
110             {
111                 Kappa = double.Parse(textBox11.Text);
112                 Theta = double.Parse(textBox10.Text);
113                 Rho = double.Parse(textBox8.Text);
114                 Nu = double.Parse(textBox7.Text);
115                 TimeSteps = int.Parse(textBox16.Text);
116                 Paths = int.Parse(textBox15.Text);
117
118                 LookbackOptions MonteCarlo = new LookbackOptions(RiskFreeRate,
                    InitialStockPrice,
119                 Kappa, Theta, Sigma, Rho, Nu, TimeSteps);
120
121
122                 Price = MonteCarlo.CalculatePrice(OptionExcercise, Paths);
123
124
125             }
126             else if (comboBox2.Text == "Cliquet")
127             {
128
129                 double Aux1 = double.Parse(textBox12.Text);
130                 double Aux2 = double.Parse(textBox17.Text);
131                 double Aux3 = double.Parse(textBox18.Text);
132
133                 double[] FixingDates = { Aux1, Aux2, Aux3};
134
135                 double PayoutRate = double.Parse(textBox13.Text);
136
137                 int Lambda = -1;
138
139                 if (comboBox1.Text == "Call")
```

```
140                 {
141                     Lambda = 1;
142                 }
143 
144 
145                 CliquetOptions Option = new CliquetOptions(RiskFreeRate,
                      FixingDates, Lambda, Sigma, StrikePrice, InitialStockPrice,
                      PayoutRate);
146                 Price = Option.CalculatePrice();
147 
148             }
149 
150 
151 
152 
153             textBox1.Text = Price.ToString();
154 
155 
156         }
157 
158         private void Form1_Load(object sender, EventArgs e)
159         {
160 
161         }
162 
163         private void label2_Click(object sender, EventArgs e)
164         {
165 
166         }
167 
168         private void label8_Click(object sender, EventArgs e)
169         {
170 
171         }
172 
173         private void label9_Click(object sender, EventArgs e)
174         {
175 
176         }
177 
178         private void textBox3_TextChanged(object sender, EventArgs e)
179         {
180 
181         }
182 
183         private void label11_Click(object sender, EventArgs e)
184         {
185 
186         }
```

```
187
188         private void label12_Click(object sender, EventArgs e)
189         {
190
191         }
192
193         private void label16_Click(object sender, EventArgs e)
194         {
195
196         }
197
198         private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
199         {
200
201         }
202
203         private void textBox1_TextChanged(object sender, EventArgs e)
204         {
205
206         }
207
208         private void comboBox2_SelectedIndexChanged(object sender, EventArgs e)
209         {
210             if(comboBox2.Text == "Plot Convergence (Monte Carlo and Formula)")
211             {
212                 button1.Enabled = false;
213                 comboBox1.Enabled = true;
214                 textBox11.Enabled = true;
215                 textBox10.Enabled = true;
216                 textBox8.Enabled = true;
217                 textBox7.Enabled = true;
218                 textBox4.Enabled = true;
219                 textBox16.Enabled = true;
220                 textBox15.Enabled = true;
221                 textBox12.Enabled = false;
222                 textBox17.Enabled = false;
223                 textBox18.Enabled = false;
224                 textBox13.Enabled = false;
225                 PlotConvergence.Visible = true;
226             }
227             else
228             {
229                 PlotConvergence.Visible = false;
230                 button1.Enabled = true;
231             }
232
233             if (comboBox2.Text == "Heston Formula")
234             {
235
```

```
236                 comboBox1.Enabled = true;
237                 textBox11.Enabled = true;
238                 textBox10.Enabled = true;
239                 textBox8.Enabled = true;
240                 textBox7.Enabled = true;
241                 textBox4.Enabled = true;
242                 textBox16.Enabled = false;
243                 textBox15.Enabled = false;
244                 textBox12.Enabled = false;
245                 textBox17.Enabled = false;
246                 textBox18.Enabled = false;
247                 textBox13.Enabled = false;
248             }else if(comboBox2.Text == "Heston Monte Carlo" )
249             {
250                 comboBox1.Enabled = true;
251                 textBox11.Enabled = true;
252                 textBox10.Enabled = true;
253                 textBox8.Enabled = true;
254                 textBox7.Enabled = true;
255                 textBox4.Enabled = true;
256                 textBox16.Enabled = true;
257                 textBox15.Enabled = true;
258                 textBox12.Enabled = false;
259                 textBox17.Enabled = false;
260                 textBox18.Enabled = false;
261                 textBox13.Enabled = false;
262 
263             }else if (comboBox2.Text == "Lookback")
264             {
265                 textBox11.Enabled = true;
266                 textBox10.Enabled = true;
267                 textBox8.Enabled = true;
268                 textBox7.Enabled = true;
269                 textBox4.Enabled = true;
270                 textBox16.Enabled = true;
271                 textBox15.Enabled = true;
272                 textBox12.Enabled = false;
273                 textBox17.Enabled = false;
274                 textBox18.Enabled = false;
275                 textBox13.Enabled = false;
276                 comboBox1.Enabled = false;
277             }
278             else if (comboBox2.Text == "Cliquet")
279             {
280                 comboBox1.Enabled = true;
281                 textBox11.Enabled = false;
282                 textBox10.Enabled = false;
283                 textBox8.Enabled = false;
284                 textBox7.Enabled = false;
```

```
285                 textBox16.Enabled = false;
286                 textBox15.Enabled = false;
287                 textBox4.Enabled = false;
288                 textBox12.Enabled = true;
289                 textBox17.Enabled = true;
290                 textBox18.Enabled = true;
291                 textBox13.Enabled = true;
292             }
293 
294         }
295 
296         private void textBox2_TextChanged(object sender, EventArgs e)
297         {
298 
299         }
300 
301         private void textBox4_TextChanged(object sender, EventArgs e)
302         {
303 
304         }
305 
306         private void textBox5_TextChanged(object sender, EventArgs e)
307         {
308 
309         }
310 
311         private void textBox6_TextChanged(object sender, EventArgs e)
312         {
313 
314         }
315 
316         private void textBox11_TextChanged(object sender, EventArgs e)
317         {
318 
319         }
320 
321         private void textBox10_TextChanged(object sender, EventArgs e)
322         {
323 
324         }
325 
326         private void textBox9_TextChanged(object sender, EventArgs e)
327         {
328 
329         }
330 
331         private void textBox8_TextChanged(object sender, EventArgs e)
332         {
333 
```

```
334         }
335
336         private void textBox7_TextChanged(object sender, EventArgs e)
337         {
338
339         }
340
341         private void textBox16_TextChanged(object sender, EventArgs e)
342         {
343
344         }
345
346         private void textBox15_TextChanged(object sender, EventArgs e)
347         {
348
349         }
350
351         private void textBox13_TextChanged(object sender, EventArgs e)
352         {
353
354         }
355
356         private void textBox12_TextChanged(object sender, EventArgs e)
357         {
358
359         }
360
361         private void textBox17_TextChanged(object sender, EventArgs e)
362         {
363
364         }
365
366         private void textBox18_TextChanged(object sender, EventArgs e)
367         {
368
369         }
370
371         private void PlotConvergence_Click(object sender, EventArgs e)
372         {
373             Form2 PlotConvergenceForm = new Form2();
374             PlotConvergenceForm.Show();
375         }
376
377         private void progressBar1_Click(object sender, EventArgs e)
378         {
379
380         }
381
382         private void groupBox1_Enter(object sender, EventArgs e)
```

```
383         {
384
385         }
386     }
387 }
388
```

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.ComponentModel;
 4  using System.Data;
 5  using System.Drawing;
 6  using System.Linq;
 7  using System.Text;
 8  using System.Threading.Tasks;
 9  using System.Windows.Forms;
10  using System.Windows.Forms.DataVisualization.Charting;
11  using FinalProject;
12  using FinalProjectTests;
13
14  namespace OptionGUI
15  {
16      public partial class Form2 : Form
17      {
18
19
20          public Form2()
21          {
22              InitializeComponent();
23
24          }
25
26          private void Form2_Load(object sender, EventArgs e)
27          {
28
29          }
30
31          private void chart1_Click(object sender, EventArgs e)
32          {
33
34          }
35          /// <summary>
36          /// This method produces the charts that we see in the GUI. It first
              calls the ConvergenceHestonMonteCarlo
37          /// class that calculates the error and then produces two series with
              all its features.
38          /// </summary>
39          /// <param name="sender"></param>
40          /// <param name="e"></param>
41          private void button2_Click(object sender, EventArgs e)
42          {
43
44
45              label6.Show();
46
47              ConvergenceHestonMonteCarlo Function = new
```

```
            ConvergenceHestonMonteCarlo();

            int[] X = { 1, 10, 100, 1000, 10000, 30000};
            double[] Y1 = Function.DataPointsPath(X);
            double[] Y2 = Function.DataPointsTimeSteps(X);
            chart1.Series.Clear();
            chart2.Series.Clear();

            var Series1 = new
              System.Windows.Forms.DataVisualization.Charting.Series
            {
                Name = "Series1",
                Color = System.Drawing.Color.Green,
                IsVisibleInLegend = false,
                IsXValueIndexed = true,
                ChartType = SeriesChartType.Spline
            };

            var Series2 = new
              System.Windows.Forms.DataVisualization.Charting.Series
            {
                Name = "Series2",
                Color = System.Drawing.Color.Blue,
                IsVisibleInLegend = false,
                IsXValueIndexed = true,
                ChartType = SeriesChartType.Spline
            };

            chart1.Series.Add(Series1);
            chart2.Series.Add(Series2);

            for (int i=1;i<X.Length; i++)
            {
                chart1.Series["Series1"].Points.AddXY(X[i], Y1[i]);
                chart2.Series["Series2"].Points.AddXY(X[i], Y2[i]);
            }
            label6.Hide();
            chart1.ChartAreas[0].AxisX.IsMarginVisible = false;
            chart2.ChartAreas[0].AxisX.IsMarginVisible = false;
        }

        private void button1_Click(object sender, EventArgs e)
        {

            this.Hide();

        }

        private void chart2_Click(object sender, EventArgs e)
```

```
94          {
95
96          }
97
98          private void eventLog2_EntryWritten(object sender,
              System.Diagnostics.EntryWrittenEventArgs e)
99          {
100
101         }
102
103         private void label2_Click(object sender, EventArgs e)
104         {
105
106         }
107
108         private void label1_Click(object sender, EventArgs e)
109         {
110
111         }
112
113         private void label3_Click(object sender, EventArgs e)
114         {
115
116         }
117
118         private void label6_Click(object sender, EventArgs e)
119         {
120
121         }
122     }
123 }
124
```

```
1  using System;
2  using System.Collections.Generic;
3  using HestonModel.Interfaces;
4  using FinalProject;
5  using System.Linq;
6
7  namespace HestonModel
8  {
9
10     /// <summary>
11     /// This class will be used for grading.
12     /// Don't remove any of the methods and don't modify their signatures.
         Don't change the namespace.
13     /// Your code should be implemented in other classes (or even projects if
         you wish), and the relevant functionality should only be called here and
         outputs returned.
14     /// You don't need to implement the interfaces that have been provided if
         you don't want to.
15     /// </summary>
16     public static class Heston
17     {
18         /// <summary>
19         /// Method for calibrating the heston model.
20         /// </summary>
21         /// <param name="guessModelParameters">Object implementing
             IHestonModelParameters interface containing the risk-free rate,
             initial stock price
22         /// and initial guess parameters to be used in the calibration.</param>
23         /// <param name="referenceData">A collection of objects implementing
             IOptionMarketData<IEuropeanOption> interface. These should contain
             the reference data used for calibration.</param>
24         /// <param name="calibrationSettings">An object implementing
             ICalibrationSettings interface.</param>
25         /// <returns>Object implementing IHestonCalibrationResult interface
             which contains calibrated model parameters and additional diagnostic
             information</returns>
26         public static HestonCalibrationResult CalibrateHestonParameters
             (IHestonModelParameters guessModelParameters,
27             IEnumerable<IOptionMarketData<IEuropeanOption>> referenceData,
                 ICalibrationSettings calibrationSettings)
28         {
29             double RiskFreeRate = guessModelParameters.RiskFreeRate;
30             double InitialStockPrice = guessModelParameters.InitialStockPrice;
31             double Kappa = guessModelParameters.VarianceParameters.Kappa;
32             double Theta = guessModelParameters.VarianceParameters.Theta;
33             double Sigma = guessModelParameters.VarianceParameters.Sigma;
34             double Rho = guessModelParameters.VarianceParameters.Rho;
35             double Nu = guessModelParameters.VarianceParameters.V0;
36             double Accuracy = calibrationSettings.Accuracy;
```

```
37              int MaxIterations = calibrationSettings.MaximumNumberOfIterations;
38              int j = 0;
39
40              double[] StrikePrices = new double[referenceData.Count()];
41              double[] OptionExerciseTimes = new double[referenceData.Count()];
42              double[] Prices = new double[referenceData.Count()];
43
44              foreach (IOptionMarketData<IEuropeanOption> Data in referenceData)
45              {
46                  StrikePrices[j] = Data.Option.StrikePrice;
47                  OptionExerciseTimes[j] = Data.Option.Maturity;
48                  Prices[j] = Data.Price;
49                  j++;
50              }
51
52              HestonCalibration Calibrator = new HestonCalibration(RiskFreeRate,
                  InitialStockPrice, Accuracy, MaxIterations);
53              Calibrator.SetGuessParameters(Kappa, Theta, Sigma, Rho, Nu);
54              for (int i = 0; i < Prices.Length; ++i)
55              {
56                  Calibrator.AddObservedOption(OptionExerciseTimes[i],
                      StrikePrices[i], Prices[i]);
57              }
58              Calibrator.Calibrate();
59              double Error = 0;
60              FinalProject.CalibrationOutcome Outcome =
                  FinalProject.CalibrationOutcome.NotStarted;
61              Calibrator.GetCalibrationStatus(ref Outcome, ref Error);
62              HestonFormula CalibratedModel = Calibrator.GetCalibratedModel();
63              double[] Param = CalibratedModel.ConvertCalibrationParamsToArray();
64              VarianceProcessParameters VParam = new VarianceProcessParameters
                  (Param[0], Param[1], Param[2], Param[4], Param[3]);
65              HestonModelParameters HestonMP = new HestonModelParameters
                  (InitialStockPrice, RiskFreeRate, VParam);
66              HestonCalibrationResult Result;
67
68              if (Outcome == FinalProject.CalibrationOutcome.FailedOtherReason)
69              {
70                  Result = new HestonCalibrationResult(Error, HestonMP,
                      CalibrationOutcome.FailedOtherReason);
71              }
72              else if (Outcome ==
                  FinalProject.CalibrationOutcome.FailedMaxItReached)
73              {
74                  Result = new HestonCalibrationResult(Error, HestonMP,
                      CalibrationOutcome.FailedMaxItReached);
75              }
76              else if (Outcome == FinalProject.CalibrationOutcome.NotStarted)
77              {
```

```
78                  Result = new HestonCalibrationResult(Error, HestonMP,
                      CalibrationOutcome.NotStarted);
79              }
80              else
81              {
82                  Result = new HestonCalibrationResult(Error, HestonMP,
                      CalibrationOutcome.FinishedOK);
83              }
84
85              return Result;
86
87          }
88
89          /// <summary>
90          /// Price a European option in the Heston model using the Heston
              formula. This should be accurate to 5 decimal places
91          /// </summary>
92          /// <param name="parameters">Object implementing IHestonModelParameters
               interface, containing model parameters.</param>
93          /// <param name="europeanOption">Object implementing IEuropeanOption
              interface, containing the option parameters.</param>
94          /// <returns>Option price</returns>
95          public static double HestonEuropeanOptionPrice(IHestonModelParameters
              parameters, IEuropeanOption europeanOption)
96          {
97              double RiskFreeRate = parameters.RiskFreeRate;
98              double StrikePrice = europeanOption.StrikePrice;
99              double OptionExercise = europeanOption.Maturity;
100             double Kappa = parameters.VarianceParameters.Kappa;
101             double Theta = parameters.VarianceParameters.Theta;
102             double Sigma = parameters.VarianceParameters.Sigma;
103             double Rho = parameters.VarianceParameters.Rho;
104             double InitialStockPrice = parameters.InitialStockPrice;
105             double Nu = parameters.VarianceParameters.V0;
106
107             HestonFormula Formula = new HestonFormula(RiskFreeRate,
                  InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu);
108
109             double Price = 0;
110
111             if (europeanOption.Type == PayoffType.Call)
112             {
113                 Price = Formula.CalculateCallPrice(StrikePrice,
                      OptionExercise);
114             }
115             else if (europeanOption.Type == PayoffType.Put)
116             {
117                 Price = Formula.CalculatePutPrice(StrikePrice, OptionExercise);
118             }
```

```
119
120             return Price;
121
122         }
123
124         /// <summary>
125         /// Price a European option in the Heston model using the Monte-Carlo
              method. Accuracy will depend on number of time steps and samples
126         /// </summary>
127         /// <param name="parameters">Object implementing IHestonModelParameters
              interface, containing model parameters.</param>
128         /// <param name="europeanOption">Object implementing IEuropeanOption
              interface, containing the option parameters.</param>
129         /// <param name="monteCarloSimulationSettings">An object implementing
              IMonteCarloSettings object and containing simulation settings.</
              param>
130         /// <returns>Option price</returns>
131         public static double HestonEuropeanOptionPriceMC(IHestonModelParameters
              parameters, IEuropeanOption europeanOption, IMonteCarloSettings
              monteCarloSimulationSettings)
132         {
133
134             double RiskFreeRate = parameters.RiskFreeRate;
135             double StrikePrice = europeanOption.StrikePrice;
136             double OptionExercise = europeanOption.Maturity;
137             double Kappa = parameters.VarianceParameters.Kappa;
138             double Theta = parameters.VarianceParameters.Theta;
139             double Sigma = parameters.VarianceParameters.Sigma;
140             double Rho = parameters.VarianceParameters.Rho;
141             double InitialStockPrice = parameters.InitialStockPrice;
142             double Nu = parameters.VarianceParameters.V0;
143             int TimeSteps = monteCarloSimulationSettings.NumberOfTimeSteps;
144             int Paths = monteCarloSimulationSettings.NumberOfTrials;
145
146             HestonMonteCarlo MonteCarlo = new HestonMonteCarlo(RiskFreeRate,
                  StrikePrice, InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu,
                  TimeSteps);
147
148             double Price = 0;
149
150             if (europeanOption.Type == PayoffType.Call)
151             {
152                 Price = MonteCarlo.CalculatePrice(OptionExercise, Paths);
153             }
154             else if (europeanOption.Type == PayoffType.Put)
155             {
156                 Price = MonteCarlo.CalculatePutPrice(OptionExercise, Paths);
157             }
158
```

```
159             return Price;
160         }
161
162         /// <summary>
163         /// Price a Asian option in the Heston model using the
164         /// Monte-Carlo method. Accuracy will depend on number of time steps
              and samples</summary>
165         /// <param name="parameters">Object implementing IHestonModelParameters
               interface, containing model parameters.</param>
166         /// <param name="asianOption">Object implementing IAsian interface,
              containing the option parameters.</param>
167         /// <param name="monteCarloSimulationSettings">An object implementing
              IMonteCarloSettings object and containing simulation settings.</
              param>
168         /// <returns>Option price</returns>
169         public static double HestonAsianOptionPriceMC(IHestonModelParameters
              parameters, IAsianOption asianOption, IMonteCarloSettings
              monteCarloSimulationSettings)
170         {
171             IEnumerable<double> Dates = asianOption.MonitoringTimes;
172             double RiskFreeRate = parameters.RiskFreeRate;
173             double StrikePrice = asianOption.StrikePrice;
174             double OptionExercise = asianOption.Maturity;
175             double Kappa = parameters.VarianceParameters.Kappa;
176             double Theta = parameters.VarianceParameters.Theta;
177             double Sigma = parameters.VarianceParameters.Sigma;
178             double Rho = parameters.VarianceParameters.Rho;
179             double InitialStockPrice = parameters.InitialStockPrice;
180             double Nu = parameters.VarianceParameters.V0;
181             int TimeSteps = monteCarloSimulationSettings.NumberOfTimeSteps;
182             int Paths = monteCarloSimulationSettings.NumberOfTrials;
183
184             AsianOptions AsianOption = new AsianOptions(Dates, RiskFreeRate,
                  StrikePrice, InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu,
                  TimeSteps);
185
186             double Price = 0;
187
188             if (asianOption.Type == PayoffType.Call)
189             {
190                 Price = AsianOption.CalculatePrice(OptionExercise, Paths);
191             }
192             else if (asianOption.Type == PayoffType.Put)
193             {
194                 Price = AsianOption.CalculatePutPrice(OptionExercise, Paths);
195             }
196
197             return Price;
198         }
```

```
199
200         /// <summary>
201         /// Price a lookback option in the Heston model using the
202         /// a Monte-Carlo method. Accuracy will depend on number of time steps
              and samples </summary>
203         /// <param name="parameters">Object implementing IHestonModelParameters
              interface, containing model parameters.</param>
204         /// <param name="maturity">An object implementing IOption interface and
              containing option's maturity</param>
205         /// <param name="monteCarloSimulationSettings">An object implementing
              IMonteCarloSettings object and containing simulation settings.</
              param>
206         /// <returns>Option price</returns>
207         public static double HestonLookbackOptionPriceMC(IHestonModelParameters
              parameters, IOption maturity, IMonteCarloSettings
              monteCarloSimulationSettings)
208         {
209             double RiskFreeRate = parameters.RiskFreeRate;
210
211             double OptionExercise = maturity.Maturity;
212             double Kappa = parameters.VarianceParameters.Kappa;
213             double Theta = parameters.VarianceParameters.Theta;
214             double Sigma = parameters.VarianceParameters.Sigma;
215             double Rho = parameters.VarianceParameters.Rho;
216             double InitialStockPrice = parameters.InitialStockPrice;
217             double Nu = parameters.VarianceParameters.V0;
218             int TimeSteps = monteCarloSimulationSettings.NumberOfTimeSteps;
219             int Paths = monteCarloSimulationSettings.NumberOfTrials;
220
221             LookbackOptions Lookback = new LookbackOptions(RiskFreeRate,
                  InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu, TimeSteps);
222
223             double Price = Lookback.CalculatePrice(OptionExercise, Paths);
224
225             return Price;
226         }
227     }
228 }
229
```

```
1  using System;
2  using System.Collections.Generic;
3
4
5  namespace FinalProject
6  {
7      /// <summary>
8      /// Exception that will be thrown if the calibration fails.
9      /// </summary>
10     /// <exception cref="CalibrationFailedException"></exception>
11     public class CalibrationFailedException : Exception
12     {
13         public CalibrationFailedException()
14         {
15         }
16         public CalibrationFailedException(string message)
17             : base(message)
18         {
19         }
20     }
21
22     /// <summary>
23     /// Enumerator used to identify the type of outcome of the calibration.
24     /// </summary>
25     public enum CalibrationOutcome
26     {
27         NotStarted,
28         FinishedOK,
29         FailedMaxItReached,
30         FailedOtherReason
31     };
32
33     /// <summary>
34     /// Struct that contains the information needed to create instances of
         HestonFormula and
35     /// minimize the squared error.
36     /// </summary>
37     public struct CallOptionMarketData
38     {
39
40         public double OptionExercise;
41         public double StrikePrice;
42         public double MarketCallPrice;
43     }
44
45     /// <summary>
46     /// Class used to calibrate the Kappa, Theta, Sigma, Rho and Nu parameters
         from the HestonFormula class.
47     /// The calibration seeks to reduce the squared error between the model and
```

```
        the observed prices.
48      /// </summary>
49      public class HestonCalibration
50      {
51          private double[] CalibratedParams;
52          private readonly double InitialStockPrice;
53          private readonly double Accuracy;
54          private readonly double RiskFreeRate;
55          private readonly int MaxIterations;
56          private LinkedList<CallOptionMarketData> MarketOptionsList;
57          private CalibrationOutcome Outcome;
58
59
60          /// <summary>
61          /// Constructor for the class.
62          /// </summary>
63          /// <remarks>
64          /// An initial guess is provided but it can be changed.
65          /// <see cref="SetGuessParameters(double,double,double,double,double)"/
              >
66          /// </remarks>
67          public HestonCalibration(double RiskFreeRate, double InitialStockPrice,
                double Accuracy, int MaxIterations)
68          {
69              this.RiskFreeRate = RiskFreeRate;
70              this.InitialStockPrice = InitialStockPrice;
71              this.Accuracy = Accuracy;
72              this.MaxIterations = MaxIterations;
73              MarketOptionsList = new LinkedList<CallOptionMarketData>();
74              CalibratedParams = new double[] { 0.5, 0.01, 0.2, 0.1, 0.4 };
75          }
76
77          /// <summary>
78          /// Gives the option to change the initial guess.
79          /// </summary>
80          /// <param name="Kappa">Double precision parameter.</param>
81          /// <param name="Theta">Double precision parameter.</param>
82          /// <param name="Sigma">Volatility.</param>
83          /// <param name="Rho">Double precision parameter.</param>
84          /// <param name="Nu">Initial variance.</param>
85          /// <remarks>
86          /// By creating an instance of the class HestonFormula we make sure all
                parameters are fit for purpose.
87          /// </remarks>
88          public void SetGuessParameters( double Kappa, double Theta, double
              Sigma, double Rho, double Nu)
89          {
90              HestonFormula m = new HestonFormula(RiskFreeRate,
                  InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu);
```

```
91              CalibratedParams = m.ConvertCalibrationParamsToArray();
92          }
93
94          /// <summary>
95          /// Adds an observation that will be used in the calibration.
96          /// </summary>
97          /// <param name="OptionExercise">Double precision parameter.</param>
98          /// <param name="StrikePrice">Double precision parameter.</param>
99          /// <param name="MarketCallPrice">Double precision parameter.</param>
100         public void AddObservedOption(double OptionExercise, double
              StrikePrice, double MarketCallPrice)
101         {
102             CallOptionMarketData ObservedOption;
103             ObservedOption.StrikePrice = StrikePrice;
104             ObservedOption.OptionExercise = OptionExercise;
105             ObservedOption.MarketCallPrice = MarketCallPrice;
106             MarketOptionsList.AddLast(ObservedOption);
107         }
108
109         /// <summary>
110         /// Calculates the difference between the observed and the model
              prices.
111         /// </summary>
112         /// <returns>
113         /// A double precision number that represents the squared error.
114         /// </returns>
115         /// <param name="m">Instance of HestonFormula used for the model
              prices.</param>
116         public double CalcMeanSquareErrorBetweenModelAndMarket(HestonFormula m)
117         {
118             double MeanSqErr = 0;
119             foreach (CallOptionMarketData Option in MarketOptionsList)
120             {
121                 double StrikePrice = Option.StrikePrice;
122                 double OptionExercise = Option.OptionExercise;
123                 double ModelPrice = m.CalculateCallPrice(StrikePrice,
                      OptionExercise);
124
125                 double Difference = ModelPrice - Option.MarketCallPrice;
126                 MeanSqErr += Difference * Difference;
127             }
128             return MeanSqErr;
129         }
130
131         /// <summary>
132         /// Defines the function used by the Alglib minimization algorithm.
133         /// </summary>
134         /// <param name="ParamsArray">Array of doubles ordered as follows:
              Kappa, Theta, Sigma, Rho and Nu.</param>
```

```
135         /// <param name="func">Function used passed by reference.</param>
136         /// <param name="obj">Used by Alglib.</param>
137         public void CalibrationObjectiveFunction(double[] ParamsArray, ref
              double func, object obj)
138         {
139             HestonFormula m = new HestonFormula(RiskFreeRate,
                  InitialStockPrice, ParamsArray);
140             func = CalcMeanSquareErrorBetweenModelAndMarket(m);
141         }
142
143         /// <summary>
144         /// Main function of the class. Uses Alglib to reduce the error between
               the model and the observed
145         /// prices.
146         /// </summary>
147         /// <remarks>
148         /// <para>Though the function does not return a value, <paramref
              name="CalibratedParams"/> contains
149         /// the result if there was not an exception.</para>
150         /// <para>The value of <paramref name="Stpmax"/> is very important for
              the convergence of the algorithm.</para>
151         /// <para>Epsg, Epsf, and Epsx are taken equal.</para>
152         /// <para>The method prints out the result in the console.</para>
153         /// </remarks>
154         /// <exception cref="CalibrationFailedException">
155         /// Thrown if the outcome type is not either - 1,2,4 or 5, i.e it did
              not converge or reach the maximum
156         /// number of iterations.
157         /// </exception>
158         public void Calibrate()
159         {
160             Outcome = CalibrationOutcome.NotStarted;
161             double[] InitialParams = new double[5];
162             CalibratedParams.CopyTo(InitialParams, 0);
163             double Epsg = Accuracy;
164             double Epsf = Accuracy;
165             double Epsx = Accuracy;
166             double Diffstep = 1.0e-6;
167             int Maxits = MaxIterations;
168             double Stpmax = 0.05;
169
170             alglib.minlbfgscreatef(5, InitialParams, Diffstep, out
                  alglib.minlbfgsstate State);
171             alglib.minlbfgssetcond(State, Epsg, Epsf, Epsx, Maxits);
172             alglib.minlbfgssetstpmax(State, Stpmax);
173
174             alglib.minlbfgsoptimize(State, CalibrationObjectiveFunction, null,
                  null);
175
```

```
176             double[] ResultParams = new double[5];
177             alglib.minlbfgsresults(State, out ResultParams, out
                  alglib.minlbfgsreport Rep);
178
179             System.Console.WriteLine("Termination type: {0}",
                  Rep.terminationtype);
180             System.Console.WriteLine("Num iterations {0}",
                  Rep.iterationscount);
181             System.Console.WriteLine("{0}", alglib.ap.format(ResultParams, 5));
182
183             if (Rep.terminationtype == 1
184                 || Rep.terminationtype == 2
185                 || Rep.terminationtype == 4)
186             {
187                 Outcome = CalibrationOutcome.FinishedOK;
188
189                 CalibratedParams = ResultParams;
190             }
191             else if (Rep.terminationtype == 5)
192             {
193                 Outcome = CalibrationOutcome.FailedMaxItReached;
194
195                 CalibratedParams = ResultParams;
196
197             }
198             else
199             {
200                 Outcome = CalibrationOutcome.FailedOtherReason;
201                 throw new CalibrationFailedException("-- Model calibration
                      failed badly. --");
202             }
203         }
204
205         /// <summary>
206         /// Used to know the outcome and error of the calibration.
207         /// </summary>
208         /// <param name="CalibOutcome">Must be a CalibrationOutcome object. It
              is modified by reference.</param>
209         /// <param name="PricingError">Represents the error. It is modified by
              reference.</param>
210         public void GetCalibrationStatus(ref CalibrationOutcome CalibOutcome,
              ref double PricingError)
211         {
212             CalibOutcome = Outcome;
213             HestonFormula m = new HestonFormula(RiskFreeRate,
                  InitialStockPrice, CalibratedParams);
214             PricingError = CalcMeanSquareErrorBetweenModelAndMarket(m);
215         }
216
```

```
217         /// <summary>
218         /// A way to get all the parameters of the model.
219         /// </summary>
220         /// <returns>
221         /// A HestonFormula object with all the information of the model.
222         /// </returns>
223         public HestonFormula GetCalibratedModel()
224         {
225             HestonFormula m = new HestonFormula(RiskFreeRate,
                  InitialStockPrice, CalibratedParams);
226             return m;
227         }
228     }
229
230 }
231
232
233
```

```
1  using System;
2  using System.Numerics;
3  using MathNet.Numerics.Integration;
4
5  namespace FinalProject
6  {
7      /// <summary>
8      /// Class for pricing European put and call options using the Heston
         formula.
9      /// <see href="final-project-almedina12/HestonModel.pdf">Reference.</see>
10     /// </summary>
11     /// <remarks>
12     /// The names of the global variables  and functions are meant to resemble
         the ones in
13     /// the documentation for easier understanding.
14     /// </remarks>
15     public class HestonFormula
16     {
17         private double RiskFreeRate { get; }
18         private double StrikePrice { get; set; }
19         private double InitialStockPrice { get;}
20         private double Kappa { get; }
21         private double Theta { get; }
22         private double Sigma { get; }
23         private double Rho { get; }
24         private double Nu { get; set; }
25         private double[] u { get; }
26         private double[] b { get; }
27         private double a { get; }
28         private Complex i = Complex.ImaginaryOne;
29         private int NumModelParams { get; }
30
31
32         /// <summary>
33         /// Constructor for the class. Global variables <paramref name="u"/>,
             <paramref name="a"/>
34         /// and <paramref name="b"/> are calculated.
35         /// </summary>
36         /// <exception cref="ArgumentException">Thrown when one of the
37         /// arguments provided to a method is not valid.</exception>
38         public HestonFormula(
39         double RiskFreeRate,
40         double InitialStockPrice,
41         double Kappa,
42         double Theta,
43         double Sigma,
44         double Rho,
45         double Nu)
46         {
```

```
47              this.RiskFreeRate = RiskFreeRate;
48              this.InitialStockPrice = InitialStockPrice;
49              this.Kappa = Kappa;
50              this.Theta = Theta;
51
52              if (Sigma < 0)
53                  throw new ArgumentException("--- Error: Sigma must not be
                      negative. ---");
54              this.Sigma = Sigma;
55
56              this.Rho = Rho;
57              this.Nu = Nu;
58              NumModelParams = 5;
59
60              b = new double[2];
61              u = new double[2];
62
63              a = Kappa * Theta;
64              b[0] = Kappa - Sigma * Rho;
65              b[1] = Kappa;
66              u[0] = 0.5;
67              u[1] = -0.5;
68
69          }
70
71          /// <summary>
72          /// Constructor for the class. Global variables <paramref name="u"/>,
                <paramref name="a"/>
73          /// and <paramref name="b"/> are calculated.
74          /// </summary>
75          /// <param name="Param"> Array arranged in the following order: Kappa,
                Theta, Sigma, Rho and Nu .</param>
76          /// <remarks>
77          /// No exception is thrown to prevent errors during calibration.
78          /// </remarks>
79          public HestonFormula(
80          double RiskFreeRate,
81          double InitialStockPrice,
82
83          double[] Param)
84          {
85              this.RiskFreeRate = RiskFreeRate;
86              this.InitialStockPrice = InitialStockPrice;
87
88              Kappa = Param[0];
89              Theta = Param[1];
90
91              Sigma = Param[2];
92
```

```
93              Rho = Param[3];
94              Nu = Param[4];
95              NumModelParams = 5;
96
97              b = new double[2];
98              u = new double[2];
99
100             a = Kappa * Theta;
101
102             b[0] = Kappa - Sigma * Rho;
103             b[1] = Kappa;
104             u[0] = 0.5;
105             u[1] = -0.5;
106
107         }
108
109
110         private Complex d(int j, double Phi)
111         {
112             Complex Aux1 = Complex.Pow(Rho * Sigma * Phi * i - b[j], 2);
113             Complex Aux2 = Sigma * Sigma * (2 * u[j] * Phi * i - Phi * Phi);
114             return Complex.Sqrt(Aux1 - Aux2);
115         }
116
117
118
119         private Complex g(int j, double Phi)
120         {
121             Complex Aux1 = b[j] - Rho * Sigma * Phi * i - d(j, Phi);
122             Complex Aux2 = b[j] - Rho * Sigma * Phi * i + d(j, Phi);
123
124             if (Aux2 == 0) throw new DivideByZeroException();
125
126             return Aux1 / Aux2;
127         }
128
129
130         private Complex C(int j, double Tau, double Phi)
131         {
132             Complex Aux1 = RiskFreeRate * Phi * i * Tau;
133             Complex Aux2 = (b[j] - Rho * Sigma * Phi * i - d(j, Phi)) * Tau;
134             Complex Aux3 = 1 - g(j, Phi) * Complex.Exp(-Tau * d(j, Phi));
135             Complex Aux4 = 1 - g(j, Phi);
136
137             if (Aux4 == 0) throw new DivideByZeroException();
138
139             return Aux1 + a / Complex.Pow(Sigma, 2) * (Aux2 - 2 * Complex.Log
                  (Aux3 / Aux4));
140         }
```

```
141
142
143         private Complex D(int j, double Tau, double Phi)
144         {
145             Complex Aux1 = b[j] - Rho * Sigma * Phi * i - d(j, Phi);
146             Complex Aux2 = 1 - Complex.Exp(-Tau * d(j, Phi));
147             Complex Aux3 = 1 - g(j, Phi) * Complex.Exp(-Tau * d(j, Phi));
148
149             if (Aux3 == 0) throw new DivideByZeroException();
150
151             return Aux1 / Complex.Pow(Sigma, 2) * (Aux2 / Aux3);
152
153         }
154
155         /// <summary>
156         /// Formula for the function Phi found in the documentation.
157         /// </summary>
158         /// <remarks>
159         /// There is a slight change with the initial formula shown in the
              documentation.
160         /// t is takes as 0 and T is taken as a variable.
161         /// </remarks>
162         private Complex PhiFunction(int j, double T, double x, double Phi)
163         {
164             return Complex.Exp(C(j, T, Phi) + D(j, T, Phi) * Nu + i * Phi * x);
165         }
166
167         /// <summary>
168         /// Formula for the P found in the documentation.
169         /// </summary>
170         /// <returns>
171         /// A double precision number.
172         /// </returns>
173         /// <param name="j">Index of arrays. Either 0 or 1.</param>
174         /// <param name="T">A double precision number.</param>
175         /// <param name="x">A double precision number.</param>
176         /// <remarks>
177         /// <para>The MathNet.Numerics.Integration.SimpsonRule is used to find
              the integral.</para
178         /// </remarks>
179         private double P(int j, double T, double x)
180         {
181             double Integral;
182             double IntervalBegin = 0.00001;
183             double IntervalEnd = 100;
184             int NumPartitions = 100;
185             Func<double, double> AuxFunction;
186
187
```

```
188                 AuxFunction = Phi => (Complex.Exp(-i * Phi * Math.Log
                      (StrikePrice)) * PhiFunction(j, T, x, Phi) / (i * Phi)).Real;
189                 Integral = SimpsonRule.IntegrateComposite
                      (AuxFunction,IntervalBegin, IntervalEnd, NumPartitions);
190
191
192             return 0.5 + (1 / Math.PI) * Integral;
193         }
194
195
196         /// <summary>
197         /// Formula for the c found in the documentation.
198         /// </summary>
199         /// <returns>
200         /// The European Call price of the option.
201         /// </returns>
202         /// <param name="StrikePrice">The strike price of the option.</param>
203         /// <param name="OptionExcercise">The exercise date of the option.</
              param>
204         /// <remarks>
205         /// There was a small problem with the initial formula. T
              (OptionExcercise) must replace t, except
206         /// in the exponential function when t is 0 and T remains the same.
207         /// </remarks>
208         public double CalculateCallPrice(double StrikePrice, double
              OptionExcercise)
209         {
210             this.StrikePrice = StrikePrice;
211             double Aux1 = InitialStockPrice * P(0, OptionExcercise, Math.Log
                  (InitialStockPrice));
212             double Aux2 = StrikePrice * Math.Exp(-RiskFreeRate *
                  (OptionExcercise));
213             double Aux3 = P(1, OptionExcercise, Math.Log(InitialStockPrice));
214             double Aux4 = Aux1 - Aux2 * Aux3;
215             return Aux4;
216         }
217
218         /// <summary>
219         /// Formula for the European Put option based on the Put-Call option
              parity.
220         /// </summary>
221         /// <returns>
222         /// The European Put price of the option.
223         /// </returns>
224         /// <param name="StrikePrice">The strike price of the option.</param>
225         /// <param name="OptionExcercise">The option exercise date of the
              option.</param>
226         public double CalculatePutPrice(double StrikePrice, double
              OptionExcercise)
```

```
227         {
228             double CallPrice = CalculateCallPrice(StrikePrice,
                  OptionExcercise);
229             return CallPrice + StrikePrice * Math.Exp(-RiskFreeRate *
                  OptionExcercise) - InitialStockPrice;
230         }
231
232         /// <summary>
233         /// Getter for the parameters optimized in task 2.3.
234         /// </summary>
235         /// <returns>
236         /// Array of doubles ordered as follows: Kappa, Theta, Sigma, Rho and
              Nu.
237         /// </returns>
238         /// <remarks>
239         /// Only for use in calibration.
240         /// </remarks>
241         public double[] ConvertCalibrationParamsToArray()
242         {
243             double[] ParamsArray = new double[NumModelParams];
244             ParamsArray[0] = Kappa;
245             ParamsArray[1] = Theta;
246             ParamsArray[2] = Sigma;
247             ParamsArray[3] = Rho;
248             ParamsArray[4] = Nu;
249             return ParamsArray;
250         }
251
252     }
253 }
254
```

```
using System;
using MathNet.Numerics.Distributions;
using System.Threading.Tasks;
using System.Threading;

namespace FinalProject
{
    /// <summary>
    /// Class for pricing European put and call options using Monte Carlo with the Heston model.
    /// <see href="final-project-almedina12/HestonModel.pdf">Reference.</see>
    /// </summary>
    /// <remarks>
    /// This is also a parent class for other types of options in the same framework,
    /// therefore the global variables are protected.
    /// </remarks>
    public class HestonMonteCarlo
    {
        protected double RiskFreeRate { get; set; }
        protected double StrikePrice { get; set; }
        protected double InitialStockPrice { get; set; }
        protected double Kappa { get; set; }
        protected double Theta { get; set; }
        protected double Sigma { get; set; }
        protected double Rho { get; set; }
        protected double Nu { get; set; }
        protected int TimeSteps { get; set; }

        protected double Alpha;
        protected double Beta;
        protected double Gamma;


        /// <summary>
        /// Constructor for the class. Global variables <paramref name="Alpha"/>, <paramref name="Beta"/>
        /// and <paramref name="Gamma"/> are calculated.
        /// </summary>
        /// <exception cref="ArgumentException">Thrown when one of the
        /// arguments provided to a method is not valid.</exception>
        public HestonMonteCarlo(
            double RiskFreeRate,
            double StrikePrice,
            double InitialStockPrice,
            double Kappa,
            double Theta,
            double Sigma,
            double Rho,
```

```
47              double Vu,
48              int TimeSteps =365)
49
50          {
51
52              this.RiskFreeRate = RiskFreeRate;
53              this.StrikePrice = StrikePrice;
54              this.InitialStockPrice = InitialStockPrice;
55              this.Kappa = Kappa;
56              this.Theta = Theta;
57
58              if (Sigma < 0)
59                  throw new ArgumentException("--- Error: Sigma must not be
                      negative. ---");
60              this.Sigma = Sigma;
61
62              if (Math.Abs(Rho) > 1)
63                  throw new ArgumentException("--- Error: Rho must be less than
                      1. ---");
64              this.Rho = Rho;
65
66              this.Nu = Vu;
67
68              if (TimeSteps <= 0)
69                  throw new ArgumentException("--- Error: The number of time
                      steps must be a positive integer. ---");
70              this.TimeSteps = TimeSteps;
71
72              if (2 * Kappa * Theta <= Math.Pow(Sigma, 2))
73                  throw new ArgumentException("--- Error: Feller Condition not
                      met. ---");
74
75              Alpha = (4 * Kappa * Theta - Math.Pow(Sigma, 2)) / 8;
76              Beta = -Kappa / 2;
77              Gamma = Sigma / 2;
78
79          }
80
81          /// <summary>
82          /// Generates samples from a Normal distribution with mean 0 and
              variance 1.
83          /// </summary>
84          /// <returns>
85          /// Array of doubles.
86          /// </returns>
87          /// <param name="NumSamples">Integer. Number of Samples.</param>
88          protected double[] GenerateSample(int NumSamples)
89          {
90              double[] Z = new double[NumSamples];
```

```
91              Normal.Samples(Z, 0, 1);
92              return Z;
93          }
94
95          /// <summary>
96          /// Payoff of the option. It is virtual to allow for it to be
              overridden.
97          /// </summary>
98          /// <returns>
99          /// Double precision number.
100         /// </returns>
101         /// <param name="Path"> Array of doubles that represent the path of the
               underlaying asset.</param>
102         /// <remarks>
103         /// In this case the payoff is that of a European Call option.
104         /// </remarks>
105         protected virtual double Payoff(double[] Path)
106         {
107             return Math.Max(0, Path[Path.Length - 1] - StrikePrice);
108         }
109
110         /// <summary>
111         /// Generates one path for the option, based on the Monte Carlo
              Formulas from the Heston Model.
112         /// </summary>
113         /// <returns>
114         /// The path, which is an array of doubles, represents the price as
              function of time.
115         /// </returns>
116         /// <param name="NumSamples"> Number of elements of the path.</param>
117         /// <param name="Tau"> Represents the marginal increment of time.</
              param>
118         /// <exception cref="DivideByZeroException">
119         /// Thrown when:
120         /// <code>
121         /// Beta * Tau == 1
122         /// </code>
123         /// </exception>
124         public double[] GeneratePath(int NumSamples, double Tau)
125         {
126             double[] SamplePaths = new double[NumSamples];
127             double[] X1 = GenerateSample(NumSamples);
128             double[] X2 = GenerateSample(NumSamples);
129             double DeltaZ1;
130             double DeltaZ2;
131             double Aux;
132             double Y = Math.Sqrt(Nu);
133
134             SamplePaths[0] = InitialStockPrice;
```

```
135
136             for (int k = 1; k < NumSamples; k++)
137             {
138                 DeltaZ1 = Math.Sqrt(Tau) * X1[k-1];
139                 DeltaZ2 = Math.Sqrt(Tau) * (Rho * X1[k-1] + Math.Sqrt(1 -
                      Math.Pow(Rho, 2)) * X2[k-1]);
140                 SamplePaths[k] = SamplePaths[k-1] + RiskFreeRate * SamplePaths
                      [k-1] * Tau + Y * SamplePaths[k-1] * DeltaZ1;
141
142                 if (Beta * Tau == 1) throw new DivideByZeroException();
143
144                 Aux = (Y + Gamma * DeltaZ2) / (2 * (1 - Beta * Tau));
145                 Y = Aux + Math.Sqrt(Aux * Aux + Alpha * Tau / (1 - Beta *
                      Tau));
146             }
147
148             return SamplePaths;
149
150         }
151
152         /// <summary>
153         /// Calculate the approximate Call price of an option using Monte Carlo
                and the Heston Model.
154         /// </summary>
155         /// <returns>
156         /// The approximate Call price of the option.
157         /// </returns>
158         /// <param name="OptionExcercise">The option exercise date of the
              option.</param>
159         /// <param name="NPaths">Number of paths used in the Monte Carlo
              algorithm.</param>
160         /// <remarks>
161         /// Given that in Normal.Samples() every sample is independent, we can
              use parallel threads.
162         /// </remarks>
163         public double CalculatePrice(double OptionExercise, int NPaths = 10000)
164         {
165             if (OptionExercise <= 0)
166                 throw new ArgumentException("--- Error: Option exercise date
                      must be greater than 0. ---");
167
168             if (NPaths <= 0)
169                 throw new ArgumentException("--- Error: The number of paths
                      must be a positive integer. ---");
170
171             int NumSamples = (int)Math.Ceiling(OptionExercise * TimeSteps);
172             double Tau = OptionExercise / NumSamples;
173
174             double Aux = 0;
```

```
175             double[] Path;
176
177
178             Parallel.For(0, NPaths, i => {
179                 Path = GeneratePath(NumSamples,Tau);
180                 Interlocked.Exchange(ref Aux, Payoff(Path) + Aux);
181             });
182             return Math.Exp(-RiskFreeRate * OptionExercise) * Aux / NPaths;
183         }
184
185
186         /// <summary>
187         /// Formula for the Put option based on the Put-Call option parity.
188         /// </summary>
189         /// <returns>
190         /// The approximate Put price of the option.
191         /// </returns>
192         /// <param name="OptionExcercise">The option exercise date of the
              option.</param>
193         /// <param name="NPaths">Number of paths used in the Monte Carlo
              algorithm.</param>
194         public virtual double CalculatePutPrice(double OptionExercise,  int
              NPaths = 10000)
195         {
196             double CallPrice = CalculatePrice(OptionExercise, NPaths);
197             return CallPrice + StrikePrice * Math.Exp(-RiskFreeRate *
                  OptionExercise) - InitialStockPrice;
198         }
199
200
201
202
203     }
204 }
205
```

```
1  using System;
2  using System.Linq;
3
4
5  namespace FinalProject
6  {
7      /// <summary>
8      /// Subclass of HestonMonteCarlo. Seeks to find the approximated price of
         Lookback type options
9      /// relaying on the methods from HestonMonteCarlo. The only major change is
         the Payoff function.
10     /// </summary>
11     /// <remarks>
12     /// The strike price is irrelevant for the pricing of Lookback options.
13     /// </remarks>
14     public class LookbackOptions : HestonMonteCarlo
15     {
16         public LookbackOptions(
17             double RiskFreeRate,
18             double InitialStockPrice,
19             double Kappa,
20             double Theta,
21             double Sigma,
22             double Rho,
23             double Nu,
24             int TimeSteps) :
25             base(RiskFreeRate,
26                 1,
27                 InitialStockPrice,
28                 Kappa,
29                 Theta,
30                 Sigma,
31                 Rho,
32                 Nu,
33                 TimeSteps)
34         {
35         }
36
37         /// <summary>
38         /// Payoff of the option for the Lookback Options. Overrides the method
             from the parent class.
39         /// </summary>
40         /// <returns>
41         /// Double precision number.
42         /// </returns>
43         /// <param name="Path"> Array of doubles that represent the path of the
             underlaying asset.</param>
44         protected override double Payoff(double[] Path)
45         {
```

```
46              return Math.Max(0, Path[Path.Length - 1] - Path.Min());
47          }
48
49          /// <summary>
50          /// Since there is no differentiation between Call and Puts for Lookback
               Options it calculates the same price for both.
51          /// </summary>
52          /// <param name="OptionExercise"></param>
53          /// <param name="NPaths"></param>
54          /// <returns>The Same price as CalculatePrice </returns>
55          public override double CalculatePutPrice(double OptionExercise, int
              NPaths = 10000)
56          {
57              return CalculatePrice(OptionExercise, NPaths);
58          }
59
60
61      }
62  }
63
```

```
1  using System;
2  using FinalProject;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7  using HestonModel;
8
9  namespace HestonCmdLine
10 {
11     /// <summary>
12     /// This is the main class of the Console app where all the other functions
       are called. I used this for debugging.
13     /// The purpose of this is to expose all of the functionality implemented in
          all the projects.
14     /// </summary>
15     class Program
16     {
17         static void Main(string[] args)
18         {
19             //Methods in the result class:
20
21             //---------------------------------
22             //Result.HestonFormulaResult();
23             //Result.HestonMonteCarloResult();
24             //Result.AsianOptionsResult();
25             //Result.LookbackOptionsResult();
26             //Result.CliquetOptionResult();
27            // Result.EverestOptionsResult();
28             //---------------------------------
29
30             //CheckingCalibration.CheckCalibration();
31
32             //Methods for the correct set up.
33
34             //---------------------------------
35             //CorrectSetUp.FormulaSetUp();
36             //CorrectSetUp.MonteCarloSetUp();
37             //CorrectSetUp.LookbackOptionSetUp();
38             //CorrectSetUp.AsianOptionsSetUp();
39             //CorrectSetUp.CalibrationSetUp();
40             //---------------------------------
41
42         }
43     }
44 }
45
```

```
1  using System;
2  using FinalProject;
3
4
5  namespace HestonCmdLine
6  {
7
8      /// <summary>
9      /// This is the class I used to expose most of the functionality I
         implemented through the console App.
10     /// </summary>
11     public class Result
12     {
13         /// <summary>
14         /// This function creates an instance of HestonFormula and prints out
             the price of several European Call Options.
15         /// </summary>
16         public static void HestonFormulaResult()
17         {
18             double RiskFreeRate = 0.025;
19             double StrikePrice = 100;
20             double[] OptionExercise = { 1, 2, 3, 4, 15 };
21             double Kappa = 1.5768;
22             double Theta = 0.0398;
23             double Sigma = 0.5751;
24             double Rho = -0.5711;
25             double InitialStockPrice = 100;
26             double Nu = 0.0175;
27
28
29             HestonFormula Formula = new HestonFormula(RiskFreeRate,
                 InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu);
30
31             for (int h = 0; h < OptionExercise.Length; h++)
32             {
33                 double HestonFormula = Formula.CalculateCallPrice(StrikePrice,
                     OptionExercise[h]);
34
35                 Console.WriteLine(HestonFormula);
36
37             }
38             Console.ReadKey();
39         }
40
41         /// <summary>
42         /// This function creates an instance of HestonMonteCarlo and
             calculates the price for several oEuropean Call Options.
43         /// </summary>
44         public static void HestonMonteCarloResult()
```

```
45          {
46              Console.WriteLine("Monte Carlo:");
47              double RiskFreeRate = 0.1;
48              double StrikePrice = 100;
49              double[] OptionExercise = { 1, 2, 3, 4, 15 };
50              double Kappa = 2;
51              double Theta = 0.06;
52              double Sigma = 0.4;
53              double Rho = 0.5;
54
55              double InitialStockPrice = 100;
56              double Nu = 0.04;
57              int TimeSteps = 1000;
58              int Paths = 10000;
59
60
61              for (int h = 0; h < OptionExercise.Length; h++)
62              {
63
64                  HestonMonteCarlo MonteCarlo = new HestonMonteCarlo
                      (RiskFreeRate, StrikePrice, InitialStockPrice, Kappa, Theta,
                      Sigma, Rho, Nu, TimeSteps);
65                  double Price = MonteCarlo.CalculatePrice(OptionExercise[h],
                      Paths);
66                  Console.WriteLine(Price);
67              }
68
69              Console.ReadKey();
70          }
71
72          /// <summary>
73          /// This function creates an instance of the class AsianOptions and
              prints out the Call price for various sets of dates.
74          /// </summary>
75          public static void AsianOptionsResult()
76          {
77              double RiskFreeRate = 0.1;
78              double StrikePrice = 100;
79              double[] OptionExercise = { 1, 2, 3 };
80              double[] Aux1 = { 0.75, 1 };
81              double[] Aux2 = { 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75 };
82              double[] Aux3 = { 1, 2, 3 };
83              double[][] Dates = { Aux1, Aux2, Aux3 };
84              double Kappa = 2;
85              double Theta = 0.06;
86              double Sigma = 0.4;
87              double Rho = 0.5;
88
89              double InitialStockPrice = 100;
```

```
90              double Nu = 0.04;
91              int TimeSteps = 365;
92              int Paths = 10000;
93
94
95              for (int h = 0; h < Dates.Length; h++)
96              {
97
98                  AsianOptions Asian = new AsianOptions(Dates[h],RiskFreeRate,
                      StrikePrice, InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu,
                       TimeSteps);
99                  double Price = Asian.CalculatePrice(OptionExercise[h],Paths);
100                 Console.WriteLine(Price);
101             }
102
103
104             Console.ReadKey();
105
106         }
107
108         /// <summary>
109         /// This function creates an instance of the class LookbackOptions and
              prints out the price for various maturities.
110         /// </summary>
111         public static void LookbackOptionsResult()
112         {
113             double RiskFreeRate = 0.1;
114             double[] OptionExercise = { 1, 3, 5, 7, 9 };
115             double Kappa = 2;
116             double Theta = 0.06;
117             double Sigma = 0.4;
118             double Rho = 0.5;
119
120             double InitialStockPrice = 100;
121             double Nu = 0.04;
122             int TimeSteps = 365;
123             int Paths = 10000;
124
125
126             for (int h = 0; h < OptionExercise.Length; h++)
127             {
128
129                 LookbackOptions Asian = new LookbackOptions( RiskFreeRate,
                      InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu, TimeSteps);
130                 double Price = Asian.CalculatePrice(OptionExercise[h], Paths);
131                 Console.WriteLine(Price);
132             }
133
134
```

```
135             Console.ReadKey();
136
137         }
138         /// <summary>
139         /// This function creates an instance of CliquetOption and prints out
              the price for a set of fixing dates.
140         /// </summary>
141         public static void CliquetOptionResult()
142         {
143             double RiskFreeRate = 0.1;
144             double[] FixingDates = { 1, 2, 3};
145             int Lambda = 1;
146             double StrikePrice = 90;
147             double InitialStockPrice = 100;
148             double PayoutRate = 0.05;
149             double Sigma = 0.2;
150
151
152             CliquetOptions Option = new CliquetOptions(RiskFreeRate,
                  FixingDates, Lambda, Sigma, StrikePrice, InitialStockPrice,
                  PayoutRate);
153             double Result = Option.CalculatePrice();
154             Console.WriteLine(Result);
155             Console.ReadKey();
156
157
158         }
159         /// <summary>
160         /// This function creates an instance of EverestOptions using data from
               several different assets. It prints out the price
161         /// of the option.
162         /// </summary>
163         public static void EverestOptionsResult()
164         {
165             double RiskFreeRate = 0.1;
166             double[] StrikePrice = { 100, 110, 120, 100, 110, 120 };
167             double OptionExercise = 15;
168
169             double[] Kappa = { 2, 2.5, 3.5, 2, 2.5, 3.5 };
170             double[] Theta = { 0.6, 0.6, 0.7, 0.6, 0.6, 0.7 };
171             double[] Sigma = { 0.4, 0.3, 0.02, 0.4, 0.3, 0.02 };
172             double[] Rho = { 0.7, 0.12, 0.394, 0.7, 0.12, 0.394 };
173
174             double[] InitialStockPrice = { 98, 100, 115, 98, 100, 115 };
175             double[] Nu = { 0.07, 0.02, 0.09, 0.07, 0.02, 0.09 };
176
177
178             EverestOptions Everest = new EverestOptions(RiskFreeRate);
179
```

```
180             for (int h = 0; h < StrikePrice.Length; h++)
181                 Everest.AddAsset(StrikePrice[h], InitialStockPrice[h], Kappa
                      [h], Theta[h], Sigma[h], Rho[h], Nu[h]);
182
183             double Price = Everest.CalculatePrice(OptionExercise);
184             Console.WriteLine(Price);
185             Console.ReadKey();
186
187         }
188
189
190
191     }
192 }
193
```

```
1  using System;
2  using Microsoft.VisualStudio.TestTools.UnitTesting;
3  using FinalProject;
4  using MathNet.Numerics.Distributions;
5
6  namespace FinalProjectTests
7  {
8      /// <summary>
9      /// I created this class to test the methods: HestonFormula,
         HestonMonteCarlo and HestonCalibration. I ran them during the
10     /// development to make sure the code was functioning properly.
11     /// </summary>
12     [TestClass]
13     public class HestonTestsFinal
14     {
15         /// <summary>
16         /// The purpose of this test was to see whether HestonMonteCarlo and
             HestonPrice were producing the same output
17         /// for the same parameters.
18         /// </summary>
19         [TestMethod]
20         public void InstanciateHestonFormulaAndMonteCarloAreEqual()
21         {
22             double RiskFreeRate = 0.1;
23             double StrikePrice = 100;
24             double OptionExercise = 15;
25             double Kappa = 2;
26             double Theta = 0.06;
27             double Sigma = 0.4;
28             double Rho = 0.5;
29
30             double InitialStockPrice = 100;
31             double Nu = 0.04;
32             int TimeSteps = 365;
33             int Paths = 100000;
34
35             double Accuracy = 7;
36
37
38             double PriceHestonFormula;
39             double PriceMonteCarlo;
40
41             HestonFormula Formula = new HestonFormula(RiskFreeRate,
                 InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu);
42             HestonMonteCarlo MonteCarlo = new HestonMonteCarlo(RiskFreeRate,
                 StrikePrice, InitialStockPrice, Kappa, Theta, Sigma, Rho, Nu,
                 TimeSteps);
43
44             PriceHestonFormula = Formula.CalculateCallPrice(StrikePrice,
```

```
                OptionExercise);
45              PriceMonteCarlo = MonteCarlo.CalculatePrice(OptionExercise,Paths);
46              Assert.AreEqual(PriceHestonFormula, PriceMonteCarlo, Accuracy);
47
48
49
50          }
51
52          /// <summary>
53          /// The purpose of this test was to see if the calibration was producing
                a small error.
54          /// </summary>
55          [TestMethod]
56          public void InstanciateHestonCalibrationErrorSmallAndFinisedOK()
57          {
58              double RiskFreeRate = 0.025;
59              double InitialStockPrice = 100;
60              double Kappa = 0.5;
61              double Theta = 0.01;
62              double Sigma = 0.2;
63              double Rho = 0.1;
64              double Nu = 0.4;
65              double Accuracy = 1 / 1000;
66              int MaxIterations = 1000;
67              double[] StrikePrices = new double[] { 80, 90, 80, 100, 100 };
68              double[] OptionExerciseTimes = new double[] { 1, 1, 2, 2, 1.5 };
69              double[] Prices = new double[] { 25.72, 18.93, 30.49, 19.36,
                  16.58 };
70
71
72              HestonCalibration Calibrator = new HestonCalibration(RiskFreeRate,
                  InitialStockPrice, Accuracy, MaxIterations);
73
74              Calibrator.SetGuessParameters(Kappa, Theta, Sigma, Rho, Nu);
75
76              for (int i = 0; i < Prices.Length; ++i)
77              {
78                  Calibrator.AddObservedOption(OptionExerciseTimes[i],
                      StrikePrices[i], Prices[i]);
79              }
80              Calibrator.Calibrate();
81
82
83              double Error = 0;
84              FinalProject.CalibrationOutcome Outcome =
                  FinalProject.CalibrationOutcome.NotStarted;
85              Calibrator.GetCalibrationStatus( ref Outcome, ref Error);
86              Assert.AreEqual(0, Error, 1);
87
```

```
88
89          }
90
91
92
93
94
95      }
96  }
97
```

```
1  using System;
2  using System.Collections;
3
4  namespace FinalProject
5  {
6
7      /// <summary>
8      /// Subclass of HestonMonteCarlo. Seeks to find the approximated price of
         Call and Put Asian type options
9      /// relaying on the methods from HestonMonteCarlo.
10     /// </summary>
11     public class AsianOptions: HestonMonteCarlo {
12
13         protected internal IEnumerable Dates;
14         protected internal bool IsCall = true;
15
16         public AsianOptions(IEnumerable Dates,
17             double RiskFreeRate,
18             double StrikePrice,
19             double InitialStockPrice,
20             double Kappa,
21             double Theta,
22             double Sigma,
23             double Rho,
24             double Nu,
25             int TimeSteps) :
26             base(RiskFreeRate,
27                 StrikePrice,
28                 InitialStockPrice,
29                 Kappa,
30                 Theta,
31                 Sigma,
32                 Rho,
33                 Nu,
34                 TimeSteps)
35         {
36             this.Dates = Dates;
37         }
38
39         /// <summary>
40         /// Payoff of the option for Asian Call Options. Overrides the method
             from the parent class.
41         /// </summary>
42         /// <returns>
43         /// Double precision number.
44         /// </returns>
45         /// <param name="Path"> Array of doubles that represents the path of the
              underlaying asset.</param>
46         protected override double Payoff(double[] Path)
```

```
47          {
48              double Sum = 0;
49              double Cont = 0;
50
51              foreach(double T in Dates)
52              {
53                  Sum = Sum + Path[(int)Math.Floor(T * TimeSteps - 1)];
54                  Cont++;
55
56              }
57              Sum = Sum / Cont;
58
59              double Result;
60
61              if (IsCall)
62                  Result = Math.Max(0, Sum - StrikePrice);
63              else
64                  Result = Math.Max(0, StrikePrice - Sum);
65
66
67              return Result;
68          }
69
70          /// <summary>
71          /// Payoff of the option for Asian Put Options. Overrides the method
              from the parent class.
72          /// </summary>
73          /// <returns>
74          /// Double precision number.
75          /// </returns>
76          /// <param name="NPaths"> Number of path that will be created.</param>
77          /// <param name="OptionExercise"> The maturity/exercise date of the
              option..</param>
78          public override double CalculatePutPrice(double OptionExercise, int
              NPaths = 10000)
79          {
80              IsCall = false;
81              double Result = CalculatePrice(OptionExercise, NPaths);
82              IsCall = true;
83              return Result;
84          }
85
86
87
88
89      }
90
91
92
```

```
93
94  }
95
```