

OOPA

Adam Keogh

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	8
Options.cs class	8
OptionsMC.cs class	10
MCPaths.cs class	19
Calibrator.cs class	21
InterfaceFill.cs class	25
OtherInterfaceFill.cs class	26
AnotherInterfaceFill.cs class	27
FinalInterfaceFill.cs class	27
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 neccessary 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

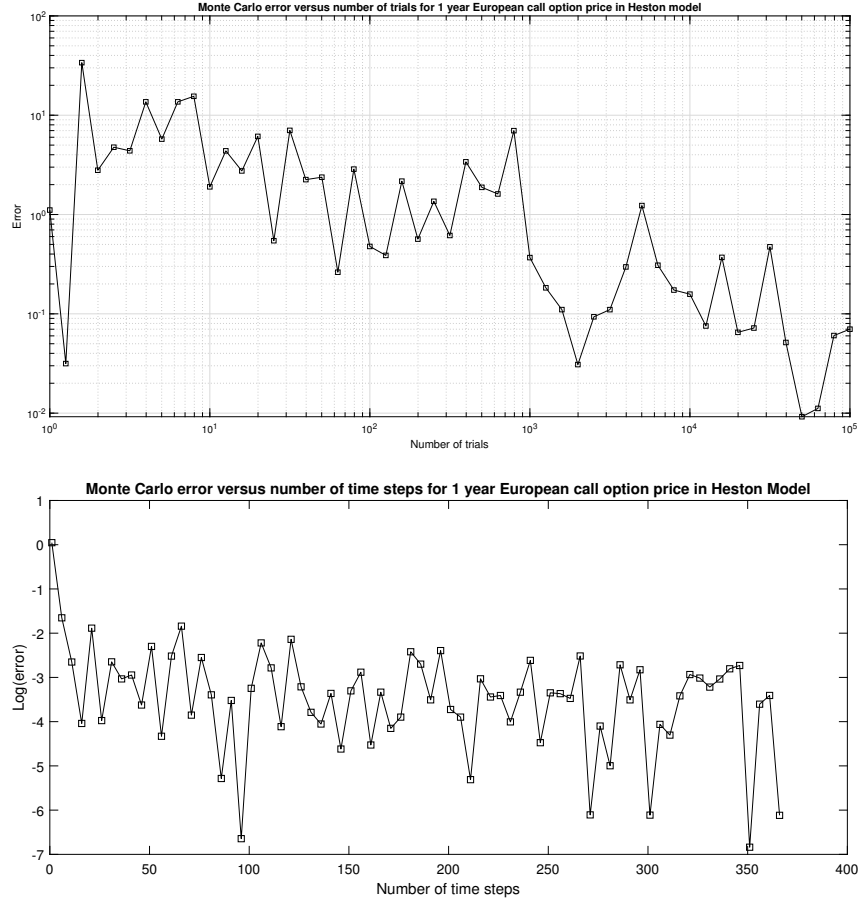
$$\begin{aligned} r &= 10\% \\ \theta^* &= 6\%, \kappa^* = 200\%, \sigma = 40\%, \rho = 50\%, v = 4\% \\ S &= 100. \end{aligned} \tag{2}$$

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.

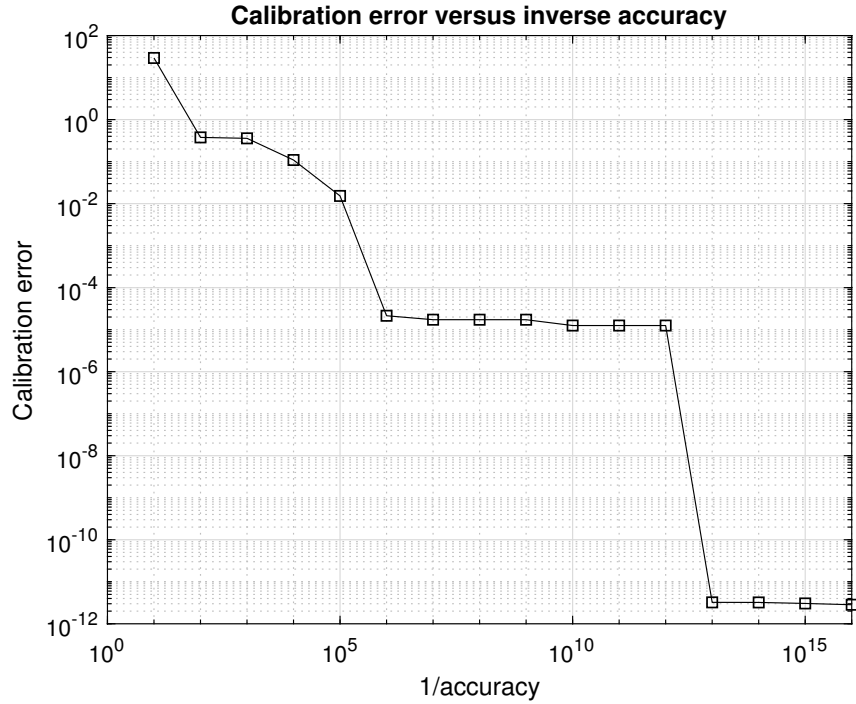
What I found was that with accuracy 1×10^{-3} the calibrated parameter values would vary dependent on the initial guess parameters, while with accuracy 1×10^{-15} they would converge to the below parameters from most reasonable starting points which I tried. The calibration returned FinishedOk in the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

```



```

36         {
37             throw new System.ArgumentException("r, S, sigma, v must be
38                 positive");
39         }
40         this.r = r; this.S = S; this.kappaStar = kappaStar; this.thetaStar
41             = thetaStar;
42         this.sigma = sigma; this.rho = rho; this.v = v;
43     }
44
45     public Options(double r, double S, double[] parameters)
46     {
47         this.r = r; this.S = S;
48         kappaStar = parameters[kappaIndex];
49         thetaStar = parameters[thetaIndex];
50         sigma = parameters[sigmaIndex];
51         rho = parameters[rhoIndex];
52         v = parameters[vIndex];
53
54         if (r <= 0 || S <= 0 || sigma <= 0 || v <= 0)
55         {
56             throw new System.ArgumentException("r, S, sigma, v must be
57                 positive");
58         }
59     }
60
61     /// <summary>
62     /// Prices a European call option within the Heston model.
63     /// </summary>
64     /// <param name = "T">The maturity date of the option in years.</param>
65     /// <param name = "K">The options' strike price.</param>
66     /// <returns>Option price.</returns>
67     public double EuropeanCallPrice(double T, double K)
68     {
69         if (T < 0 || K < 0)
70         {
71             throw new System.ArgumentException("T, K must be non-negative")
72                 ;
73         }
74
75         // set up paramters
76         double[] b = { kappaStar - rho * sigma, kappaStar };
77         double[] u = { 0.5, -0.5 };
78         double a = kappaStar * thetaStar;
79         Complex i = new Complex(0, 1);
80
81         // function implementing part of Heston formula
82         Func<int, double, double> RealP = (j, phi) =>
83         {
84             Complex temp1 = new Complex(-b[j], rho * sigma * phi);
85             Complex temp11 = new Complex(-phi * phi, 2 * u[j] * phi);
86             Complex d = Complex.Pow(temp1 * temp1 - sigma * sigma * temp11,
87                 0.5);
88             Complex g = (b[j] - rho * sigma * phi * i - d) * Complex.Pow(b[
89                 j] - rho * sigma * phi * i + d, -1);
90
91             Complex c = r * phi * T * i + (a / (sigma * sigma)) * ((b[j] -
92                 rho * sigma * phi * i - d) * T - 2 * Complex.Log((1 - g *
93                 Complex.Exp(-T * d)) / (1 - g)));
94             Complex bigD = ((b[j] - rho * sigma * phi * i - d) / (sigma *
95                 sigma)) * ((1 - Complex.Exp(-T * d)) / (1 - g * Complex.Exp
96                 (-T * d)));
97             Complex littlePhi = Complex.Exp(c + bigD * v + phi * i * Math.
98                 Log(S));
99             Complex value = Complex.Exp(-i * phi * Math.Log(K)) * littlePhi

```

```

91         / (i * phi);
92         return value.Real;
93     };
94
95     double[] P = new double[2];
96
97     // integrate with appropriate number of steps and length to
98     // approximate infinite integral
99     P[0] = 0.5 + (1.0 / Math.PI) * SimpsonRule.IntegrateComposite(x =>
100     RealP(0, x), 0.000001, 50, 100);
101     P[1] = 0.5 + (1.0 / Math.PI) * SimpsonRule.IntegrateComposite(x =>
102     RealP(1, x), 0.000001, 50, 100);
103
104     return S * P[0] - K * Math.Exp(-r * T) * P[1];
105 }
106
107 /// <summary>
108 /// Prices a European put option within the Heston model.
109 /// </summary>
110 /// <param name = "T">The maturity date of the option in years.</param>
111 >
112 /// <param name = "K">The options' strike price.</param>
113 /// <returns>Option price.</returns>
114 public double EuropeanPutPrice(double T, double K)
115 {
116     if (T < 0 || K < 0)
117     {
118         throw new System.ArgumentException("T, K must be non-negative");
119     }
120
121     // put call parity
122     return EuropeanCallPrice(T, K) - S + K * Math.Exp(-r * T);
123 }
124
125 /// <summary>
126 /// Forms an array of the model parameters.
127 /// </summary>
128 /// <returns>Model parameters.</returns>
129 public double[] ParamsAsArray()
130 {
131     double[] paramsArray = new double[Options.numberParams];
132     paramsArray[kappaIndex] = kappaStar;
133     paramsArray[thetaIndex] = thetaStar;
134     paramsArray[sigmaIndex] = sigma;
135     paramsArray[rhoIndex] = rho;
136     paramsArray[vIndex] = v;
137     return paramsArray;
138 }
139
140 }
141
142 }

```

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
    methods.
13    /// </summary>
14    public class OptionsMC
15    {
16        public const int kappaIndex = 0;
17        public const int thetaIndex = 1;
18        public const int sigmaIndex = 2;
19        public const int rhoIndex = 3;
20        public const int vIndex = 4;
21
22        private double r;
23        private double K;
24        private double kappaStar;
25        private double thetaStar;
26        private double sigma;
27        private double rho;
28        private double v;
29        private double S;
30
31        public OptionsMC(double r, double K, double kappaStar, double thetaStar
            , double sigma, double rho, double v, double S)
32        {
33            if (2 * kappaStar * thetaStar <= sigma * sigma)
34            {
35                throw new System.ArgumentException("Feller condition violated."
                    );
36            }
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
                    positive");
41            }
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
            sigma, double rho, double v, double S)
49        {
50            if (2 * kappaStar * thetaStar <= sigma * sigma)
51            {
52                throw new System.ArgumentException("Feller condition violated."
                    );
53            }
54            if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
55            {
56                throw new System.ArgumentException("r, K, sigma, S must be
                    positive");
57            }
58            this.r = r;
59            this.kappaStar = kappaStar; this.thetaStar = thetaStar;
60            this.sigma = sigma; this.rho = rho; this.v = v; this.S = S;
61        }
62
63        public OptionsMC(double r, double K, double[] paramss, double S)
64        {
65            this.r = r;
66            this.K = K; kappaStar = paramss[kappaIndex]; thetaStar = paramss[
                thetaIndex];
67            sigma = paramss[sigmaIndex]; rho = paramss[rhoIndex]; v = paramss[
                vIndex]; this.S = S;
68            if (2 * kappaStar * thetaStar <= sigma * sigma)
69            {

```

```

70         throw new System.ArgumentException("Feller condition violated."
71         );
72     }
73     if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
74     {
75         throw new System.ArgumentException("r, K, sigma, S must be
76         positive");
77     }
78 }
79
80 /// <summary>
81 /// Prices a European call option within the Heston model using Monte
82 /// Carlo methods.
83 /// </summary>
84 /// <param name = "T">The maturity date of the option in years.</param
85 >
86 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
87 our path generator to take to reach time T.</param>
88 /// <param name = "numberPaths">The number of simulations we wish to
89 run.</param>
90 /// <returns>Option price.</returns>
91 public double EuropeanCallOptionPriceMC(double T, int
92 numberTimeStepsPerPath, int numberPaths)
93 {
94     if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
95     {
96         throw new System.ArgumentException("Parameters must be positive
97         ");
98     }
99
100     // set up counter and path generator
101     double count = 0;
102     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
103
104     // generate paths, evaluate payoff on each, then average and
105     discount
106     for (int i = 0; i < numberPaths; i++)
107     {
108         count += Math.Max(path.PathGenerator(T, S,
109         numberTimeStepsPerPath) - K, 0);
110     }
111     return Math.Exp(-r * T) * count / numberPaths;
112 }
113
114 /// <summary>
115 /// Prices a European put option within the Heston model using Monte
116 /// Carlo methods.
117 /// </summary>
118 /// <param name = "T">The maturity date of the option in years.</param
119 >
120 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
121 our path generator to take to reach time T.</param>
122 /// <param name = "numberPaths">The number of simulations we wish to
123 run.</param>
124 /// <returns>Option price.</returns>
125 public double EuropeanPutOptionPriceMC(double T, int
126 numberTimeStepsPerPath, int numberPaths)
127 {
128     if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
129     {
130         throw new System.ArgumentException("Parameters must be positive
131         ");
132     }
133
134     // set up counter and path generator
135     double count = 0;
136     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

```

```

121
122         // generate paths, evaluate payoff on each, then average and
            discount
123     for (int i = 0; i < numberPaths; i++)
124     {
125         count += Math.Max(K - path.PathGenerator(T, S,
            numberTimeStepsPerPath), 0);
126     }
127     return Math.Exp(-r * T) * count / numberPaths;
128 }
129
130     /// <summary>
131     /// Prices a European call option within the Heston model using Monte
        Carlo methods with parallelisation.
132     /// </summary>
133     /// <param name = "T">The maturity date of the option in years.</param>
        >
134     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
        our path generator to take to reach time T.</param>
135     /// <param name = "numberPaths">The number of simulations we wish to
        run.</param>
136     /// <returns>Option price.</returns>
137     public double EuropeanCallOptionPriceMCParallel(double T, int
        numberTimeStepsPerPath, int numberPaths)
138     {
139         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
140         {
141             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,
                numberTimeStepsPerPath) - K, 0);
152             // update count in such a way as to protect thread safety
153             Interlocked.Exchange(ref count, count + pathAdd);
154         });
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
        Carlo methods using anithetic sampling .
162     /// </summary>
163     /// <param name = "T">The maturity date of the option in years.</param>
        >
164     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
        our path generator to take to reach time T.</param>
165     /// <param name = "numberPaths">The number of simulations we wish to
        run.</param>
166     /// <returns>Option price.</returns>
167     public double EuropeanCallOptionPriceMCAnithetic(double T, int
        numberTimeStepsPerPath, int numberPaths)
168     {
169         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
170         {
171             throw new System.ArgumentException("Parameters must be positive
                ");
172         }

```

```

173
174         // set up counter and path generator
175         double count = 0;
176         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
177
178         // generate two paths half as many times, evaluate payoff on each
179         int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
180         for (int i = 0; i < halfNumPaths; i++)
181         {
182             count += Math.Max(path.PathGeneratorAnithetic(T, S,
183                             numberTimeStepsPerPath)[0] - K, 0) + Math.Max(path.
184                             PathGeneratorAnithetic(T, S, numberTimeStepsPerPath)[1] - K
185                             , 0);
186
187         }
188
189         // average and discount
190         return Math.Exp(-r * T) * count / (2 * halfNumPaths);
191     }
192
193     /// <summary>
194     /// Prices a European call option within the Heston model using Monte
195     /// Carlo methods using both parallelisation and anithetic sampling .
196     /// </summary>
197     /// <param name = "T">The maturity date of the option in years.</param>
198     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
199     /// our path generator to take to reach time T.</param>
200     /// <param name = "numberPaths">The number of simulations we wish to
201     /// run.</param>
202     /// <returns>Option price.</returns>
203     public double EuropeanCallOptionPriceMCAnitheticParallel(double T, int
204         numberTimeStepsPerPath, int numberPaths)
205     {
206         if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
207         {
208             throw new System.ArgumentException("Parameters must be positive
209                 ");
210         }
211
212         // set up counter and path generator
213         double count = 0;
214         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
215
216         // generate two paths half as many times, in parallel, evaluate
217         // payoff on each
218         int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
219         Parallel.For(0, halfNumPaths, (i) =>
220         {
221             double[] paths = path.PathGeneratorAnithetic(T, S,
222                             numberTimeStepsPerPath);
223             double pathAdd = Math.Max(paths[0] - K, 0) + Math.Max(paths[1]
224                             - K, 0);
225             Interlocked.Exchange(ref count, count + pathAdd);
226         });
227
228         // average and discount
229         return Math.Exp(-r * T) * count / (2.0 * halfNumPaths);
230     }
231
232     /// <summary>
233     /// Prices a European put option within the Heston model using Monte
234     /// Carlo methods using both parallelisation and anithetic sampling .
235     /// </summary>
236     /// <param name = "T">The maturity date of the option in years.</param>
237     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
238     /// our path generator to take to reach time T.</param>
239     /// <param name = "numberPaths">The number of simulations we wish to
240     /// 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     /// <summary>
252     /// Checks that the times used for pricing Asian options make sense.
253     /// </summary>
254     /// <param name = "T">An array containing the onservatation times of the
        Asian option.</param>
255     /// <param name = "exerciseT">The Asian option's exercise time.</param>
256     private void CheckAsianOptionInputs(double[] T, double exerciseT)
257     {
258
259         if (T.Length == 0)
260         {
261             throw new System.ArgumentException("Need at least one
                monitoring date for Asian option.");
262
263         if (T[0] <= 0)
264             throw new System.ArgumentException("The first monitoring date
                must be positive.");
265
266         for (int i = 1; i < T.Length; ++i)
267         {
268             if (T[i - 1] >= T[i])
269                 throw new System.ArgumentException("Monitoring dates must
                be increasing");
270         }
271
272         if (T[T.Length - 1] > exerciseT)
273             throw new System.ArgumentException("Last monitoring time must
                not be greater than the exercise time");
274
275     }
276
277     /// <summary>
278     /// Prices an Asian call option within the Heston model using Monte
        Carlo methods.
279     /// </summary>
280     /// <param name = "T">An array containing the onservatation times of the
        Asian option.</param>
281     /// <param name = "exerciseT">The Asian option's exercise time.</param>
    >

```

```

280     /// <param name = "numberPaths">The number of simulations we wish to
        run.</param>
281     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
        our path generator to take to reach time exerciseT.</param>
282     /// <returns>Option price.</returns>
283     public double PriceAsianCallMC(double[] T, double exerciseT, int
        numberPaths, int numberTimeStepsPerPath)
284     {
285         if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
286         {
287             throw new System.ArgumentException("Monte Carlo settings must
                be positive");
288         }
289
290         // check parameters make sense
291         CheckAsianOptionInputs(T, exerciseT);
292
293         // set up parameter, path generator and counter
294         int M = T.Length;
295         MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
296         double pathCounter = 0;
297
298         // generate paths between observation times, evaluate payoff
                function
299         for (int i = 0; i < numberPaths; i++)
300         {
301             double priceCount = 0;
302             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
                    appropriate number time steps
311                 int stepNumber = (int)Math.Ceiling(deltaT *
                    numberTimeStepsPerPath / exerciseT);
312                 holder = path.PathGenerator(deltaT, holder, stepNumber);
313                 priceCount += holder;
314             }
315             double pathPayoff = Math.Max(priceCount / M - K, 0);
316             pathCounter += pathPayoff;
317         }
318
319         // average and discount
320         return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
321     }
322
323     /// <summary>
324     /// Prices an Asian put option within the Heston model using Monte
        Carlo methods.
325     /// </summary>
326     /// <param name = "T">An array containing the onservation times of the
        Asian option.</param>
327     /// <param name = "exerciseT">The Asian option's exercise time.</param
        >
328     /// <param name = "numberPaths">The number of simulations we wish to
        run.</param>
329     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
        our path generator to take to reach time exerciseT.</param>
330     /// <returns>Option price.</returns>
331     public double PriceAsianPutMC(double[] T, double exerciseT, int
        numberPaths, int numberTimeStepsPerPath)
332     {
333         if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)

```



```

334     {
335         throw new System.ArgumentException("Monte Carlo settings must
           be positive");
336     }
337
338     // check parameters make sense
339     CheckAsianOptionInputs(T, exerciseT);
340
341     // set up parameter, path generator and counter
342     int M = T.Length;
343     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
344     double pathCounter = 0;
345
346     // generate paths between observation times, evaluate payoff
347     function
348     for (int i = 0; i < numberPaths; i++)
349     {
350         double priceCount = 0;
351         double holder = S;
352         double deltaT = T[0];
353
354         for (int j = 0; j < M; j++)
355         {
356             if (j > 0)
357                 deltaT = T[j] - T[j - 1];
358
359             // generate path from previous time point to next with
360             // appropriate number time steps
361             int stepNumber = (int)Math.Ceiling(deltaT *
362                 numberTimeStepsPerPath / exerciseT);
363             holder = path.PathGenerator(deltaT, holder, stepNumber);
364             priceCount += holder;
365         }
366         double pathPayoff = Math.Max(K - priceCount / M, 0);
367         pathCounter += pathPayoff;
368     }
369
370     // average and discount
371     return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
372 }
373
374 /// <summary>
375 /// Prices an Asian call option within the Heston model using Monte
376 /// Carlo methods with parallelisation.
377 /// </summary>
378 /// <param name = "T">An array containing the onservation times of the
379 /// Asian option.</param>
380 /// <param name = "exerciseT">The Asian option's exercise time.</param>
381 >
382 /// <param name = "numberPaths">The number of simulations we wish to
383 run.</param>
384 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
385 our path generator to take to reach time exerciseT.</param>
386 /// <returns>Option price.</returns>
387 public double PriceAsianCallMCPParallel(double[] T, double exerciseT,
388     int numberPaths, int numberTimeStepsPerPath)
389 {
390     if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
391     {
392         throw new System.ArgumentException("Monte Carlo settings must
           be positive");
393     }
394
395     // check parameters make sense
396     CheckAsianOptionInputs(T, exerciseT);
397
398     // set up parameter, path generator and counter

```

```

390     int M = T.Length;
391     double pathCounter = 0;
392     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
393
394     // generate paths between observation times in parallel, evaluate
395     // payoff function
396     Parallel.For(0, numberPaths, (i) =>
397     {
398         double priceCount = 0;
399         double holder = S;
400         double deltaT = T[0];
401         Parallel.For(0, M, (j) =>
402         {
403             if (j > 0)
404                 deltaT = T[j] - T[j - 1];
405
406             // generate path from previous time point to next with
407             // appropriate number time steps
408             int stepNumber = (int)Math.Ceiling(deltaT *
409                 numberTimeStepsPerPath / exerciseT);
410             holder = path.PathGenerator(deltaT, holder, stepNumber);
411             Interlocked.Exchange(ref priceCount, priceCount + holder);
412         });
413
414         double pathPayoff = Math.Max((priceCount / M) - K, 0);
415         Interlocked.Exchange(ref pathCounter, pathCounter + pathPayoff);
416     });
417
418     // average and discount
419     return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
420 }
421
422 /// <summary>
423 /// Prices a Lookback option within the Heston model using Monte Carlo
424 /// methods.
425 /// </summary>
426 /// <param name = "exerciseT">The Lookback option's exercise time.</
427 /// param>
428 /// <param name = "numberPaths">The number of simulations we wish to
429 /// run.</param>
430 /// <param name = "numberTimeStepsPerPath">The number of steps we wish
431 /// our path generator to take to reach time exerciseT.</param>
432 /// <returns>Option price.</returns>
433 public double PriceLookbackCallMC(double exerciseT, int numberPaths,
434     int numberTimeStepsPerPath)
435 {
436     if(exerciseT <= 0 || numberPaths <=0 || numberTimeStepsPerPath <=
437         0)
438     {
439         throw new System.ArgumentException("Parameters must be positive
440             ");
441     }
442
443     // set up counter, parameter and path generator
444     double pathCounter = 0;
445     double deltaT = exerciseT / numberTimeStepsPerPath;
446     MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
447
448     // generate paths of length numberTimeStepsPerPaths, evaluate
449     // payoff on each
450     for (double i = 0; i < numberPaths; i++)
451     {
452         double min = S;
453         double holder = S;
454         for (double j = 0; j <= exerciseT; j += deltaT)
455         {
456             // take a step with appropriate time change and start point

```

```

445         holder = path.PathGenerator(deltaT, holder, 1);
446
447         // keep track of minimum
448         if (holder < min)
449             min = holder;
450     }
451
452     pathCounter += holder - min;
453 }
454
455 // average and discount
456 return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
457 }
458 }
459
460 }

```

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
12     /// to price options within the Heston
13     /// model in the class OptionsMC
14     /// </summary>
15     public class MCPaths
16     {
17         private double r;
18         private double rho;
19         private double kappaStar;
20         private double thetaStar;
21         private double sigma;
22         private double v;
23
24         public MCPaths(double r, double kappaStar, double thetaStar, double
25             sigma, double rho, double v)
26         {
27             if (r <= 0 || sigma <= 0 || v <= 0)
28             {
29                 throw new System.ArgumentException("r, sigma, v must be
30                     positive");
31             }
32
33             this.r = r; this.rho = rho; this.kappaStar = kappaStar;
34             this.thetaStar = thetaStar; this.sigma = sigma; this.v = v;
35         }
36
37         /// <summary>
38         /// Returns a simulated future price for a risky asset within the
39         /// Heston model.
40         /// </summary>
41         /// <param name = "T">The future time at which we wish to simulate the
42         /// asset price.</param>
43         /// <param name = "S">The initial asset price.</param>
44         /// <param name = "numberTimeStepsPerPath">The number of steps we wish
45         /// the scheme to take to reach time T.</param>
46         /// <returns>Simulated asset price.</returns>
47         public double PathGenerator(double T, double S, int
48             numberTimeStepsPerPath)

```

```

42     {
43         if (T <= 0 || S <= 0 || numberTimeStepsPerPath <= 0)
44         {
45             throw new System.ArgumentException("Parameters must be positive
46                 ");
47         }
48
49         //sample from normal dist
50         double[] x1 = new double[numberTimeStepsPerPath];
51         Normal.Samples(x1, 0, 1);
52         double[] x2 = new double[numberTimeStepsPerPath];
53         Normal.Samples(x2, 0, 1);
54
55         //set up parameters
56         double tau = T / numberTimeStepsPerPath;
57         double sqrtTau = Math.Sqrt(tau);
58         double sqrtOneMinusRhoSquared = Math.Sqrt(1 - rho * rho);
59         double alpha = (4 * kappaStar * thetaStar - sigma * sigma) / 8.0;
60         double beta = -kappaStar / 2.0;
61         double gamma = sigma / 2.0;
62
63         // set up holder variables for Monte Carlo
64         double y = Math.Sqrt(v);
65         double s = S;
66
67         // update holder variables iteratively according to MC scheme.
68         for (int i = 0; i < numberTimeStepsPerPath; i++)
69         {
70             double deltaZ1 = sqrtTau * x1[i];
71             double deltaZ2 = sqrtTau * (rho * x1[i] +
72                 sqrtOneMinusRhoSquared * x2[i]);
73             s = s + r * s * tau + y * s * deltaZ1;
74             double a = (y + gamma * deltaZ2) / (2 * (1 - beta * tau));
75             y = a + Math.Sqrt(a * a + alpha * tau / (1 - beta * tau));
76         }
77
78         return s;
79     }
80
81     /// <summary>
82     /// Returns a simulated future price for a risky asset within the
83     /// Heston model using anithetic sampling with a view to reducing
84     /// variance.
85     /// </summary>
86     /// <param name = "T">The future time at which we wish to simulate the
87     /// asset price.</param>
88     /// <param name = "S">The initial asset price.</param>
89     /// <param name = "numberTimeStepsPerPath">The number of steps we wish
90     /// the scheme to take to reach time T.</param>
91     /// <returns>Simulated asset price.</returns>
92     public double[] PathGeneratorAnithetic(double T, double S, int
93         numberTimeStepsPerPath)
94     {
95         if (T <= 0 || S <= 0 || numberTimeStepsPerPath <= 0)
96         {
97             throw new System.ArgumentException("Parameters must be positive
98                 ");
99         }
100
101         // sample from normal dist
102         double[] x1 = new double[numberTimeStepsPerPath];
103         double[] x2 = new double[numberTimeStepsPerPath];
104         Normal.Samples(x1, 0, 1);
105         Normal.Samples(x2, 0, 1);
106
107         //set up parameters
108         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
114         // negative of normal sample for anitheic path
115         for (int i = 0; i < numberTimeStepsPerPath; i++)
116         {
117             double deltaZ1 = sqrtTau * x1[i];
118             double deltaZ2 = sqrtTau * (rho * x1[i] +
119                 sqrtOneMinusRhoSquared * x2[i]);
120             s = s + r * s * tau + y * s * deltaZ1;
121             s1 = s1 + r * s1 * tau - y1 * s1 * deltaZ1;
122             double a = (y + gamma * deltaZ2) / (2 * (1 - beta * tau));
123             double aa = (y1 - gamma * deltaZ2) / (2 * (1 - beta * tau));
124             y = a + Math.Sqrt(a * a + alpha * tau / (1 - beta * tau));
125             y1 = aa + Math.Sqrt(aa * aa + alpha * tau / (1 - beta * tau));
126         }
127
128         // return array of end points of the two paths.
129         double[] sArray = { s, s1 };
130         return sArray;
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             maxIts = defaultMaxIts;
60             marketList = new List<MarketData>();
61             r = 0;
62             S = 0;
63         }
64
65         public Calibrator(double r, double S, int maxIts, double accuracy)
66         {
67             if (r <= 0 || S <= 0 || maxIts <= 0 )
68             {
69                 throw new System.ArgumentException("Parameters must be positive
70                 ");
71             }
72
73             this.accuracy = accuracy;
74             this.maxIts = maxIts;
75             marketList = new List<MarketData>();
76             this.r = r;
77             this.S = S;
78         }
79
80         /// <summary>
81         /// Sets the parameters which will be used as a starting point by the
82         /// calibrator
83         /// </summary>
84         public void SetGuessParameters(double kappaStar, double thetaStar,
85         double sigma, double rho, double v)
86         {
87             if (sigma <= 0 || v <= 0)
88             {
89                 throw new System.ArgumentException("Sigma, v must be positive")
90                 ;
91             }
92
93             Options e = new Options(r, S, kappaStar, thetaStar, sigma, rho, v);
94             calibratedParams = e.ParamsAsArray();
95         }
96
97         /// <summary>

```

```

93     /// Adds the details of a real world option to the list marketList of
94     data which will be used for calibration.
95     /// </summary>
96     /// <param name="K">Observed option's strike price.</param>
97     /// <param name="T">Observed option's maturity time.</param>
98     /// <param name="Price">Observed options price.</param>
99     public void AddObservedOption(double K, double T, double Price)
100     {
101         if (K <= 0 || T <= 0 || Price <= 0)
102         {
103             throw new System.ArgumentException("Parameters must be positive
104             ");
105         }
106         MarketData observedOption;
107         observedOption.K = K;
108         observedOption.T = T;
109         observedOption.Price = Price;
110         marketList.Add(observedOption);
111     }
112     /// <summary>
113     /// Calculates the mean squared error between the European call prices
114     /// of an instance, options,
115     /// of the class Options and the market prices found in marketList
116     /// </summary>
117     /// <param name="options">An instance of the class Options.</param>
118     public double CalculateMeanSquaredErrorBetweenModelAndMarket(Options
119     options)
120     {
121         double mse = 0;
122         foreach (MarketData data in marketList)
123         {
124             double T = data.T;
125             double K = data.K;
126             double price = options.EuropeanCallPrice(T, K);
127             double diff = price - data.Price;
128             mse += diff * diff;
129         }
130         return mse;
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     /// <summary>
142     /// Calibrates the model parameters to fit the market data as closely
143     /// as possible
144     /// </summary>
145     public void Calibrate()
146     {
147         // set up for calibration
148         outcome = CalibrationOutcome.NotStarted;
149         double[] initialParams = new double[Options.numberParams];
150
151         if (calibratedParams == null)
152         {
153             throw new System.Exception("Please add an initial guess for
154             parameters");
155         }
156     }

```

```

153     }
154
155     calibratedParams.CopyTo(initialParams, 0);
156     double epsg = accuracy;
157     double epsf = accuracy;
158     double epsx = accuracy;
159     double diffstep = 1.0e-6;
160     int maxits = maxIts;
161     double stpmax = 0.05;
162
163     alglib.minlbfgsstate state;
164     alglib.minlbfgsreport rep;
165     alglib.minlbfgscreatef(5, initialParams, diffstep, out state);
166     alglib.minlbfgssetcond(state, epsg, epsf, epsx, maxits);
167     alglib.minlbfgssetstpmax(state, stpmax);
168
169     // calibrate and return outcome, error
170     alglib.minlbfgsoptimize(state, CalibrationObjectiveFunction, null,
171                             null);
172     double[] resultParams = new double[Options.numberParams];
173     alglib.minlbfgsresults(state, out resultParams, out rep);
174
175     System.Console.WriteLine("Termination type: {0}", rep.
176                             terminationtype);
177     System.Console.WriteLine("Num iterations {0}", rep.iterationscount)
178     ;
179     System.Console.WriteLine("{0}", alglib.ap.format(resultParams, 5));
180
181     if (rep.terminationtype == 1
182         || rep.terminationtype == 2
183         || rep.terminationtype == 4)
184     {
185         outcome = CalibrationOutcome.FinishedOK;
186         calibratedParams = resultParams;
187     }
188     else if (rep.terminationtype == 5)
189     {
190         outcome = CalibrationOutcome.FailedMaxItReached;
191         calibratedParams = resultParams;
192     }
193     else
194     {
195         outcome = CalibrationOutcome.FailedOtherReason;
196         throw new CalibrationFailedException("Heston model calibration
197                                             failed badly.");
198     }
199 }
200
201 /// <summary>
202 /// Obtains the calibration status of the model, as well as the models
203 /// pricing error.
204 /// </summary>
205 public void GetCalibrationStatus(ref CalibrationOutcome calibOutcome,
206                                 ref double pricingError)
207 {
208     calibOutcome = outcome;
209     Options m = new Options(r, S, calibratedParams);
210     pricingError = CalculateMeanSquaredErrorBetweenModelAndMarket(m);
211 }
212
213 /// <summary>
214 /// Creates an instance of the class Options with the calibrated
215 /// parameters.
216 /// </summary>
217 /// <returns>Calibrated model.</returns>
218 public Options GetCalibratedModel()

```



```

213     {
214         Options m = new Options(r, S, calibratedParams);
215         return m;
216     }
217
218     }
219 }

```

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         double theta;
20         double sigma;
21         double v;
22         double rho;
23         int numberTrials;
24         int numberTimeSteps;
25         PayoffType p;
26         IEnumerable<double> timeList;
27         double K;
28         double accuracy;
29
30         public InterfaceFill(double T, double kappa, double theta, double sigma
31             , double rho, double v, int numberTrials, int numberTimeSteps,
32             PayoffType p, double[] timeArray, double K, double accuracy)
33         {
34             this.T = T; this.kappa = kappa; this.theta = theta; this.sigma =
35                 sigma;
36             this.v = v; this.rho = rho; this.numberTrials = numberTrials; this.
37                 numberTimeSteps = numberTimeSteps;
38             this.p = p; timeList = timeArray.AsEnumerable(); this.K = K;
39             this.accuracy = accuracy;
40         }
41
42         double IOption.Maturity => T;
43
44         double IVarianceProcessParameters.Kappa => kappa;
45
46         double IVarianceProcessParameters.Theta => theta;
47
48         double IVarianceProcessParameters.Sigma => sigma;
49
50         double IVarianceProcessParameters.V0 => v;
51
52         double IVarianceProcessParameters.Rho => rho;
53
54     }
55 }

```

```

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
12     /// and ICalibrationResult
13     /// </summary>
14     public class OtherInterfaceFill : IHestonModelParameters,
15         ICalibrationResult
16     {
17         double S;
18         double r;
19         IVarianceProcessParameters paramss;
20         CalibrationOutcome c;
21         double error;
22
23         public OtherInterfaceFill(double T, double kappa, double theta, double
24             sigma, double rho, double v, int numberTrials, int numberTimeSteps,
25             double S, double r, CalibrationOutcome c, double error)
26         {
27             this.S = S; this.r = r;
28             double[] TT = { 0 };
29             InterfaceFill fill = new InterfaceFill(T, kappa, theta, sigma, rho,
30                 v, numberTrials, numberTimeSteps, 0, TT, 100, 0);
31             paramss = fill;
32             this.c = c; this.error = error;
33         }
34
35         double IHestonModelParameters.InitialStockPrice => S;
36
37         double IHestonModelParameters.RiskFreeRate => r;
38
39         IVarianceProcessParameters IHestonModelParameters.VarianceParameters =>
40             paramss;
41
42         CalibrationOutcome ICalibrationResult.MinimizerStatus => c;
43
44         double ICalibrationResult.PricingError => error;
45     }
46 }

```

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,
20             double sigma, double rho, double v, int numberTrials, int
21             numberTimeSteps, double S, double r, CalibrationOutcome c, double
22             error)
23         {
24             this.c = c; this.error = error;
25             OtherInterfaceFill fill = new OtherInterfaceFill(T, kappa, theta,
26                 sigma, rho, v, numberTrials, numberTimeSteps, S, r, c, error);
27             paramms = fill;
28         }
29
30         IHestonModelParameters IHestonCalibrationResult.Parameters => paramms;
31
32         CalibrationOutcome ICalibrationResult.MinimizerStatus => c;
33
34         double ICalibrationResult.PricingError => error;
35     }
36 }
```

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         }
27     }
28 }
```

```

23         option = fill;
24     }
25
26     IEuropeanOption IOptionMarketData<IEuropeanOption>.Option => option;
27
28     double IOptionMarketData<IEuropeanOption>.Price => price;
29 }
30 }

```

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
11     /// <summary>
12     /// This class will be used for grading.
13     /// Don't remove any of the methods and don't modify their signatures. Don'
14     /// t change the namespace.
15     /// Your code should be implemented in other classes (or even projects if
16     /// you wish), and the relevant functionality should only be called here
17     /// and outputs returned.
18     /// You don't need to implement the interfaces that have been provided if
19     /// you don't want to.
20     /// </summary>
21     public static class Heston
22     {
23         /// <summary>
24         /// Method for calibrating the heston model.
25         /// </summary>
26         /// <param name="guessModelParameters">Object implementing
27         /// IHestonModelParameters interface containing the risk-free rate,
28         /// initial stock price
29         /// and initial guess parameters to be used in the calibration.</param>
30         /// <param name="referenceData">A collection of objects implementing
31         /// IOptionMarketData<IEuropeanOption> interface. These should contain
32         /// the reference data used for calibration.</param>
33         /// <param name="calibrationSettings">An object implementing
34         /// ICalibrationSettings interface.</param>
35         /// <returns>Object implementing IHestonCalibrationResult interface
36         /// which contains calibrated model parameters and additional
37         /// diagnostic information</returns>
38         public static IHestonCalibrationResult CalibrateHestonParameters(
39             IHestonModelParameters guessModelParameters, IEnumerable<
40             IOptionMarketData<IEuropeanOption>> referenceData,
41             ICalibrationSettings calibrationSettings)
42         {
43             // set up calibrator
44             Calibrator cal = new Calibrator(guessModelParameters.RiskFreeRate,
45                 guessModelParameters.InitialStockPrice, calibrationSettings.
46                 MaximumNumberOfIterations, calibrationSettings.Accuracy);
47             cal.SetGuessParameters(guessModelParameters.VarianceParameters.
48                 Kappa, guessModelParameters.VarianceParameters.Theta,
49                 guessModelParameters.VarianceParameters.Sigma,
50                 guessModelParameters.VarianceParameters.Rho,
51                 guessModelParameters.VarianceParameters.V0);
52
53             // add market data
54             foreach (IOptionMarketData<IEuropeanOption> data in referenceData)
55             {

```

```

37         cal.AddObservedOption(data.Option.StrikePrice, data.Option.
38             Maturity, data.Price);
39     }
40     // calibrate, get error, outcome, parameters
41     cal.Calibrate();
42     double error = 0;
43     HestonCalibrationAndPricing.CalibrationOutcome outcome =
44         HestonCalibrationAndPricing.CalibrationOutcome.NotStarted;
45     cal.GetCalibrationStatus(ref outcome, ref error);
46     Options e = cal.GetCalibratedModel();
47     double[] paramArray = e.ParamsAsArray();
48     CalibrationOutcome outcome1 = (CalibrationOutcome)outcome;
49
50     // implement and return IHestonCalibrationResult
51     AnotherInterfaceFill fill = new AnotherInterfaceFill(0, paramArray[
52         Options.kappaIndex], paramArray[Options.thetaIndex], paramArray[
53         Options.sigmaIndex], paramArray[Options.rhoIndex], paramArray[
54         Options.vIndex], 0, 0, 0, 0, outcome1, error);
55     return fill;
56 }
57
58 /// <summary>
59 /// Price a European option in the Heston model using the Heston
60 /// formula. This should be accurate to 5 decimal places
61 /// </summary>
62 /// <param name="parameters">Object implementing IHestonModelParameters
63 /// interface, containing model parameters.</param>
64 /// <param name="europeanOption">Object implementing IEuropeanOption
65 /// interface, containing the option parameters.</param>
66 /// <returns>Option price</returns>
67 public static double HestonEuropeanOptionPrice(IHestonModelParameters
68     parameters, IEuropeanOption europeanOption)
69 {
70     Options eur = new Options(parameters.RiskFreeRate, parameters.
71         InitialStockPrice,
72         parameters.VarianceParameters.Kappa, parameters.
73         VarianceParameters.Theta, parameters.VarianceParameters.
74         Sigma,
75         parameters.VarianceParameters.Rho, parameters.VarianceParameters
76         .VO);
77
78     if (europeanOption.Type == 0)
79     {
80         return eur.EuropeanCallPrice(europeanOption.Maturity,
81             europeanOption.StrikePrice);
82     }
83     else
84     {
85         return eur.EuropeanPutPrice(europeanOption.Maturity,
86             europeanOption.StrikePrice);
87     }
88 }
89
90 /// <summary>
91 /// Price a European option in the Heston model using the Monte-Carlo
92 /// method. Accuracy will depend on number of time steps and samples
93 /// </summary>
94 /// <param name="parameters">Object implementing IHestonModelParameters
95 /// interface, containing model parameters.</param>
96 /// <param name="europeanOption">Object implementing IEuropeanOption
97 /// interface, containing the option parameters.</param>
98 /// <param name="monteCarloSimulationSettings">An object implementing
99 /// IMonteCarloSettings object and containing simulation settings.</
100 param>
101 /// <returns>Option price</returns>
102 public static double HestonEuropeanOptionPriceMC(IHestonModelParameters
103     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
        interface, containing model parameters.</param>
100    /// <param name="asianOption">Object implementing IAsian interface,
        containing the option parameters.</param>
101    /// <param name="monteCarloSimulationSettings">An object implementing
        IMonteCarloSettings object and containing simulation settings.</
        param>
102    /// <returns>Option price</returns>
103    public static double HestonAsianOptionPriceMC(IHestonModelParameters
        parameters, IAsianOption asianOption, IMonteCarloSettings
        monteCarloSimulationSettings)
104    {
105        OptionsMC asian = new OptionsMC(parameters.RiskFreeRate,
            asianOption.StrikePrice, parameters.VarianceParameters.Kappa,
106            parameters.VarianceParameters.Theta, parameters.
                VarianceParameters.Sigma, parameters.VarianceParameters.Rho,
107            parameters.VarianceParameters.V0, parameters.InitialStockPrice);
108
109        if (asianOption.Type == 0)
110        {
111            return asian.PriceAsianCallMC(asianOption.MonitoringTimes.
                ToArray(), asianOption.Maturity,
112                monteCarloSimulationSettings.NumberOfTrials,
                monteCarloSimulationSettings.NumberOfTimeSteps);
113        }
114        else
115        {
            return asian.PriceAsianPutMC(asianOption.MonitoringTimes.
                ToArray(), asianOption.Maturity,
116                monteCarloSimulationSettings.NumberOfTrials,
                monteCarloSimulationSettings.NumberOfTimeSteps);
117        }
118
119        /// <summary>
120        /// Price a lookback option in the Heston model using the
121        /// a Monte-Carlo method. Accuracy will depend on number of time steps
        and samples </summary>
122        /// <param name="parameters">Object implementing IHestonModelParameters
        interface, containing model parameters.</param>

```

```

123     /// <param name="maturity">An object implementing IOption interface and
124     containing option's maturity</param>
125     /// <param name="monteCarloSimulationSettings">An object implementing
126     IMonteCarloSettings object and containing simulation settings.</
127     param>
128     /// <returns>Option price</returns>
129     public static double HestonLookbackOptionPriceMC(IHestonModelParameters
130     parameters, IOption maturity, IMonteCarloSettings
131     monteCarloSimulationSettings)
132     {
133         OptionsMC lookback = new OptionsMC(parameters.RiskFreeRate,
134         parameters.VarianceParameters.Kappa,
135         parameters.VarianceParameters.Theta, parameters.
136         VarianceParameters.Sigma, parameters.
137         VarianceParameters.Rho,
138         parameters.VarianceParameters.V0, parameters.
139         InitialStockPrice);
140
141         return lookback.PriceLookbackCallMC(maturity.Maturity,
142         monteCarloSimulationSettings.NumberOfTrials,
143         monteCarloSimulationSettings.NumberOfTimeSteps);
144     }
145 }

```
