

Object-Oriented Programming with Applications: Final Project

Adam Keogh, S1876558

January 2019

Contents

A quick note on code layout	1
Task 2.2	2
Task 2.3	2
Task 2.4	2
Task 2.5	3
Task 2.6	4
Task 2.7	6
Task 2.8	6
Additional investigation	6
Code	9
Options.cs class	9
OptionsMC.cs class	11
MCPaths.cs class	19
Calibrator.cs class	22
InterfaceFill.cs class	25
OtherInterfaceFill.cs class	26
AnotherInterfaceFill.cs class	27
FinalInterfaceFill.cs class	28
Heston.cs	28

A quick note on code layout

After much thought, I decided to layout my code as follows. I have a class called Options.cs which is used to price European options using the Heston model. This class is then used within the Calibrator.cs class which is used to calibrate the Heston model to real world data. On top of these classes, I have the class MCPaths.cs, which is used to generate the paths (either as standard or with anithetic sampling) for the Monte Carlo pricing class, which is called OptionsMC.cs. This class is used to price European, Asian and lookback calls, both with standard Monte Carlo and with parallel Monte Carlo using anithetic sampling. Finally, I wrote several classes which implement some of the interfaces which we were given. The first such class, InterfaceFill.cs, implements

the base level interfaces. OtherInterfaceFill.cs and AnotherInterfaceFill.cs are used to implement interfaces which are derived from these base level interfaces. FinalInterfaceFill.cs was used to implement other interfaces, but was only necessary for my own testing of the Heston.cs class.

Task 2.2

Having written code to price European put and call options in the Heston model I was able to fill out the below table as requested, where the parameters used were

$$\begin{aligned} r &= 2.5\% \\ \theta^* &= 3.98\%, \kappa^* = 157.68\%, \sigma = 57.51\%, \rho = -57.11\%, v = 1.75\% \\ S &= 100. \end{aligned} \tag{1}$$

Strike K	Option exercise T	Price $C(0, S, v)$
100	1	7.27
100	2	11.74
100	3	15.48
100	4	18.77
100	15	43.17

Table 1: Prices of "at the money" call options in the Heston model using Heston formula.

Task 2.3

Having written code to price European put and call options in the Heston model using a Monte Carlo Algorithm I was able to fill out the below table as requested, where the parameters used were

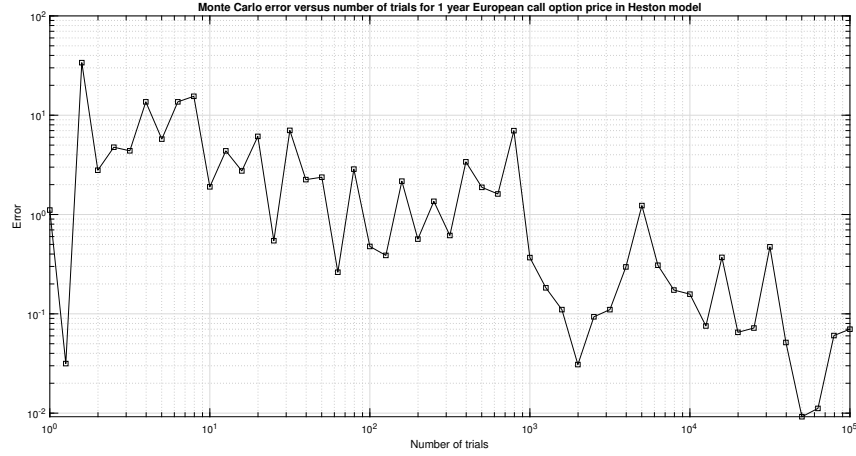
$$\begin{aligned} r &= 10\% \\ \theta^* &= 6\%, \kappa^* = 200\%, \sigma = 40\%, \rho = 50\%, v = 4\% \\ S &= 100. \end{aligned} \tag{2}$$

Strike K	Option exercise T	Price $C(0, S, v)$
100	1	13.6
100	2	22.6
100	3	30.1
100	4	37.2
100	15	77.9

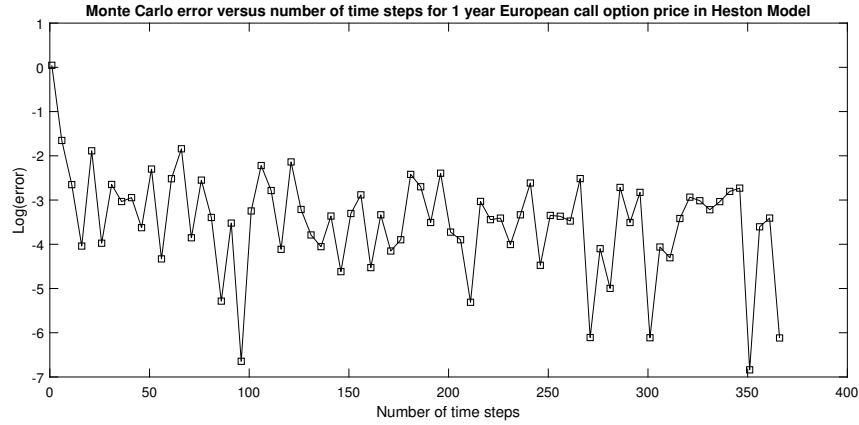
Table 2: Prices of "at the money" call options in the Heston model using a Monte Carlo method.

Task 2.4

The above graph shows the convergence of the Monte Carlo pricing and the Heston formula pricing of a 1 year European Call option as the number of paths used increases. The number of time steps used was fixed at 365 and the Heston parameters were as seen in (2). The graph below shows



the convergence of the pricing when the number of trials was fixed at 100000 and the time steps increase. It is clear that the number of trials used is more important for the accuracy of our Monte Carlo, once we are working with a reasonable number of time steps per year



Task 2.5

Having used the BFGS algorithm to create a calibrator for the Heston model using the market data given in the question I was able to achieve the following results.

Parameters	Calibrated value
κ^*	1.60287
θ^*	0.10323
σ	0.45889
ρ	-0.47647
v	0.09975

Table 3: Calibrated parameters using accuracy 1×10^{-3} , maxIts = 1000, S = 100 and parameters (1) as initial guess.

What I found was that with accuracy 1×10^{-3} the calibrated parameter values would vary dependent on the initial guess parameters, while with accuracy 1×10^{-15} they would converge to the below parameters from most reasonable starting points which I tried. The calibration returned

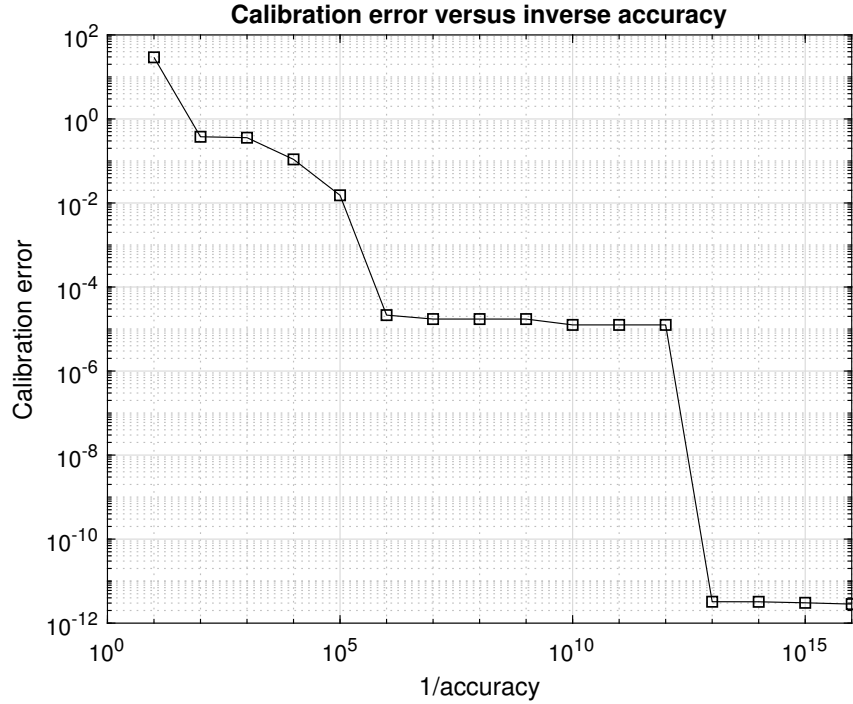
FinishedOk in the cases of both of these tables, with mean square error to the order of 0.01 in the first case and 1×10^{-12} in the second, as I will explore in the next task.

Parameters	Calibrated value
κ^*	1.94372
θ^*	0.09980
σ	0.35828
ρ	-0.56084
v	0.09891

Table 4: Calibrated parameters using accuracy 1×10^{-15} , maxIts = 1000, S = 100 and parameters (1) as initial guess.

Task 2.6

As I have previously noted, I found that when I used a smaller accuracy parameter my parameters would calibrate to the same values, while with larger accuracy it would depend on the initial guess. The below graph shows the mean squared error of the calibrated model when different accuracies were used, with initial guess and market data as in Task 2.5.



After calibrating the model with the two accuracy levels in Task 2.5, I used the parameters in Tables 3 and 4 to test the calibrated models against the market data we were given. As expected given the tiny error of the model with parameters from Table 4, its prices agreed to several decimal places with the market data. This wasn't the case with the previous model, as can be seen below.

Strike K	Option exercise T	Market Call Price
80	1	25.72
90	1	18.93
80	2	30.49
100	2	19.36
100	1.5	16.58

Table 5: Market data from the question

Strike K	Option exercise T	Market Call Price
80	1	25.80
90	1	18.95
80	2	30.59
100	2	19.31
100	1.5	16.52

Table 6: Calibrated model prices with parameters from Table 3

I was also able to find situations where the calibration did not work as it should. For example, using parameters (1), I created market data using the Heston formula using the same strike prices and maturities as in the market data in the question. I then, from several initial guesses, with accuracy 1×10^{-15} , observed whether the calibrated model's parameters matched the actual parameters I had used. In many cases they did, but in certain cases, whilst the calibrator converged, it was not to the right parameters. For example, with initial guess $\kappa^* = \theta^* = \sigma = \rho = v = 10\%$, the calibrator converged to the following parameters, with a mean squared error of 0.01 :

$$\theta^* = -0.31\%, \kappa^* = -491.12\%, \sigma = 50.25\%, \rho = -0.83\%, v = 2.92\%$$

And while we can see that with these parameters the prices will roughly match the market data, it is clear that with any other strike prices and maturities this calibrated model will be way off.

This is a case of a "local minimum" for the Heston calibration function, whereby it converges,

Strike K	Option exercise T	Model price	Market Price
80	1	23.00	23.03
90	1	14.47	14.51
80	2	26.24	26.19
100	2	11.69	11.74
100	1.5	9.68	9.62

Table 7: Calibrated model prices with parameters from Table 3

but not to the right parameters! Using very large parameters in the original guess also lead to situations like this, with calibration being successful but with huge mean squared error between the calibrated model prices and the market data.

Task 2.7

Having written code to price Asian arithmetic options in the Heston model using a Monte Carlo algorithm I was able to fill out the below table as requested, using parameters (2).

Strike K	Option exercise T	T_1, \dots, T_m	MC Price
100	1	0.75, 1.00	12.0
100	2	0.25, 0.50, 0.75, 1.00, 1.25, 1.50, 1.75	11.3
100	3	1.0, 2.0, 3.0	19.9

Table 8: Prices of "at the money" Asian call options in the Heston model using a Monte Carlo method.

Task 2.8

Having written code to price lookback options in the Heston model using a Monte Carlo algorithm I was able to fill out the below table as requested, using parameters (2).

Option Exercise T	MC Price
1	19.1
3	36.7
5	49.5
7	58.9
9	66.9

Table 9: Prices of lookback options in the Heston model using a Monte Carlo method.

Additional investigation

One means of additional investigation I undertook was to write code for parallel implementation of some of our Monte Carlo methods. I used `Parallel.For` loops in my code to allow parallel implementation and then used the `Interlocked.Exchange` command to update variables so as to ensure that everything happened in the right order. Interestingly, I found that this consistently reduced computation time for European and Asian options by 15-30%, but, while on average it reduced computation time for lookback options there was a lot more variance in the times taken here. I priced the same option 5 times in a row with both methods using the `Stopwatch` class in `C#` to get computation times.

Normal MC	Parallel MC
7.4789	6.1622
7.1997	6.0032
7.3187	6.0654
7.7478	6.3696
6.9976	5.2767

Table 10: 5 computation time in seconds for price of 1 year European call with 100000 trials and 365 time steps.

The average time for normal MC was 7.3485 seconds compared to 5.9754 seconds for parallel MC, an improvement of 18.69%.

Normal MC	Parallel MC
6.1713	4.3416
5.8967	3.7910
5.9571	3.9080
5.9075	4.0570
5.9759	4.3758

Table 11: Computation time in seconds for price of 1 year Asian call with 100000 trials, 365 time steps, and observation times 0.25, 0.5, 0.75, 1.

The average time for normal MC was 5.9817 seconds compared to 4.0947 seconds for parallel MC, an improvement of 31.55%.

Normal MC	Parallel MC
27.70	27.71
29.41	22.44
27.76	24.36
28.02	21.78
24.80	27.25

Table 12: Computation time in seconds for price of 1 year lookback option with 100000 trials and 365 time steps.

The average time for normal MC was 27.5395 seconds compared to 24.7094 seconds for parallel MC, an improvement of 10.28%.

The other additional investigation I undertook was to implement variance reduction methods for Monte Carlo. I used anithetic sampling whereby you generate two paths at once. You first sample from the normal distribution and generate a path as usual. The difference is that you then change the sign of your normal sample and use this to generate a second path. In theory this method has two benefits. Firstly you need only sample from the normal distribution half as many times to generate N paths as you would with normal sampling. Secondly it is often the case that this kind of sampling reduces the variance of Monte Carlo prices. Intuitively we ensure that the normal samples for each trial are stratified, with exactly as many coming from the negative side of the normal distribution as from the positive. Mathematically if we have two samples Y_1 and Y_2 and some function f it follows that

$$Var\left(\frac{f(Y_1) + f(Y_2)}{2}\right) = \frac{Var(f(Y_1)) + Var(f(Y_2)) + 2Cov(f(Y_1), f(Y_2))}{4}$$

and so if $Cov(f(Y_1), f(Y_2))$ is negative we get a reduction in variance, which is often the case for antithetic samples ($Y_1 = -Y_2$). This was the case in my implementation of this method in the Heston model for pricing European options. Antithetic sampling reduced the variance of the Monte Carlo prices by quite a large factor. I priced the same 1 year European call option 25 times with both methods, computed the variance of their prices and repeated 5 times.

Normal MC	Antithetic MC
0.020	0.005
0.013	0.009
0.024	0.006
0.015	0.013
0.110	0.009

Table 13: Variance of 25 pricings of a 1 year European call option with 100000 trials and 365 price steps.

This is useful as it makes our method more accurate. So to summarise, in my additional investigation, I used parallelisation and antithetic sampling to successfully reduce both the computational time and variance of my Monte Carlo options pricing.

Code

Options.cs class

```
1
2 using System;
3 using MathNet.Numerics.Integration;
4 using System.Numerics;
5 using System.Collections.Generic;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9
10 namespace HestonCalibrationAndPricing
11 {
12     /// <summary>
13     /// This class prices options within the Heston model
14     /// </summary>
15     public class Options
16     {
17         public const int numberParams = 5;
18         public const int kappaIndex = 0;
19         public const int thetaIndex = 1;
20         public const int sigmaIndex = 2;
21         public const int rhoIndex = 3;
22         public const int vIndex = 4;
23
24         private double r;
25         private double kappaStar;
26         private double thetaStar;
27         private double sigma;
28         private double rho;
29         private double v;
30         private double S;
31
32
33         public Options(double r, double S, double kappaStar, double thetaStar,
34             double sigma, double rho, double v)
35         {
36             if (r <= 0 || S <= 0 || sigma <= 0 || v <= 0)
37             {
38                 throw new System.ArgumentException("r, S, sigma, v must be
39                     positive");
40             }
41             this.r = r; this.S = S; this.kappaStar = kappaStar; this.thetaStar
42                 = thetaStar;
43             this.sigma = sigma; this.rho = rho; this.v = v;
44         }
45
46         public Options(double r, double S, double[] parameters)
47         {
48             this.r = r; this.S = S;
49             kappaStar = parameters[kappaIndex];
50             thetaStar = parameters[thetaIndex];
51             sigma = parameters[sigmaIndex];
52             rho = parameters[rhoIndex];
53             v = parameters[vIndex];
54
55             if (r <= 0 || S <= 0 || sigma <= 0 || v <= 0)
56             {
57                 throw new System.ArgumentException("r, S, sigma, v must be
58                     positive");
59             }
60         }
61
62         /// <summary>
```

```

60     /// Prices a European call option within the Heston model.
61     /// </summary>
62     /// <param name = "T">The maturity date of the option in years.</param>
63     /// <param name = "K">The options' strike price.</param>
64     /// <returns>Option price.</returns>
65     public double EuropeanCallPrice(double T, double K)
66     {
67         if (T < 0 || K < 0)
68         {
69             throw new System.ArgumentException("T, K must be non-negative");
70         }
71
72         // set up parameters
73         double[] b = { kappaStar - rho * sigma, kappaStar };
74         double[] u = { 0.5, -0.5 };
75         double a = kappaStar * thetaStar;
76         Complex i = new Complex(0, 1);
77
78         // function implementing part of Heston formula
79         Func<int, double, double> RealP = (j, phi) =>
80         {
81
82             Complex temp1 = new Complex(-b[j], rho * sigma * phi);
83             Complex tempp1 = new Complex(-phi * phi, 2 * u[j] * phi);
84             Complex d = Complex.Pow(temp1 * temp1 - sigma * sigma * tempp1,
85                                     0.5);
86             Complex g = (b[j] - rho * sigma * phi * i - d) * Complex.Pow(b[j]
87                                     - rho * sigma * phi * i + d, -1);
88
89             Complex c = r * phi * T * i + (a / (sigma * sigma)) * ((b[j] -
90                                     rho * sigma * phi * i - d) * T - 2 * Complex.Log((1 - g *
91                                     Complex.Exp(-T * d)) / (1 - g)));
92             Complex bigD = ((b[j] - rho * sigma * phi * i - d) / (sigma *
93                                     sigma)) * ((1 - Complex.Exp(-T * d)) / (1 - g * Complex.Exp
94                                     (-T * d)));
95             Complex littlePhi = Complex.Exp(c + bigD * v + phi * i * Math.
96                                     Log(S));
97             Complex value = Complex.Exp(-i * phi * Math.Log(K)) * littlePhi
98                                     / (i * phi);
99             return value.Real;
100         };
101
102         double[] P = new double[2];
103
104         // integrate with appropriate number of steps and length to
105         // approximate infinite integral
106         P[0] = 0.5 + (1.0 / Math.PI) * SimpsonRule.IntegrateComposite(x =>
107             RealP(0, x), 0.000001, 50, 100);
108         P[1] = 0.5 + (1.0 / Math.PI) * SimpsonRule.IntegrateComposite(x =>
109             RealP(1, x), 0.000001, 50, 100);
110
111         return S * P[0] - K * Math.Exp(-r * T) * P[1];
112     }
113
114     /// <summary>
115     /// Prices a European put option within the Heston model.
116     /// </summary>
117     /// <param name = "T">The maturity date of the option in years.</param>
118     /// <param name = "K">The options' strike price.</param>
119     /// <returns>Option price.</returns>
120     public double EuropeanPutPrice(double T, double K)
121     {
122         if (T < 0 || K < 0)
123         {

```

```

113         throw new System.ArgumentException("T, K must be non-negative")
114         ;
115     }
116     // put call parity
117     return EuropeanCallPrice(T, K) - S + K * Math.Exp(-r * T);
118 }
119
120 /// <summary>
121 /// Forms an array of the model parameters.
122 /// </summary>
123 /// <returns>Model parameters.</returns>
124 public double[] ParamsAsArray()
125 {
126     double[] paramsArray = new double[Options.numberParams];
127     paramsArray[kappaIndex] = kappaStar;
128     paramsArray[thetaIndex] = thetaStar;
129     paramsArray[sigmaIndex] = sigma;
130     paramsArray[rhoIndex] = rho;
131     paramsArray[vIndex] = v;
132     return paramsArray;
133 }
134
135 }
136 }

```

OptionsMC.cs class

```

1 using System;
2 using MathNet.Numerics.Distributions;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7 using System.Threading;
8
9 namespace HestonCalibrationAndPricing
10 {
11     /// <summary>
12     /// This class prices options within the Heston model using Monte Carlo
13     /// methods.
14     /// </summary>
15     public class OptionsMC
16     {
17         public const int kappaIndex = 0;
18         public const int thetaIndex = 1;
19         public const int sigmaIndex = 2;
20         public const int rhoIndex = 3;
21         public const int vIndex = 4;
22
23         private double r;
24         private double K;
25         private double kappaStar;
26         private double thetaStar;
27         private double sigma;
28         private double rho;
29         private double v;
30         private double S;
31
32         public OptionsMC(double r, double K, double kappaStar, double thetaStar
33             , double sigma, double rho, double v, double S)
34         {
35             if (2 * kappaStar * thetaStar <= sigma * sigma)
36             {
37                 throw new System.ArgumentException("Feller condition violated."
38                     );
39             }
40         }
41     }
42 }

```

```

37
38         if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
39         {
40             throw new System.ArgumentException("r, K, sigma, S must be
41                 positive");
42         }
43         this.r = r;
44         this.K = K; this.kappaStar = kappaStar; this.thetaStar = thetaStar;
45         this.sigma = sigma; this.rho = rho; this.v = v; this.S = S;
46     }
47
48     public OptionsMC(double r, double kappaStar, double thetaStar, double
49         sigma, double rho, double v, double S)
50     {
51         if (2 * kappaStar * thetaStar <= sigma * sigma)
52         {
53             throw new System.ArgumentException("Feller condition violated."
54                 );
55         }
56         if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
57         {
58             throw new System.ArgumentException("r, K, sigma, S must be
59                 positive");
60         }
61         this.r = r;
62         this.kappaStar = kappaStar; this.thetaStar = thetaStar;
63         this.sigma = sigma; this.rho = rho; this.v = v; this.S = S;
64     }
65
66     public OptionsMC(double r, double K, double[] paramss, double S)
67     {
68         this.r = r;
69         this.K = K; kappaStar = paramss[kappaIndex]; thetaStar = paramss[
70             thetaIndex];
71         sigma = paramss[sigmaIndex]; rho = paramss[rhoIndex]; v = paramss[
72             vIndex]; this.S = S;
73         if (2 * kappaStar * thetaStar <= sigma * sigma)
74         {
75             throw new System.ArgumentException("Feller condition violated."
76                 );
77         }
78         if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
79         {
80             throw new System.ArgumentException("r, K, sigma, S must be
81                 positive");
82         }
83     }
84
85     /// <summary>
86     /// Prices a European call option within the Heston model using Monte
87     /// Carlo methods.
88     /// </summary>
89     /// <param name = "T">The maturity date of the option in years.</param
90     >
91     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
92     our path generator to take to reach time T.</param>
93     /// <param name = "numberPaths">The number of simulations we wish to
94     run.</param>
95     /// <returns>Option price.</returns>
96     public double EuropeanCallOptionPriceMC(double T, int
97         numberTimeStepsPerPath, int numberPaths)
98     {
99         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
100         {
101             throw new System.ArgumentException("Parameters must be positive
102                 ");
103         }

```

```

90         }
91
92         // set up counter and path generator
93         double count = 0;
94         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
95
96         // generate paths, evaluate payoff on each, then average and
97         // discount
98         for (int i = 0; i < numberPaths; i++)
99         {
100             count += Math.Max(path.PathGenerator(T, S,
101                                     numberTimeStepsPerPath) - K, 0);
102         }
103         return Math.Exp(-r * T) * count / numberPaths;
104     }
105
106     /// <summary>
107     /// Prices a European put option within the Heston model using Monte
108     /// Carlo methods.
109     /// </summary>
110     /// <param name = "T">The maturity date of the option in years.</param>
111     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
112     /// our path generator to take to reach time T.</param>
113     /// <param name = "numberPaths">The number of simulations we wish to
114     /// run.</param>
115     /// <returns>Option price.</returns>
116     public double EuropeanPutOptionPriceMC(double T, int
117         numberTimeStepsPerPath, int numberPaths)
118     {
119         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
120         {
121             throw new System.ArgumentException("Parameters must be positive
122                 ");
123         }
124
125         // set up counter and path generator
126         double count = 0;
127         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
128
129         // generate paths, evaluate payoff on each, then average and
130         // discount
131         for (int i = 0; i < numberPaths; i++)
132         {
133             count += Math.Max(K - path.PathGenerator(T, S,
134                                     numberTimeStepsPerPath), 0);
135         }
136         return Math.Exp(-r * T) * count / numberPaths;
137     }
138
139     /// <summary>
140     /// Prices a European call option within the Heston model using Monte
141     /// Carlo methods with parallelisation.
142     /// </summary>
143     /// <param name = "T">The maturity date of the option in years.</param>
144     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
145     /// our path generator to take to reach time T.</param>
146     /// <param name = "numberPaths">The number of simulations we wish to
147     /// run.</param>
148     /// <returns>Option price.</returns>
149     public double EuropeanCallOptionPriceMCParallel(double T, int
150         numberTimeStepsPerPath, int numberPaths)
151     {
152         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
153         {
154             throw new System.ArgumentException("Parameters must be positive

```

```

142         ");
143     }
144     // set up counter and path generator
145     double count = 0;
146     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
147
148     // generate paths in parallel, evaluate payoff on each
149     Parallel.For(0, numberPaths, (i) =>
150     {
151         double pathAdd = Math.Max(path.PathGenerator(T, S,
152             numberTimeStepsPerPath) - K, 0);
153         // update count in such a way as to protect thread safety
154         Interlocked.Exchange(ref count, count + pathAdd);
155     });
156     // average and discount
157     return Math.Exp(-r * T) * count / numberPaths;
158 }
159
160 /// <summary>
161 /// Prices a European call option within the Heston model using Monte
162 /// Carlo methods using anithetic sampling .
163 /// </summary>
164 /// <param name = "T">The maturity date of the option in years.</param>
165 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
166 /// our path generator to take to reach time T.</param>
167 /// <param name = "numberPaths">The number of simulations we wish to
168 /// run.</param>
169 /// <returns>Option price.</returns>
170 public double EuropeanCallOptionPriceMCAnithetic(double T, int
171     numberTimeStepsPerPath, int numberPaths)
172 {
173     if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
174     {
175         throw new System.ArgumentException("Parameters must be positive
176             ");
177     }
178     // set up counter and path generator
179     double count = 0;
180     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
181
182     // generate two paths half as many times, evaluate payoff on each
183     int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
184     for (int i = 0; i < halfNumPaths; i++)
185     {
186         count += Math.Max(path.PathGeneratorAnithetic(T, S,
187             numberTimeStepsPerPath)[0] - K, 0) + Math.Max(path.
188             PathGeneratorAnithetic(T, S, numberTimeStepsPerPath)[1] - K
189             , 0);
190     }
191     // average and discount
192     return Math.Exp(-r * T) * count / (2 * halfNumPaths);
193 }
194
195 /// <summary>
196 /// Prices a European call option within the Heston model using Monte
197 /// Carlo methods using both parallelisation and anithetic sampling .
198 /// </summary>
199 /// <param name = "T">The maturity date of the option in years.</param>
200 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
201 /// our path generator to take to reach time T.</param>
202 /// <param name = "numberPaths">The number of simulations we wish to
203 /// run.</param>

```

```

195     /// <returns>Option price.</returns>
196     public double EuropeanCallOptionPriceMCAnitheticParallel(double T, int
        numberTimeStepsPerPath, int numberPaths)
197     {
198         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
199         {
200             throw new System.ArgumentException("Parameters must be positive
                ");
201         }
202
203         // set up counter and path generator
204         double count = 0;
205         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
206
207         // generate two paths half as many times, in parallel, evaluate
            payoff on each
208         int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
209         Parallel.For(0, halfNumPaths, (i) =>
210         {
211             double[] paths = path.PathGeneratorAnithetic(T, S,
                numberTimeStepsPerPath);
212             double pathAdd = Math.Max(paths[0] - K, 0) + Math.Max(paths[1]
                - K, 0);
213             Interlocked.Exchange(ref count, count + pathAdd);
214         });
215
216         // average and discount
217         return Math.Exp(-r * T) * count / (2.0 * halfNumPaths);
218     }
219
220     /// <summary>
221     /// Prices a European put option within the Heston model using Monte
        Carlo methods using both parallelisation and anithetic sampling .
222     /// </summary>
223     /// <param name = "T">The maturity date of the option in years.</param>
224     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
        our path generator to take to reach time T.</param>
225     /// <param name = "numberPaths">The number of simulations we wish to
        run.</param>
226     /// <returns>Option price.</returns>
227     public double EuropeanPutOptionPriceMCAnitheticParallel(double T, int
        numberTimeStepsPerPath, int numberPaths)
228     {
229         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
230         {
231             throw new System.ArgumentException("Parameters must be positive
                ");
232         }
233
234         // set up counter and path generator
235         double count = 0;
236         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
237
238         // generate two paths half as many times, in parallel, evaluate
            payoff on each
239         int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
240         Parallel.For(0, halfNumPaths, (i) =>
241         {
242             double[] paths = path.PathGeneratorAnithetic(T, S,
                numberTimeStepsPerPath);
243             double pathAdd = Math.Max(K - paths[0], 0) + Math.Max(K - paths
                [1], 0);
244             Interlocked.Exchange(ref count, count + pathAdd);
245         });
246
247         // average and discount
248         return Math.Exp(-r * T) * count / (2.0 * halfNumPaths);

```

```

249     }
250
251
252     /// <summary>
253     /// Checks that the times used for pricing Asian options make sense.
254     /// </summary>
255     /// <param name = "T">An array containing the onservaion times of the
256     /// Asian option.</param>
257     /// <param name = "exerciseT">The Asian option's exercise time.</param>
258     private void CheckAsianOptionInputs(double[] T, double exerciseT)
259     {
260         if (T.Length == 0)
261             throw new System.ArgumentException("Need at least one
262             monitoring date for Asian option.");
263
264         if (T[0] <= 0)
265             throw new System.ArgumentException("The first monitoring date
266             must be positive.");
267
268         for (int i = 1; i < T.Length; ++i)
269         {
270             if (T[i - 1] >= T[i])
271                 throw new System.ArgumentException("Monitoring dates must
272                 be increasing");
273         }
274
275         if (T[T.Length - 1] > exerciseT)
276             throw new System.ArgumentException("Last monitoring time must
277             not be greater than the exercise time");
278     }
279
280     /// <summary>
281     /// Prices an Asian call option within the Heston model using Monte
282     /// Carlo methods.
283     /// </summary>
284     /// <param name = "T">An array containing the onservaion times of the
285     /// Asian option.</param>
286     /// <param name = "exerciseT">The Asian option's exercise time.</param>
287     >
288     /// <param name = "numberPaths">The number of simulations we wish to
289     /// run.</param>
290     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
291     /// our path generator to take to reach time exerciseT.</param>
292     /// <returns>Option price.</returns>
293     public double PriceAsianCallMC(double[] T, double exerciseT, int
294     numberPaths, int numberTimeStepsPerPath)
295     {
296         if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
297         {
298             throw new System.ArgumentException("Monte Carlo settings must
299             be positive");
300         }
301
302         // check parameters make sense
303         CheckAsianOptionInputs(T, exerciseT);
304
305         // set up parameter, path generator and counter
306         int M = T.Length;
307         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
308         double pathCounter = 0;
309
310         // generate paths between observation times, evaluate payoff
311         function
312         for (int i = 0; i < numberPaths; i++)
313         {
314             double priceCount = 0;
315             double holder = S;

```



```

303         double deltaT = T[0];
304
305         for (int j = 0; j < M; j++)
306         {
307             if (j > 0)
308                 deltaT = T[j] - T[j - 1];
309
310             // generate path from previous time point to next with
311             // appropriate number time steps
312             int stepNumber = (int)Math.Ceiling(deltaT *
313                 numberTimeStepsPerPath / exerciseT);
314             holder = path.PathGenerator(deltaT, holder, stepNumber);
315             priceCount += holder;
316         }
317         double pathPayoff = Math.Max(priceCount / M - K, 0);
318         pathCounter += pathPayoff;
319
320     // average and discount
321     return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
322 }
323
324 /// <summary>
325 /// Prices an Asian put option within the Heston model using Monte
326 /// Carlo methods.
327 /// </summary>
328 /// <param name = "T">An array containing the onservation times of the
329 /// Asian option.</param>
330 /// <param name = "exerciseT">The Asian option's exercise time.</param>
331 >
332 /// <param name = "numberPaths">The number of simulations we wish to
333 run.</param>
334 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
335 our path generator to take to reach time exerciseT.</param>
336 /// <returns>Option price.</returns>
337 public double PriceAsianPutMC(double[] T, double exerciseT, int
338     numberPaths, int numberTimeStepsPerPath)
339 {
340     if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
341     {
342         throw new System.ArgumentException("Monte Carlo settings must
343             be positive");
344     }
345
346     // check parameters make sense
347     CheckAsianOptionInputs(T, exerciseT);
348
349     // set up parameter, path generator and counter
350     int M = T.Length;
351     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
352     double pathCounter = 0;
353
354     // generate paths between observation times, evaluate payoff
355     function
356     for (int i = 0; i < numberPaths; i++)
357     {
358         double priceCount = 0;
359         double holder = S;
360         double deltaT = T[0];
361
362         for (int j = 0; j < M; j++)
363         {
364             if (j > 0)
365                 deltaT = T[j] - T[j - 1];
366
367             // generate path from previous time point to next with
368             // appropriate number time steps

```

```

359         int stepNumber = (int)Math.Ceiling(deltaT *
360             numberTimeStepsPerPath / exerciseT);
361         holder = path.PathGenerator(deltaT, holder, stepNumber);
362         priceCount += holder;
363     }
364     double pathPayoff = Math.Max(K - priceCount / M, 0);
365     pathCounter += pathPayoff;
366 }
367 // average and discount
368 return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
369 }
370
371 /// <summary>
372 /// Prices an Asian call option within the Heston model using Monte
373 /// Carlo methods with parallelisation.
374 /// </summary>
375 /// <param name = "T">An array containing the onservation times of the
376 /// Asian option.</param>
377 /// <param name = "exerciseT">The Asian option's exercise time.</param>
378 >
379 /// <param name = "numberPaths">The number of simulations we wish to
380 run.</param>
381 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
382 our path generator to take to reach time exerciseT.</param>
383 /// <returns>Option price.</returns>
384 public double PriceAsianCallMCPParallel(double[] T, double exerciseT,
385     int numberPaths, int numberTimeStepsPerPath)
386 {
387     if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
388     {
389         throw new System.ArgumentException("Monte Carlo settings must
390             be positive");
391     }
392
393     // check parameters make sense
394     CheckAsianOptionInputs(T, exerciseT);
395
396     // set up parameter, path generator and counter
397     int M = T.Length;
398     double pathCounter = 0;
399     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
400
401     // generate paths between observation times in parallel, evaluate
402     // payoff function
403     Parallel.For(0, numberPaths, (i) =>
404     {
405         double priceCount = 0;
406         double holder = S;
407         double deltaT = T[0];
408         Parallel.For(0, M, (j) =>
409         {
410             if (j > 0)
411                 deltaT = T[j] - T[j - 1];
412
413             // generate path from previous time point to next with
414             // appropriate number time steps
415             int stepNumber = (int)Math.Ceiling(deltaT *
416                 numberTimeStepsPerPath / exerciseT);
417             holder = path.PathGenerator(deltaT, holder, stepNumber);
418             Interlocked.Exchange(ref priceCount, priceCount + holder);
419         });
420         double pathPayoff = Math.Max((priceCount / M) - K, 0);
421         Interlocked.Exchange(ref pathCounter, pathCounter + pathPayoff);
422     });
423 }

```

```

414         // average and discount
415         return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
416     }
417
418     /// <summary>
419     /// Prices a Lookback option within the Heston model using Monte Carlo
420     /// methods.
421     /// </summary>
422     /// <param name = "exerciseT">The Lookback option's exercise time.</
423     /// param>
424     /// <param name = "numberPaths">The number of simulations we wish to
425     /// run.</param>
426     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
427     /// our path generator to take to reach time exerciseT.</param>
428     /// <returns>Option price.</returns>
429     public double PriceLookbackCallMC(double exerciseT, int numberPaths,
430         int numberTimeStepsPerPath)
431     {
432         if(exerciseT <= 0 || numberPaths <=0 || numberTimeStepsPerPath <=
433             0)
434         {
435             throw new System.ArgumentException("Parameters must be positive
436                 ");
437         }
438
439         // set up counter, parameter and path generator
440         double pathCounter = 0;
441         double deltaT = exerciseT / numberTimeStepsPerPath;
442         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
443
444         // generate paths of length numberTimeStepsPerPaths, evaluate
445         // payoff on each
446         for (double i = 0; i < numberPaths; i++)
447         {
448             double min = S;
449             double holder = S;
450             for (double j = 0; j <= exerciseT; j += deltaT)
451             {
452                 // take a step with appropriate time change and start point
453                 holder = path.PathGenerator(deltaT, holder, 1);
454
455                 // keep track of minimum
456                 if (holder < min)
457                     min = holder;
458             }
459
460             pathCounter += holder - min;
461         }
462
463         // average and discount
464         return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
465     }
466 }

```

MCPaths.cs class

```

1 using System;
2 using MathNet.Numerics.Distributions;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace HestonCalibrationAndPricing
9 {

```

```

10     /// <summary>
11     /// This class is used to create the Monte Carlo paths which will be used
        to price options within the Heston
12     /// model in the class OptionsMC
13     /// </summary>
14     public class MCPaths
15     {
16         private double r;
17         private double rho;
18         private double kappaStar;
19         private double thetaStar;
20         private double sigma;
21         private double v;
22
23         public MCPaths(double r, double kappaStar, double thetaStar, double
            sigma, double rho, double v)
24         {
25             if (r <= 0 || sigma <= 0 || v <= 0)
26             {
27                 throw new System.ArgumentException("r, sigma, v must be
                    positive");
28             }
29
30             this.r = r; this.rho = rho; this.kappaStar = kappaStar;
31             this.thetaStar = thetaStar; this.sigma = sigma; this.v = v;
32         }
33
34         /// <summary>
35         /// Returns a simulated future price for a risky asset within the
            Heston model.
36         /// </summary>
37         /// <param name = "T">The future time at which we wish to simulate the
            asset price.</param>
38         /// <param name = "S">The initial asset price.</param>
39         /// <param name = "numberTimeStepsPerPath">The number of steps we wish
            the scheme to take to reach time T.</param>
40         /// <returns>Simulated asset price.</returns>
41         public double PathGenerator(double T, double S, int
            numberTimeStepsPerPath)
42         {
43             if (T <= 0 || S <= 0 || numberTimeStepsPerPath <= 0)
44             {
45                 throw new System.ArgumentException("Parameters must be positive
                    ");
46             }
47
48             //sample from normal dist
49             double[] x1 = new double[numberTimeStepsPerPath];
50             Normal.Samples(x1, 0, 1);
51             double[] x2 = new double[numberTimeStepsPerPath];
52             Normal.Samples(x2, 0, 1);
53
54             //set up parameters
55             double tau = T / numberTimeStepsPerPath;
56             double sqrtTau = Math.Sqrt(tau);
57             double sqrtOneMinusRhoSquared = Math.Sqrt(1 - rho * rho);
58             double alpha = (4 * kappaStar * thetaStar - sigma * sigma) / 8.0;
59             double beta = -kappaStar / 2.0;
60             double gamma = sigma / 2.0;
61
62             // set up holder variables for Monte Carlo
63             double y = Math.Sqrt(v);
64             double s = S;
65
66             // update holder variables iteratively according to MC scheme.
67             for (int i = 0; i < numberTimeStepsPerPath; i++)
68             {

```

```

69         double deltaZ1 = sqrtTau * x1[i];
70         double deltaZ2 = sqrtTau * (rho * x1[i] +
            sqrtOneMinusRhoSquared * x2[i]);
71         s = s + r * s * tau + y * s * deltaZ1;
72         double a = (y + gamma * deltaZ2) / (2 * (1 - beta * tau));
73         y = a + Math.Sqrt(a * a + alpha * tau / (1 - beta * tau));
74     }
75
76     return s;
77 }
78
79 /// <summary>
80 /// Returns a simulated future price for a risky asset within the
    Heston model using anithetic sampling with a view to reducing
    variance.
81 /// </summary>
82 /// <param name = "T">The future time at which we wish to simulate the
    asset price.</param>
83 /// <param name = "S">The initial asset price.</param>
84 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
    the scheme to take to reach time T.</param>
85 /// <returns>Simulated asset price.</returns>
86 public double[] PathGeneratorAnithetic(double T, double S, int
    numberTimeStepsPerPath)
87 {
88     if (T <= 0 || S <= 0 || numberTimeStepsPerPath <= 0)
89     {
90         throw new System.ArgumentException("Parameters must be positive
            ");
91     }
92
93     // sample from normal dist
94     double[] x1 = new double[numberTimeStepsPerPath];
95     double[] x2 = new double[numberTimeStepsPerPath];
96     Normal.Samples(x1, 0, 1);
97     Normal.Samples(x2, 0, 1);
98
99     //set up parameters
100     double tau = T / numberTimeStepsPerPath;
101     double sqrtTau = Math.Sqrt(tau);
102     double sqrtOneMinusRhoSquared = Math.Sqrt(1 - rho * rho);
103     double alpha = (4 * kappaStar * thetaStar - sigma * sigma) / 8.0;
104     double beta = -kappaStar / 2.0;
105     double gamma = sigma / 2.0;
106
107     // set up holder variables, now two sets due to anithetic sampling
108     double y = Math.Sqrt(v);
109     double y1 = Math.Sqrt(v);
110     double s = S;
111     double s1 = S;
112
113     // update both sets of holder variables iteratively, using the
        negative of normal sample for anitheic path
114     for (int i = 0; i < numberTimeStepsPerPath; i++)
115     {
116         double deltaZ1 = sqrtTau * x1[i];
117         double deltaZ2 = sqrtTau * (rho * x1[i] +
            sqrtOneMinusRhoSquared * x2[i]);
118         s = s + r * s * tau + y * s * deltaZ1;
119         s1 = s1 + r * s1 * tau - y1 * s1 * deltaZ1;
120         double a = (y + gamma * deltaZ2) / (2 * (1 - beta * tau));
121         double aa = (y1 - gamma * deltaZ2) / (2 * (1 - beta * tau));
122         y = a + Math.Sqrt(a * a + alpha * tau / (1 - beta * tau));
123         y1 = aa + Math.Sqrt(aa * aa + alpha * tau / (1 - beta * tau));
124     }
125
126     // return array of end points of the two paths.

```

```

127         double[] sArray = { s, s1 };
128         return sArray;
129     }
130
131 }
132 }

```

Calibrator.cs class

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7
8 namespace HestonCalibrationAndPricing
9 {
10
11     public class CalibrationFailedException : Exception
12     {
13         public CalibrationFailedException()
14         {
15         }
16         public CalibrationFailedException(string message)
17             : base(message)
18         {
19         }
20     }
21
22     public struct MarketData
23     {
24         public double K;
25         public double T;
26         public double Price;
27     }
28
29     public enum CalibrationOutcome
30     {
31         NotStarted,
32         FinishedOK,
33         FailedMaxItReached,
34         FailedOtherReason
35     };
36
37     /// <summary>
38     /// This class is used to calibrate the parameters of the Heston Model to
39     /// real world data
40     /// </summary>
41     public class Calibrator
42     {
43         private const double defaultAccuracy = 1.0e-15;
44         private const int defaultMaxIts = 1000;
45         private double accuracy;
46         private int maxIts;
47
48         private List<MarketData> marketList;
49         private double r;
50         private double S;
51
52         private CalibrationOutcome outcome;
53
54         private double[] calibratedParams;
55
56         public Calibrator()
57         {
58             accuracy = defaultAccuracy;
59

```

```

58         maxIts = defaultMaxIts;
59         marketList = new List<MarketData>();
60         r = 0;
61         S = 0;
62     }
63
64     public Calibrator(double r, double S, int maxIts, double accuracy)
65     {
66         if (r <= 0 || S <= 0 || maxIts <= 0 )
67         {
68             throw new System.ArgumentException("Parameters must be positive
69             ");
70         }
71
72         this.accuracy = accuracy;
73         this.maxIts = maxIts;
74         marketList = new List<MarketData>();
75         this.r = r;
76         this.S = S;
77     }
78
79     /// <summary>
80     /// Sets the parameters which will be used as a starting point by the
81     /// calibrator
82     /// </summary>
83     public void SetGuessParameters(double kappaStar, double thetaStar,
84     double sigma, double rho, double v)
85     {
86         if (sigma <= 0 || v <= 0)
87         {
88             throw new System.ArgumentException("Sigma, v must be positive")
89             ;
90         }
91
92         Options e = new Options(r, S, kappaStar, thetaStar, sigma, rho, v);
93         calibratedParams = e.ParamsAsArray();
94     }
95
96     /// <summary>
97     /// Adds the details of a real world option to the list marketList of
98     /// data which will be used for calibration.
99     /// </summary>
100     /// <param name="K">Observed option's strike price.</param>
101     /// <param name="T">Observed option's maturity time.</param>
102     /// <param name="Price">Observed options price.</param>
103     public void AddObservedOption(double K, double T, double Price)
104     {
105         if (K <= 0 || T <= 0 || Price <= 0)
106         {
107             throw new System.ArgumentException("Parameters must be positive
108             ");
109         }
110
111         MarketData observedOption;
112         observedOption.K = K;
113         observedOption.T = T;
114         observedOption.Price = Price;
115         marketList.Add(observedOption);
116     }
117
118     /// <summary>
119     /// Calculates the mean squared error between the European call prices
120     /// of an instance, options,
121     /// of the class Options and the market prices found in marketList
122     /// </summary>
123     /// <param name="options">An instance of the class Options.</param>
124     public double CalculateMeanSquaredErrorBetweenModelAndMarket(Options

```

```

118         options)
119     {
120         double mse = 0;
121         foreach (MarketData data in marketList)
122         {
123             double T = data.T;
124             double K = data.K;
125             double price = options.EuropeanCallPrice(T, K);
126             double diff = price - data.Price;
127             mse += diff * diff;
128         }
129         return mse;
130     }
131
132     /// <summary>
133     /// This is the function which will be used by the calibrator
134     /// </summary>
135     public void CalibrationObjectiveFunction(double[] paramsarray, ref
136         double func, object obj)
137     {
138         Options european = new Options(r, S, paramsarray);
139         func = CalculateMeanSquaredErrorBetweenModelAndMarket(european);
140     }
141
142     /// <summary>
143     /// Calibrates the model parameters to fit the market data as closely
144     /// as possible
145     /// </summary>
146     public void Calibrate()
147     {
148
149         // set up for calibration
150         outcome = CalibrationOutcome.NotStarted;
151         double[] initialParams = new double[Options.numberParams];
152
153         if (calibratedParams == null)
154         {
155             throw new System.Exception("Please add an initial guess for
156                 parameters");
157         }
158
159         calibratedParams.CopyTo(initialParams, 0);
160         double epsg = accuracy;
161         double epsf = accuracy;
162         double epsx = accuracy;
163         double diffstep = 1.0e-6;
164         int maxits = maxIts;
165         double stpmax = 0.05;
166
167         alglib.minlbfgsstate state;
168         alglib.minlbfgsreport rep;
169         alglib.minlbfgscreatef(5, initialParams, diffstep, out state);
170         alglib.minlbfgssetcond(state, epsg, epsf, epsx, maxits);
171         alglib.minlbfgssetstpmax(state, stpmax);
172
173         // calibrate and return outcome, error
174         alglib.minlbfgsoptimize(state, CalibrationObjectiveFunction, null,
175             null);
176         double[] resultParams = new double[Options.numberParams];
177         alglib.minlbfgsresults(state, out resultParams, out rep);
178
179         System.Console.WriteLine("Termination type: {0}", rep.
180             terminationtype);
181         System.Console.WriteLine("Num iterations {0}", rep.iterationscount)
182             ;
183         System.Console.WriteLine("{0}", alglib.ap.format(resultParams, 5));
184     }

```



```

178         if (rep.terminationtype == 1
179             || rep.terminationtype == 2
180             || rep.terminationtype == 4)
181         {
182             outcome = CalibrationOutcome.FinishedOK;
183             calibratedParams = resultParams;
184         }
185         else if (rep.terminationtype == 5)
186         {
187             outcome = CalibrationOutcome.FailedMaxItReached;
188             calibratedParams = resultParams;
189         }
190         else
191         {
192             outcome = CalibrationOutcome.FailedOtherReason;
193             throw new CalibrationFailedException("Heston model calibration
194                 failed badly.");
195         }
196     }
197
198     /// <summary>
199     /// Obtains the calibration status of the model, as well as the models
200     /// pricing error.
201     /// </summary>
202     public void GetCalibrationStatus(ref CalibrationOutcome calibOutcome,
203                                     ref double pricingError)
204     {
205         calibOutcome = outcome;
206         Options m = new Options(r, S, calibratedParams);
207         pricingError = CalculateMeanSquaredErrorBetweenModelAndMarket(m);
208     }
209
210     /// <summary>
211     /// Creates an instance of the class Options with the calibrated
212     /// parameters.
213     /// </summary>
214     /// <returns>Calibrated model.</returns>
215     public Options GetCalibratedModel()
216     {
217         Options m = new Options(r, S, calibratedParams);
218         return m;
219     }
220 }

```

InterfaceFill.cs class

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using HestonModel.Interfaces;
7
8 namespace HestonModel.InterfaceImplement
9 {
10     /// <summary>
11     /// This class is used to implement several interfaces.
12     /// </summary>
13     public class InterfaceFill : IOption, IVarianceProcessParameters,
14                                 IMonteCarloSettings, IAsianOption, IEuropeanOption,
15                                 ICalibrationSettings
16     {
17         double T;
18         double kappa;
19     }
20 }

```

```

17     double theta;
18     double sigma;
19     double v;
20     double rho;
21     int numberTrials;
22     int numberTimeSteps;
23     PayoffType p;
24     IEnumerable<double> timeList;
25     double K;
26     double accuracy;
27
28
29
30     public InterfaceFill(double T, double kappa, double theta, double sigma
        , double rho, double v, int numberTrials, int numberTimeSteps,
        PayoffType p, double[] timeArray, double K, double accuracy)
31     {
32         this.T = T; this.kappa = kappa; this.theta = theta; this.sigma =
            sigma;
33         this.v = v; this.rho = rho; this.numberTrials = numberTrials; this.
            numberTimeSteps = numberTimeSteps;
34         this.p = p; timeList = timeArray.AsEnumerable(); this.K = K;
35         this.accuracy = accuracy;
36     }
37
38
39
40     double IOption.Maturity => T;
41
42     double IVarianceProcessParameters.Kappa => kappa;
43
44     double IVarianceProcessParameters.Theta => theta;
45
46     double IVarianceProcessParameters.Sigma => sigma;
47
48     double IVarianceProcessParameters.V0 => v;
49
50     double IVarianceProcessParameters.Rho => rho;
51
52     int IMonteCarloSettings.NumberOfTrials => numberTrials;
53
54     int IMonteCarloSettings.NumberOfTimeSteps => numberTimeSteps;
55
56     IEnumerable<double> IAsianOption.MonitoringTimes => timeList;
57
58     PayoffType IEuropeanOption.Type => p;
59
60     double IEuropeanOption.StrikePrice => K;
61
62     double ICalibrationSettings.Accuracy => accuracy;
63
64     int ICalibrationSettings.MaximumNumberOfIterations => 1000;
65 }
66 }

```

OtherInterfaceFill.cs class

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using HestonModel.Interfaces;
7
8 namespace HestonModel.InterfaceImplement
9 {
10     /// <summary>

```

```

11     /// This class is used to implement the interfaces IHestonModelParameters
    and ICalibrationResult
12     /// </summary>
13     public class OtherInterfaceFill : IHestonModelParameters,
        ICalibrationResult
14     {
15         double S;
16         double r;
17         IVarianceProcessParameters paramss;
18         CalibrationOutcome c;
19         double error;
20
21         public OtherInterfaceFill(double T, double kappa, double theta, double
            sigma, double rho, double v, int numberTrials, int numberTimeSteps,
            double S, double r, CalibrationOutcome c, double error)
22         {
23             this.S = S; this.r = r;
24             double[] TT = { 0 };
25             InterfaceFill fill = new InterfaceFill(T, kappa, theta, sigma, rho,
                v, numberTrials, numberTimeSteps, 0, TT, 100, 0);
26             paramss = fill;
27             this.c = c; this.error = error;
28         }
29
30         double IHestonModelParameters.InitialStockPrice => S;
31
32         double IHestonModelParameters.RiskFreeRate => r;
33
34         IVarianceProcessParameters IHestonModelParameters.VarianceParameters =>
            paramss;
35
36         CalibrationOutcome ICalibrationResult.MinimizerStatus => c;
37
38         double ICalibrationResult.PricingError => error;
39     }
40 }
41 }

```

AnotherInterfaceFill.cs class

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using HestonModel.Interfaces;
7
8 namespace HestonModel.InterfaceImplement
9 {
10     /// <summary>
11     /// This class is used to implement the interface IHestonCalibrationResult.
12     /// </summary>
13     public class AnotherInterfaceFill : IHestonCalibrationResult
14     {
15         IHestonModelParameters paramms;
16         double error;
17         CalibrationOutcome c;
18
19         public AnotherInterfaceFill(double T, double kappa, double theta,
            double sigma, double rho, double v, int numberTrials, int
            numberTimeSteps, double S, double r, CalibrationOutcome c, double
            error)
20         {
21             this.c = c; this.error = error;
22             OtherInterfaceFill fill = new OtherInterfaceFill(T, kappa, theta,
                sigma, rho, v, numberTrials, numberTimeSteps, S, r, c, error);
23             paramms = fill;

```

```

24     }
25
26     IHestonModelParameters IHestonCalibrationResult.Parameters => paramms;
27
28     CalibrationOutcome ICalibrationResult.MinimizerStatus => c;
29
30     double ICalibrationResult.PricingError => error;
31 }
32 }

```

FinalInterfaceFill.cs class

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using HestonModel.Interfaces;
7
8 namespace HestonModel.InterfaceImplement
9 {
10     /// <summary>
11     /// This class is used to implement the interface IOptionMarketData when it
12     /// takes IEuropeanOption.
13     /// </summary>
14     public class FinalInterfaceFill : IOptionMarketData<IEuropeanOption>
15     {
16         double price;
17         IEuropeanOption option;
18
19         public FinalInterfaceFill(double price, double T, PayoffType p, double
20             K)
21         {
22             this.price = price;
23             double[] TT = { 0 };
24             InterfaceFill fill = new InterfaceFill(T, 0, 0, 0, 0, 0, 0, 0, p,
25                 TT, K, 0);
26             option = fill;
27         }
28
29         IEuropeanOption IOptionMarketData<IEuropeanOption>.Option => option;
30
31         double IOptionMarketData<IEuropeanOption>.Price => price;
32     }
33 }

```

Heston.cs

```

1 using System;
2 using System.Collections.Generic;
3 using HestonModel.Interfaces;
4 using HestonCalibrationAndPricing;
5 using System.Linq;
6 using HestonModel.InterfaceImplement;
7
8 namespace HestonModel
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'
13     /// t change the namespace.
14     /// Your code should be implemented in other classes (or even projects if
15     /// you wish), and the relevant functionality should only be called here
16     /// and outputs returned.

```

```

15     /// You don't need to implement the interfaces that have been provided if
16     you don't want to.
17     /// </summary>
18     public static class Heston
19     {
20         /// <summary>
21         /// Method for calibrating the heston model.
22         /// </summary>
23         /// <param name="guessModelParameters">Object implementing
24         IHestonModelParameters interface containing the risk-free rate,
25         initial stock price
26         and initial guess parameters to be used in the calibration.</param>
27         /// <param name="referenceData">A collection of objects implementing
28         IOptionMarketData<IEuropeanOption> interface. These should contain
29         the reference data used for calibration.</param>
30         /// <param name="calibrationSettings">An object implementing
31         ICalibrationSettings interface.</param>
32         /// <returns>Object implementing IHestonCalibrationResult interface
33         which contains calibrated model parameters and additional
34         diagnostic information</returns>
35     public static IHestonCalibrationResult CalibrateHestonParameters(
36         IHestonModelParameters guessModelParameters, IEnumerable<
37         IOptionMarketData<IEuropeanOption>> referenceData,
38         ICalibrationSettings calibrationSettings)
39     {
40         // set up calibrator
41         Calibrator cal = new Calibrator(guessModelParameters.RiskFreeRate,
42             guessModelParameters.InitialStockPrice, calibrationSettings.
43             MaximumNumberOfIterations, calibrationSettings.Accuracy);
44         cal.SetGuessParameters(guessModelParameters.VarianceParameters.
45             Kappa, guessModelParameters.VarianceParameters.Theta,
46             guessModelParameters.VarianceParameters.Sigma,
47             guessModelParameters.VarianceParameters.Rho,
48             guessModelParameters.VarianceParameters.V0);
49
50         // add market data
51         foreach (IOptionMarketData<IEuropeanOption> data in referenceData)
52         {
53             cal.AddObservedOption(data.Option.StrikePrice, data.Option.
54                 Maturity, data.Price);
55         }
56
57         // calibrate, get error, outcome, parameters
58         cal.Calibrate();
59         double error = 0;
60         HestonCalibrationAndPricing.CalibrationOutcome outcome =
61             HestonCalibrationAndPricing.CalibrationOutcome.NotStarted;
62         cal.GetCalibrationStatus(ref outcome, ref error);
63
64         Options e = cal.GetCalibratedModel();
65         double[] paramArray = e.ParamsAsArray();
66         CalibrationOutcome outcome1 = (CalibrationOutcome)outcome;
67
68         // implement and return IHestonCalibrationResult
69         AnotherInterfaceFill fill = new AnotherInterfaceFill(0, paramArray[
70             Options.kappaIndex], paramArray[Options.thetaIndex], paramArray[
71             Options.sigmaIndex], paramArray[Options.rhoIndex], paramArray[
72             Options.vIndex], 0, 0, 0, 0, outcome1, error);
73         return fill;
74     }
75
76     /// <summary>
77     /// Price a European option in the Heston model using the Heston
78     formula. This should be accurate to 5 decimal places
79     /// </summary>
80     /// <param name="parameters">Object implementing IHestonModelParameters
81     interface, containing model parameters.</param>

```

```

59     /// <param name="europeanOption">Object implementing IEuropeanOption
    interface, containing the option parameters.</param>
60     /// <returns>Option price</returns>
61     public static double HestonEuropeanOptionPrice(IHestonModelParameters
    parameters, IEuropeanOption europeanOption)
62     {
63         Options eur = new Options(parameters.RiskFreeRate, parameters.
    InitialStockPrice,
64         parameters.VarianceParameters.Kappa, parameters.
    VarianceParameters.Theta, parameters.VarianceParameters.
    Sigma,
65         parameters.VarianceParameters.Rho, parameters.VarianceParameters
    .V0);
66
67         if (europeanOption.Type == 0)
68         {
69             return eur.EuropeanCallPrice(europeanOption.Maturity,
    europeanOption.StrikePrice);
70         }
71         else
72             return eur.EuropeanPutPrice(europeanOption.Maturity,
    europeanOption.StrikePrice);
73     }
74
75     /// <summary>
76     /// Price a European option in the Heston model using the Monte-Carlo
    method. Accuracy will depend on number of time steps and samples
77     /// </summary>
78     /// <param name="parameters">Object implementing IHestonModelParameters
    interface, containing model parameters.</param>
79     /// <param name="europeanOption">Object implementing IEuropeanOption
    interface, containing the option parameters.</param>
80     /// <param name="monteCarloSimulationSettings">An object implementing
    IMonteCarloSettings object and containing simulation settings.</
    param>
81     /// <returns>Option price</returns>
82     public static double HestonEuropeanOptionPriceMC(IHestonModelParameters
    parameters, IEuropeanOption europeanOption, IMonteCarloSettings
    monteCarloSimulationSettings)
83     {
84         OptionsMC option = new OptionsMC(parameters.RiskFreeRate,
    europeanOption.StrikePrice, parameters.VarianceParameters.Kappa
    ,
85         parameters.VarianceParameters.Theta, parameters.
    VarianceParameters.Sigma, parameters.VarianceParameters.Rho
    ,
86         parameters.VarianceParameters.V0, parameters.InitialStockPrice)
    ;
87
88         if(europeanOption.Type == 0)
89         {
90             return option.EuropeanCallOptionPriceMCAnitheticParallel(
    europeanOption.Maturity, monteCarloSimulationSettings.
    NumberOfTimeSteps, monteCarloSimulationSettings.
    NumberOfTrials);
91         }
92         else
93             return option.EuropeanPutOptionPriceMCAnitheticParallel(
    europeanOption.Maturity, monteCarloSimulationSettings.
    NumberOfTimeSteps, monteCarloSimulationSettings.
    NumberOfTrials);
94     }
95
96     /// <summary>
97     /// Price a Asian option in the Heston model using the
98     /// Monte-Carlo method. Accuracy will depend on number of time steps
    and samples</summary>

```

```

99      /// <param name="parameters">Object implementing IHestonModelParameters
100      /// interface, containing model parameters.</param>
101      /// <param name="asianOption">Object implementing IAsian interface,
102      /// containing the option parameters.</param>
103      /// <param name="monteCarloSimulationSettings">An object implementing
104      /// IMonteCarloSettings object and containing simulation settings.</
105      param>
106      /// <returns>Option price</returns>
107      public static double HestonAsianOptionPriceMC(IHestonModelParameters
108      parameters, IAsianOption asianOption, IMonteCarloSettings
109      monteCarloSimulationSettings)
110      {
111          OptionsMC asian = new OptionsMC(parameters.RiskFreeRate,
112          asianOption.StrikePrice, parameters.VarianceParameters.Kappa,
113          parameters.VarianceParameters.Theta, parameters.
114          VarianceParameters.Sigma, parameters.VarianceParameters.Rho,
115          parameters.VarianceParameters.V0, parameters.InitialStockPrice);
116
117          if (asianOption.Type == 0)
118          {
119              return asian.PriceAsianCallMC(asianOption.MonitoringTimes.
120              ToArray(), asianOption.Maturity,
121              monteCarloSimulationSettings.NumberOfTrials,
122              monteCarloSimulationSettings.NumberOfTimeSteps);
123          }
124          else
125              return asian.PriceAsianPutMC(asianOption.MonitoringTimes.
126              ToArray(), asianOption.Maturity,
127              monteCarloSimulationSettings.NumberOfTrials,
128              monteCarloSimulationSettings.NumberOfTimeSteps);
129      }
130
131      /// <summary>
132      /// Price a lookback option in the Heston model using the
133      /// a Monte-Carlo method. Accuracy will depend on number of time steps
134      /// and samples </summary>
135      /// <param name="parameters">Object implementing IHestonModelParameters
136      /// interface, containing model parameters.</param>
137      /// <param name="maturity">An object implementing IOption interface and
138      /// containing option's maturity</param>
139      /// <param name="monteCarloSimulationSettings">An object implementing
140      /// IMonteCarloSettings object and containing simulation settings.</
141      param>
142      /// <returns>Option price</returns>
143      public static double HestonLookbackOptionPriceMC(IHestonModelParameters
144      parameters, IOption maturity, IMonteCarloSettings
145      monteCarloSimulationSettings)
146      {
147          OptionsMC lookback = new OptionsMC(parameters.RiskFreeRate,
148          parameters.VarianceParameters.Kappa,
149          parameters.VarianceParameters.Theta, parameters.
150          VarianceParameters.Sigma, parameters.
151          VarianceParameters.Rho,
152          parameters.VarianceParameters.V0, parameters.
153          InitialStockPrice);
154
155          return lookback.PriceLookbackCallMC(maturity.Maturity,
156          monteCarloSimulationSettings.NumberOfTrials,
157          monteCarloSimulationSettings.NumberOfTimeSteps);
158      }
159  }
160 }

```