OOPA

Adam Keogh

January 2019

Contents

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 neccesary 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

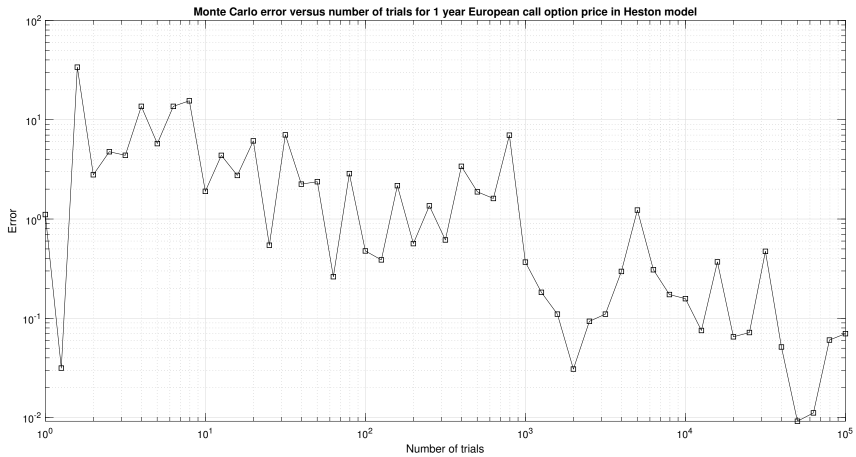
$$\begin{aligned} r &= 10\% \\ \theta^* &= 6\%, \ \kappa^* = 200\%, \ \sigma = 40\%, \ \rho = 50\%, \ v = 4\% \\ S &= 100. \end{aligned} \tag{2}$$

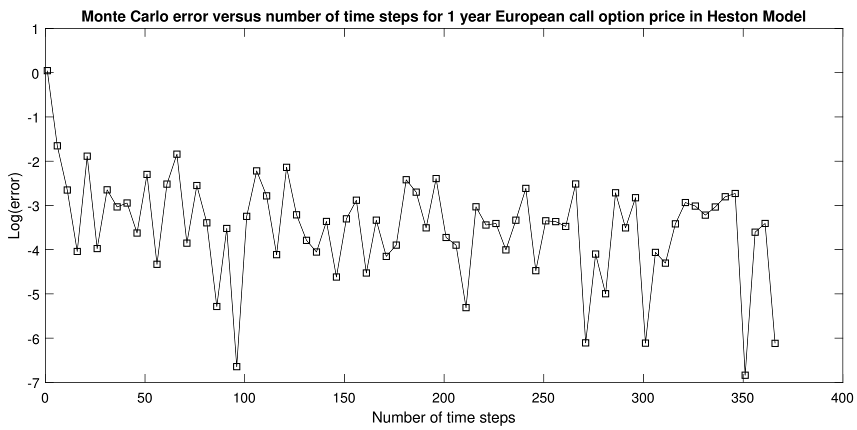
Strike K	Option exercise T	Price $C(0, S, v)$
100	1	13.6
100	2	22.6
100	3	30.1
100	4	37.2
100	15	77.9

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.

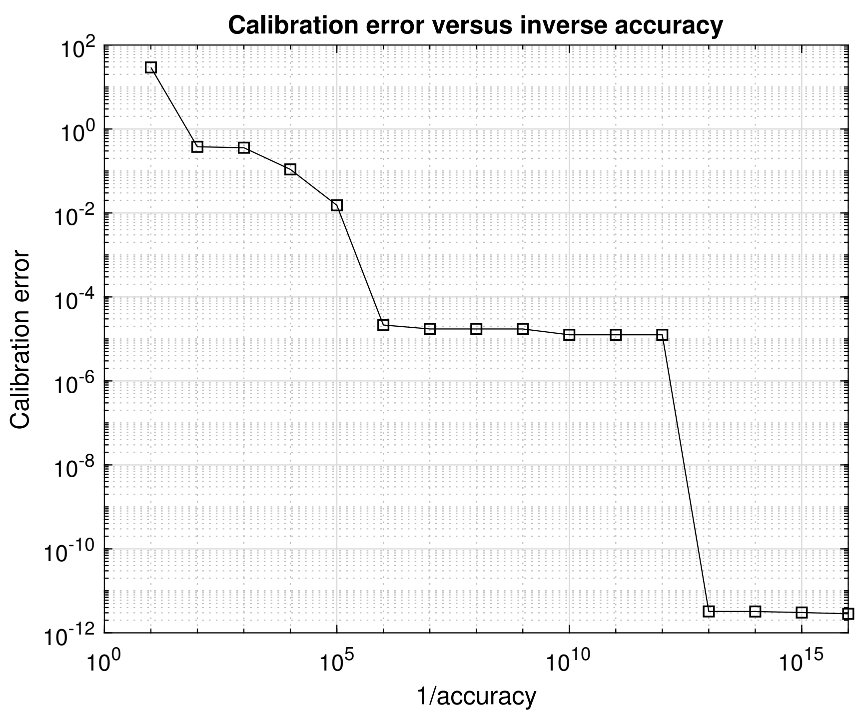
What I found was that with accuracy 1×10^{-3} the calibrated parameter values would vary dependent on the initial guess parameters, while with accuracy 1×10^{-15} they would converge to the below parameters from most reasonable starting points which I tried. The calibration returned FinishedOk in the 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, it's 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	$T_1, ..., T_m$	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 if often the case for anthethic samples ($Y_1 = -Y_2$). This was the case in my implementation of this method in the Heston model for pricing European options. Anithetic 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	Anithetic 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 anithetic sampling to successfully reduce both the computational time and variance of my Monte Carlo options pricing.

Code

Options.cs class

```

using System;
using MathNet.Numerics.Integration;
using System.Numerics;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HestonCalibrationAndPricing
{
    /// <summary>
    /// This class prices options within the Heston model
    /// </summary>
    public class Options
    {
        public const int numberParams = 5;
        public const int kappaIndex = 0;
        public const int thetaIndex = 1;
        public const int sigmaIndex = 2;
        public const int rhoIndex = 3;
        public const int vIndex = 4;

        private double r;
        private double kappaStar;
        private double thetaStar;
        private double sigma;
        private double rho;
        private double v;
        private double S;


        public Options(double r, double S, double kappaStar, double thetaStar,
            double sigma, double rho, double v)
        {
            if (r <= 0 || S <= 0 || sigma <= 0 || v <= 0)
```

```
            {
                throw new System.ArgumentException("r, S, sigma, v must be
                    positive");
            }
            this.r = r; this.S = S; this.kappaStar = kappaStar; this.thetaStar
                = thetaStar;
            this.sigma = sigma; this.rho = rho; this.v = v;
        }

        public Options(double r, double S, double[] parameters)
        {
            this.r = r; this.S = S;
            kappaStar = parameters[kappaIndex];
            thetaStar = parameters[thetaIndex];
            sigma = parameters[sigmaIndex];
            rho = parameters[rhoIndex];
            v = parameters[vIndex];

            if (r <= 0 || S <= 0 || sigma <= 0 || v <= 0)
            {
                throw new System.ArgumentException("r, S, sigma, v must be
                    positive");
            }

        }

         /// <summary>
         /// Prices a European call option within the Heston model.
         /// </summary>
         /// <param name = "T">The maturity date of the option in years.</param
             >
         /// <param name = "K">The options' strike price.</param>
         /// <returns>Option price.</returns>
        public double EuropeanCallPrice(double T, double K)
        {
            if (T < 0 || K < 0)
            {
                throw new System.ArgumentException("T, K must be non-negative")
                    ;
            }

            // set up paramters
            double[] b = { kappaStar - rho * sigma, kappaStar };
            double[] u = { 0.5, -0.5 };
            double a = kappaStar * thetaStar;
            Complex i = new Complex(0, 1);

            // function implementing part of Heston formula
            Func<int, double, double> RealP = (j, phi) =>
            {

                Complex temp1 = new Complex(-b[j], rho * sigma * phi);
                Complex tempp1 = new Complex(-phi * phi, 2 * u[j] * phi);
                Complex d = Complex.Pow(temp1 * temp1 - sigma * sigma * tempp1,
                     0.5);
                Complex g = (b[j] - rho * sigma * phi * i - d) * Complex.Pow(b[
                    j] - rho * sigma * phi * i + d, -1);

                Complex c = r * phi * T * i + (a / (sigma * sigma)) * ((b[j] -
                    rho * sigma * phi * i - d) * T - 2 * Complex.Log((1 - g *
                    Complex.Exp(-T * d)) / (1 - g)));
                Complex bigD = ((b[j] - rho * sigma * phi * i - d) / (sigma *
                    sigma)) * ((1 - Complex.Exp(-T * d)) / (1 - g * Complex.Exp
                    (-T * d)));
                Complex littlePhi = Complex.Exp(c + bigD * v + phi * i * Math.
                    Log(S));
                Complex value = Complex.Exp(-i * phi * Math.Log(K)) * littlePhi
```

```
                    / (i * phi);
                return value.Real;
            };

            double[] P = new double[2];

            // integrate with appropriate number of steps and length to
                approximate infinite integral
            P[0] = 0.5 + (1.0 / Math.PI) * SimpsonRule.IntegrateComposite(x =>
                RealP(0, x), 0.000001, 50, 100);
            P[1] = 0.5 + (1.0 / Math.PI) * SimpsonRule.IntegrateComposite(x =>
                RealP(1, x), 0.000001, 50, 100);

            return S * P[0] - K * Math.Exp(-r * T) * P[1];
        }

         /// <summary>
         /// Prices a European put option within the Heston model.
         /// </summary>
         /// <param name = "T">The maturity date of the option in years.</param
            >
         /// <param name = "K">The options' strike price.</param>
         /// <returns>Option price.</returns>
        public double EuropeanPutPrice(double T, double K)
        {
            if (T < 0 || K < 0)
            {
                throw new System.ArgumentException("T, K must be non-negative")
                    ;
            }

            // put call parity
            return EuropeanCallPrice(T, K) - S + K * Math.Exp(-r * T);
        }

         /// <summary>
         /// Forms an array of the model parameters.
         /// </summary>
         /// <returns>Model parameters.</returns>
        public double[] ParamsAsArray()
        {
            double[] paramsArray = new double[Options.numberParams];
            paramsArray[kappaIndex] = kappaStar;
            paramsArray[thetaIndex] = thetaStar;
            paramsArray[sigmaIndex] = sigma;
            paramsArray[rhoIndex] = rho;
            paramsArray[vIndex] = v;
            return paramsArray;
        }

    }
}
```

OptionsMC.cs class

```
using System;
using MathNet.Numerics.Distributions;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace HestonCalibrationAndPricing
{
    /// <summary>
```

```
    /// This class prices options within the Heston model using Monte Carlo methods.
    /// </summary>
    public class OptionsMC
    {
        public const int kappaIndex = 0;
        public const int thetaIndex = 1;
        public const int sigmaIndex = 2;
        public const int rhoIndex = 3;
        public const int vIndex = 4;

        private double r;
        private double K;
        private double kappaStar;
        private double thetaStar;
        private double sigma;
        private double rho;
        private double v;
        private double S;

        public OptionsMC(double r, double K, double kappaStar, double thetaStar, double sigma, double rho, double v, double S)
        {
            if (2 * kappaStar * thetaStar <= sigma * sigma)
            {
                throw new System.ArgumentException("Feller condition violated.");
            }

            if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
            {
                throw new System.ArgumentException("r, K, sigma, S must be positive");
            }

            this.r = r;
            this.K = K; this.kappaStar = kappaStar; this.thetaStar = thetaStar;
            this.sigma = sigma; this.rho = rho; this.v = v; this.S = S;
        }

        public OptionsMC(double r, double kappaStar, double thetaStar, double sigma, double rho, double v, double S)
        {
            if (2 * kappaStar * thetaStar <= sigma * sigma)
            {
                throw new System.ArgumentException("Feller condition violated.");
            }
            if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
            {
                throw new System.ArgumentException("r, K, sigma, S must be positive");
            }
            this.r = r;
            this.kappaStar = kappaStar; this.thetaStar = thetaStar;
            this.sigma = sigma; this.rho = rho; this.v = v; this.S = S;
        }

        public OptionsMC(double r, double K, double[] paramss, double S)
        {
            this.r = r;
            this.K = K; kappaStar = paramss[kappaIndex]; thetaStar = paramss[thetaIndex];
            sigma = paramss[sigmaIndex]; rho = paramss[rhoIndex]; v = paramss[vIndex]; this.S = S;
            if (2 * kappaStar * thetaStar <= sigma * sigma)
            {
```

```
                throw new System.ArgumentException("Feller condition violated.");
            }
            if (r <= 0 || K <= 0 || sigma <= 0 || S <= 0 || v <= 0)
            {
                throw new System.ArgumentException("r, K, sigma, S must be positive");
            }
        }

        /// <summary>
        /// Prices a European call option within the Heston model using Monte Carlo methods.
        /// </summary>
        /// <param name = "T">The maturity date of the option in years.</param>
        /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time T.</param>
        /// <param name = "numberPaths">The number of simulations we wish to run.</param>
        /// <returns>Option price.</returns>
        public double EuropeanCallOptionPriceMC(double T, int numberTimeStepsPerPath, int numberPaths)
        {
            if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            // set up counter and path generator
            double count = 0;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

            // generate paths, evaluate payoff on each, then average and discount
            for (int i = 0; i < numberPaths; i++)
            {
                count += Math.Max(path.PathGenerator(T, S, numberTimeStepsPerPath) - K, 0);
            }
            return Math.Exp(-r * T) * count / numberPaths;
        }

        /// <summary>
        /// Prices a European put option within the Heston model using Monte Carlo methods.
        /// </summary>
        /// <param name = "T">The maturity date of the option in years.</param>
        /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time T.</param>
        /// <param name = "numberPaths">The number of simulations we wish to run.</param>
        /// <returns>Option price.</returns>
        public double EuropeanPutOptionPriceMC(double T, int numberTimeStepsPerPath, int numberPaths)
        {
            if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            // set up counter and path generator
            double count = 0;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
```

```
            // generate paths, evaluate payoff on each, then average and discount
            for (int i = 0; i < numberPaths; i++)
            {
                count += Math.Max(K - path.PathGenerator(T, S, numberTimeStepsPerPath), 0);
            }
            return Math.Exp(-r * T) * count / numberPaths;
        }

        /// <summary>
        /// Prices a European call option within the Heston model using Monte Carlo methods with parallelisation.
        /// </summary>
        /// <param name = "T">The maturity date of the option in years.</param>
        /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time T.</param>
        /// <param name = "numberPaths">The number of simulations we wish to run.</param>
        /// <returns>Option price.</returns>
        public double EuropeanCallOptionPriceMCParallel(double T, int numberTimeStepsPerPath, int numberPaths)
        {
            if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            // set up counter and path generator
            double count = 0;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

            // generate paths in parallel, evaluate payoff on each
            Parallel.For(0, numberPaths, (i) =>
            {
                double pathAdd = Math.Max(path.PathGenerator(T, S, numberTimeStepsPerPath) - K, 0);
                // update count in such a way as to protect thread safety
                Interlocked.Exchange(ref count, count + pathAdd);
            });

            // average and discount
            return Math.Exp(-r * T) * count / numberPaths;
        }

        /// <summary>
        /// Prices a European call option within the Heston model using Monte Carlo methods using anithetic sampling .
        /// </summary>
        /// <param name = "T">The maturity date of the option in years.</param>
        /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time T.</param>
        /// <param name = "numberPaths">The number of simulations we wish to run.</param>
        /// <returns>Option price.</returns>
        public double EuropeanCallOptionPriceMCAnithetic(double T, int numberTimeStepsPerPath, int numberPaths)
        {
            if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }
```

```
            // set up counter and path generator
            double count = 0;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

            // generate two paths half as many times, evaluate payoff on each
            int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
            for (int i = 0; i < halfNumPaths; i++)
            {
                count += Math.Max(path.PathGeneratorAnithetic(T, S, numberTimeStepsPerPath)[0] - K, 0) + Math.Max(path.PathGeneratorAnithetic(T, S, numberTimeStepsPerPath)[1] - K, 0);
            }

            // average and discount
            return Math.Exp(-r * T) * count / (2 * halfNumPaths);
        }

        /// <summary>
        /// Prices a European call option within the Heston model using Monte Carlo methods using both parallelisation and anithetic sampling.
        /// </summary>
        /// <param name = "T">The maturity date of the option in years.</param>
        /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time T.</param>
        /// <param name = "numberPaths">The number of simulations we wish to run.</param>
        /// <returns>Option price.</returns>
        public double EuropeanCallOptionPriceMCAnitheticParallel(double T, int numberTimeStepsPerPath, int numberPaths)
        {
            if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            // set up counter and path generator
            double count = 0;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

            // generate two paths half as many times, in parallel, evaluate payoff on each
            int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
            Parallel.For(0, halfNumPaths, (i) =>
            {
                double[] paths = path.PathGeneratorAnithetic(T, S, numberTimeStepsPerPath);
                double pathAdd = Math.Max(paths[0] - K, 0) + Math.Max(paths[1] - K, 0);
                Interlocked.Exchange(ref count, count + pathAdd);
            });

            // average and discount
            return Math.Exp(-r * T) * count / (2.0 * halfNumPaths);
        }

        /// <summary>
        /// Prices a European put option within the Heston model using Monte Carlo methods using both parallelisation and anithetic sampling.
        /// </summary>
        /// <param name = "T">The maturity date of the option in years.</param>
        /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time T.</param>
        /// <param name = "numberPaths">The number of simulations we wish to run.</param>
```

```
        /// <returns>Option price.</returns>
        public double EuropeanPutOptionPriceMCAnitheticParallel(double T, int numberTimeStepsPerPath, int numberPaths)
        {
            if (T <= 0 || numberTimeStepsPerPath <= 0 || numberPaths <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            // set up counter and path generator
            double count = 0;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

            // generate two paths half as many times, in parallel, evaluate payoff on each
            int halfNumPaths = (int)Math.Ceiling(numberPaths / 2.0);
            Parallel.For(0, halfNumPaths, (i) =>
            {
                double[] paths = path.PathGeneratorAnithetic(T, S, numberTimeStepsPerPath);
                double pathAdd = Math.Max(K - paths[0], 0) + Math.Max(K - paths[1], 0);
                Interlocked.Exchange(ref count, count + pathAdd);
            });

            // average and discount
            return Math.Exp(-r * T) * count / (2.0 * halfNumPaths);
        }


        /// <summary>
        /// Checks that the times used for pricing Asian options make sense.
        /// </summary>
        /// <param name = "T">An array containing the onservation times of the Asian option.</param>
        /// <param name = "exerciseT">The Asian option's exercise time.</param>
        private void CheckAsianOptionInputs(double[] T, double exerciseT)
        {
            if (T.Length == 0)
                throw new System.ArgumentException("Need at least one monitoring date for Asian option.");

            if (T[0] <= 0)
                throw new System.ArgumentException("The first monitoring date must be positive.");

            for (int i = 1; i < T.Length; ++i)
            {
                if (T[i - 1] >= T[i])
                    throw new System.ArgumentException("Monitoring dates must be increasing");
            }

            if (T[T.Length - 1] > exerciseT)
                throw new System.ArgumentException("Last monitoring time must not be greater than the exercise time");
        }

         /// <summary>
         /// Prices an Asian call option within the Heston model using Monte Carlo methods.
         /// </summary>
         /// <param name = "T">An array containing the onservation times of the Asian option.</param>
         /// <param name = "exerciseT">The Asian option's exercise time.</param>
```

```
 /// <param name = "numberPaths">The number of simulations we wish to run.</param>
 /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time exerciseT.</param>
 /// <returns>Option price.</returns>
public double PriceAsianCallMC(double[] T, double exerciseT, int numberPaths, int numberTimeStepsPerPath)
{
    if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
    {
        throw new System.ArgumentException("Monte Carlo settings must be positive");
    }

    // check parameters make sense
    CheckAsianOptionInputs(T, exerciseT);

    // set up parameter, path generator and counter
    int M = T.Length;
    MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
    double pathCounter = 0;

    // generate paths between observation times, evaluate payoff function
    for (int i = 0; i < numberPaths; i++)
    {
        double priceCount = 0;
        double holder = S;
        double deltaT = T[0];

        for (int j = 0; j < M; j++)
        {
            if (j > 0)
                deltaT = T[j] - T[j - 1];

            // generate path from previous time point to next with appropriate number time steps
            int stepNumber = (int)Math.Ceiling(deltaT * numberTimeStepsPerPath / exerciseT);
            holder = path.PathGenerator(deltaT, holder, stepNumber);
            priceCount += holder;
        }
        double pathPayoff = Math.Max(priceCount / M - K, 0);
        pathCounter += pathPayoff;
    }

    // average and discount
    return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
}

/// <summary>
 /// Prices an Asian put option within the Heston model using Monte Carlo methods.
 /// </summary>
 /// <param name = "T">An array containing the onservation times of the Asian option.</param>
 /// <param name = "exerciseT">The Asian option's exercise time.</param>
 /// <param name = "numberPaths">The number of simulations we wish to run.</param>
 /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time exerciseT.</param>
 /// <returns>Option price.</returns>
public double PriceAsianPutMC(double[] T, double exerciseT, int numberPaths, int numberTimeStepsPerPath)
{
    if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
```

```
            {
                throw new System.ArgumentException("Monte Carlo settings must be positive");
            }

            // check parameters make sense
            CheckAsianOptionInputs(T, exerciseT);

            // set up parameter, path generator and counter
            int M = T.Length;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);
            double pathCounter = 0;

            // generate paths between observation times, evaluate payoff function
            for (int i = 0; i < numberPaths; i++)
            {
                double priceCount = 0;
                double holder = S;
                double deltaT = T[0];

                for (int j = 0; j < M; j++)
                {
                    if (j > 0)
                        deltaT = T[j] - T[j - 1];

                    // generate path from previous time point to next with appropriate number time steps
                    int stepNumber = (int)Math.Ceiling(deltaT * numberTimeStepsPerPath / exerciseT);
                    holder = path.PathGenerator(deltaT, holder, stepNumber);
                    priceCount += holder;
                }
                double pathPayoff = Math.Max(K - priceCount / M, 0);
                pathCounter += pathPayoff;
            }

            // average and discount
            return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
        }

         /// <summary>
         /// Prices an Asian call option within the Heston model using Monte Carlo methods with parallelisation.
         /// </summary>
         /// <param name = "T">An array containing the onservation times of the Asian option.</param>
         /// <param name = "exerciseT">The Asian option's exercise time.</param>
         /// <param name = "numberPaths">The number of simulations we wish to run.</param>
         /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time exerciseT.</param>
         /// <returns>Option price.</returns>
        public double PriceAsianCallMCParallel(double[] T, double exerciseT, int numberPaths, int numberTimeStepsPerPath)
        {
            if (numberTimeStepsPerPath <= 0 || numberPaths <= 0)
            {
                throw new System.ArgumentException("Monte Carlo settings must be positive");
            }

            // check parameters make sense
            CheckAsianOptionInputs(T, exerciseT);

            // set up parameter, path generator and counter
```

```
            int M = T.Length;
            double pathCounter = 0;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

            // generate paths between observation times in parallel, evaluate payoff function
            Parallel.For(0, numberPaths, (i) =>
            {
                double priceCount = 0;
                double holder = S;
                double deltaT = T[0];
                Parallel.For(0, M, (j) =>
                {
                    if (j > 0)
                        deltaT = T[j] - T[j - 1];

                    // generate path from previous time point to next with appropriate number time steps
                    int stepNumber = (int)Math.Ceiling(deltaT * numberTimeStepsPerPath / exerciseT);
                    holder = path.PathGenerator(deltaT, holder, stepNumber);
                    Interlocked.Exchange(ref priceCount, priceCount + holder);
                });
                double pathPayoff = Math.Max((priceCount / M) - K, 0);
                Interlocked.Exchange(ref pathCounter, pathCounter + pathPayoff);
            });

            // average and discount
            return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
        }

        /// <summary>
        /// Prices a Lookback option within the Heston model using Monte Carlo methods.
        /// </summary>
        /// <param name = "exerciseT">The Lookback option's exercise time.</param>
        /// <param name = "numberPaths">The number of simulations we wish to run.</param>
        /// <param name = "numberTimeStepsPerPath">The number of steps we wish our path generator to take to reach time exerciseT.</param>
        /// <returns>Option price.</returns>
        public double PriceLookbackCallMC(double exerciseT, int numberPaths, int numberTimeStepsPerPath)
        {
            if(exerciseT <= 0 || numberPaths <=0 || numberTimeStepsPerPath <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            // set up counter, parameter and path generator
            double pathCounter = 0;
            double deltaT = exerciseT / numberTimeStepsPerPath;
            MCPaths path = new MCPaths(r, kappaStar, thetaStar, sigma, rho, v);

            // generate paths of length numberTimeStepsPerPaths, evaluate payoff on each
            for (double i = 0; i < numberPaths; i++)
            {
                double min = S;
                double holder = S;
                for (double j = 0; j <= exerciseT; j += deltaT)
                {
                    // take a step with appropriate time change and start point
```

```
                    holder = path.PathGenerator(deltaT, holder, 1);

                    // keep track of minimum
                    if (holder < min)
                        min = holder;
                }

                pathCounter += holder - min;
            }

            // average and discount
            return Math.Exp(-r * exerciseT) * (pathCounter / numberPaths);
        }
    }

}
```

MCPaths.cs class

```
using System;
using MathNet.Numerics.Distributions;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HestonCalibrationAndPricing
{
     /// <summary>
     /// This class is used to create the Monte Carlo paths which will be used
         to price options within the Heston
     /// model in the class OptionsMC
     /// </summary>
    public class MCPaths
    {
        private double r;
        private double rho;
        private double kappaStar;
        private double thetaStar;
        private double sigma;
        private double v;

        public MCPaths(double r, double kappaStar, double thetaStar, double
            sigma, double rho, double v)
        {
            if (r <= 0 || sigma <= 0 || v <= 0)
            {
                throw new System.ArgumentException("r, sigma, v must be
                    positive");
            }

            this.r = r; this.rho = rho; this.kappaStar = kappaStar;
            this.thetaStar = thetaStar; this.sigma = sigma; this.v = v;
        }

         /// <summary>
         /// Returns a simulated future price for a risky asset within the
             Heston model.
         /// </summary>
         /// <param name = "T">The future time at which we wish to simulate the
              asset price.</param>
         /// <param name = "S">The initial asset price.</param>
         /// <param name = "numberTimeStepsPerPath">The number of steps we wish
              the scheme to take to reach time T.</param>
         /// <returns>Simulated asset price.</returns>
        public double PathGenerator(double T, double S, int
            numberTimeStepsPerPath)
```

```
        {
            if (T <= 0 || S <= 0 || numberTimeStepsPerPath <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            //sample from normal dist
            double[] x1 = new double[numberTimeStepsPerPath];
            Normal.Samples(x1, 0, 1);
            double[] x2 = new double[numberTimeStepsPerPath];
            Normal.Samples(x2, 0, 1);

            //set up parameters
            double tau = T / numberTimeStepsPerPath;
            double sqrtTau = Math.Sqrt(tau);
            double sqrtOneMinusRhoSquared = Math.Sqrt(1 - rho * rho);
            double alpha = (4 * kappaStar * thetaStar - sigma * sigma) / 8.0;
            double beta = -kappaStar / 2.0;
            double gamma = sigma / 2.0;

            // set up holder variables for Monte Carlo
            double y = Math.Sqrt(v);
            double s = S;

            // update holder variables iteratively according to MC scheme.
            for (int i = 0; i < numberTimeStepsPerPath; i++)
            {
                double deltaZ1 = sqrtTau * x1[i];
                double deltaZ2 = sqrtTau * (rho * x1[i] + sqrtOneMinusRhoSquared * x2[i]);
                s = s + r * s * tau + y * s * deltaZ1;
                double a = (y + gamma * deltaZ2) / (2 * (1 - beta * tau));
                y = a + Math.Sqrt(a * a + alpha * tau / (1 - beta * tau));
            }

            return s;
        }

         /// <summary>
         /// Returns a simulated future price for a risky asset within the Heston model using anithetic sampling with a view to reducing variance.
         /// </summary>
         /// <param name = "T">The future time at which we wish to simulate the asset price.</param>
         /// <param name = "S">The initial asset price.</param>
         /// <param name = "numberTimeStepsPerPath">The number of steps we wish the scheme to take to reach time T.</param>
         /// <returns>Simulated asset price.</returns>
        public double[] PathGeneratorAnithetic(double T, double S, int numberTimeStepsPerPath)
        {
            if (T <= 0 || S <= 0 || numberTimeStepsPerPath <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            // sample from normal dist
            double[] x1 = new double[numberTimeStepsPerPath];
            double[] x2 = new double[numberTimeStepsPerPath];
            Normal.Samples(x1, 0, 1);
            Normal.Samples(x2, 0, 1);

            //set up parameters
            double tau = T / numberTimeStepsPerPath;
```

```
            double sqrtTau = Math.Sqrt(tau);
            double sqrtOneMinusRhoSquared = Math.Sqrt(1 - rho * rho);
            double alpha = (4 * kappaStar * thetaStar - sigma * sigma) / 8.0;
            double beta = -kappaStar / 2.0;
            double gamma = sigma / 2.0;

            // set up holder variables, now two sets due to anithetic sampling
            double y = Math.Sqrt(v);
            double y1 = Math.Sqrt(v);
            double s = S;
            double s1 = S;

            // update both sets of holder variables iteratively, using the
                negative of normal sample for anitheic path
            for (int i = 0; i < numberTimeStepsPerPath; i++)
            {
                double deltaZ1 = sqrtTau * x1[i];
                double deltaZ2 = sqrtTau * (rho * x1[i] +
                    sqrtOneMinusRhoSquared * x2[i]);
                s = s + r * s * tau + y * s * deltaZ1;
                s1 = s1 + r * s1 * tau - y1 * s1 * deltaZ1;
                double a = (y + gamma * deltaZ2) / (2 * (1 - beta * tau));
                double aa = (y1 - gamma * deltaZ2) / (2 * (1 - beta * tau));
                y = a + Math.Sqrt(a * a + alpha * tau / (1 - beta * tau));
                y1 = aa + Math.Sqrt(aa * aa + alpha * tau / (1 - beta * tau));
            }

            // return array of end points of the two paths.
            double[] sArray = { s, s1 };
            return sArray;
        }

    }
}
```

Calibrator.cs class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace HestonCalibrationAndPricing
{

    public class CalibrationFailedException : Exception
    {
        public CalibrationFailedException()
        {
        }
        public CalibrationFailedException(string message)
            : base(message)
        {
        }
    }

    public struct MarketData
    {
        public double K;
        public double T;
        public double Price;
    }

    public enum CalibrationOutcome
    {
```

```
        NotStarted,
        FinishedOK,
        FailedMaxItReached,
        FailedOtherReason
    };

    /// <summary>
    /// This class is used to calibrate the parameters of the Heston Model to
        real world data
    /// </summary>
    public class Calibrator
    {
        private const double defaultAccuracy = 1.0e-15;
        private const int defaultMaxIts = 1000;
        private double accuracy;
        private int maxIts;

        private List<MarketData> marketList;
        private double r;
        private double S;

        private CalibrationOutcome outcome;

        private double[] calibratedParams;

        public Calibrator()
        {
            accuracy = defaultAccuracy;
            maxIts = defaultMaxIts;
            marketList = new List<MarketData>();
            r = 0;
            S = 0;
        }

        public Calibrator(double r, double S, int maxIts, double accuracy)
        {
            if (r <= 0 || S <= 0 || maxIts <= 0 )
            {
                throw new System.ArgumentException("Parameters must be positive
                    ");
            }

            this.accuracy = accuracy;
            this.maxIts = maxIts;
            marketList = new List<MarketData>();
            this.r = r;
            this.S = S;
        }

        /// <summary>
        /// Sets the parameters which will be used as a starting point by the
            calibrator
        /// </summary>
        public void SetGuessParameters(double kappaStar, double thetaStar,
            double sigma, double rho, double v)
        {
            if (sigma <= 0 || v <= 0)
            {
                throw new System.ArgumentException("Sigma, v must be positive")
                    ;
            }

            Options e = new Options(r, S, kappaStar, thetaStar, sigma, rho, v);
            calibratedParams = e.ParamsAsArray();
        }

        /// <summary>
```

```
        /// Adds the details of a real world option to the list marketList of data which will be used for calibration.
        /// </summary>
        /// <param name="K">Observed option's strike price.</param>
        /// <param name="T">Observed option's maturity time.</param>
        /// <param name="Price">Observed options price.</param>
        public void AddObservedOption(double K, double T, double Price)
        {
            if (K <= 0 || T <= 0 || Price <= 0)
            {
                throw new System.ArgumentException("Parameters must be positive");
            }

            MarketData observedOption;
            observedOption.K = K;
            observedOption.T = T;
            observedOption.Price = Price;
            marketList.Add(observedOption);
        }

        /// <summary>
        /// Calculates the mean squared error between the European call prices of an instance, options,
        /// of the class Options and the market prices found in marketList
        /// </summary>
        /// <param name="options">An instance of the class Options.</param>
        public double CalculateMeanSquaredErrorBetweenModelAndMarket(Options options)
        {
            double mse = 0;
            foreach (MarketData data in marketList)
            {
                double T = data.T;
                double K = data.K;
                double price = options.EuropeanCallPrice(T, K);
                double diff = price - data.Price;
                mse += diff * diff;
            }
            return mse;
        }

         /// <summary>
         /// This is the function which will be used by the calibrator
         /// </summary>
        public void CalibrationObjectiveFunction(double[] paramsarray, ref double func, object obj)
        {
            Options european = new Options(r, S, paramsarray);
            func = CalculateMeanSquaredErrorBetweenModelAndMarket(european);
        }

         /// <summary>
         /// Calibrates the model parameters to fit the market data as closely as possible
         /// </summary>
        public void Calibrate()
        {

            // set up for calibration
            outcome = CalibrationOutcome.NotStarted;
            double[] initialParams = new double[Options.numberParams];

            if (calibratedParams == null)
            {
                throw new System.Exception("Please add an initial guess for parameters");
```

```
            }

            calibratedParams.CopyTo(initialParams, 0);
            double epsg = accuracy;
            double epsf = accuracy;
            double epsx = accuracy;
            double diffstep = 1.0e-6;
            int maxits = maxIts;
            double stpmax = 0.05;

            alglib.minlbfgsstate state;
            alglib.minlbfgsreport rep;
            alglib.minlbfgscreatef(5, initialParams, diffstep, out state);
            alglib.minlbfgssetcond(state, epsg, epsf, epsx, maxits);
            alglib.minlbfgssetstpmax(state, stpmax);

            // calibrate and return outcome, error
            alglib.minlbfgsoptimize(state, CalibrationObjectiveFunction, null, null);
            double[] resultParams = new double[Options.numberParams];
            alglib.minlbfgsresults(state, out resultParams, out rep);

            System.Console.WriteLine("Termination type: {0}", rep.terminationtype);
            System.Console.WriteLine("Num iterations {0}", rep.iterationscount);
            System.Console.WriteLine("{0}", alglib.ap.format(resultParams, 5));

            if (rep.terminationtype == 1
                || rep.terminationtype == 2
                || rep.terminationtype == 4)
            {
                outcome = CalibrationOutcome.FinishedOK;
                calibratedParams = resultParams;
            }
            else if (rep.terminationtype == 5)
            {
                outcome = CalibrationOutcome.FailedMaxItReached;
                calibratedParams = resultParams;

            }
            else
            {
                outcome = CalibrationOutcome.FailedOtherReason;
                throw new CalibrationFailedException("Heston model calibration failed badly.");
            }
        }

         /// <summary>
         /// Obtains the calibration status of the model, as well as the models pricing error.
         /// </summary>
        public void GetCalibrationStatus(ref CalibrationOutcome calibOutcome, ref double pricingError)
        {
            calibOutcome = outcome;
            Options m = new Options(r, S, calibratedParams);
            pricingError = CalculateMeanSquaredErrorBetweenModelAndMarket(m);
        }

         /// <summary>
         /// Creates an instance of the class Options with the calibrated parameters.
         /// </summary>
         /// <returns>Calibrated model.</returns>
        public Options GetCalibratedModel()
```

```
        {
            Options m = new Options(r, S, calibratedParams);
            return m;
        }

    }
}
```

InterfaceFill.cs class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HestonModel.Interfaces;

namespace HestonModel.InterfaceImplement
{
    /// <summary>
    /// This class is used to implement several interfaces.
    /// </summary>
    public class InterfaceFill : IOption, IVarianceProcessParameters,
        IMonteCarloSettings, IAsianOption, IEuropeanOption,
        ICalibrationSettings
    {
        double T;
        double kappa;
        double theta;
        double sigma;
        double v;
        double rho;
        int numberTrials;
        int numberTimeSteps;
        PayoffType p;
        IEnumerable<double> timeList;
        double K;
        double accuracy;



        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)
        {
            this.T = T; this.kappa = kappa; this.theta = theta; this.sigma =
                sigma;
            this.v = v; this.rho = rho; this.numberTrials = numberTrials; this.
                numberTimeSteps = numberTimeSteps;
            this.p = p; timeList = timeArray.AsEnumerable(); this.K = K;
            this.accuracy = accuracy;
        }



        double IOption.Maturity => T;

        double IVarianceProcessParameters.Kappa => kappa;

        double IVarianceProcessParameters.Theta => theta;

        double IVarianceProcessParameters.Sigma => sigma;

        double IVarianceProcessParameters.V0 => v;

        double IVarianceProcessParameters.Rho => rho;

```

```
        int IMonteCarloSettings.NumberOfTrials => numberTrials;

        int IMonteCarloSettings.NumberOfTimeSteps => numberTimeSteps;

        IEnumerable<double> IAsianOption.MonitoringTimes => timeList;

        PayoffType IEuropeanOption.Type => p;

        double IEuropeanOption.StrikePrice => K;

        double ICalibrationSettings.Accuracy => accuracy;

        int ICalibrationSettings.MaximumNumberOfIterations => 1000;
    }
}
```

OtherInterfaceFill.cs class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HestonModel.Interfaces;

namespace HestonModel.InterfaceImplement
{
    /// <summary>
    /// This class is used to implement the interfaces IHestonModelParameters
        and ICalibrationResult
    /// </summary>
    public class OtherInterfaceFill : IHestonModelParameters,
        ICalibrationResult
    {
        double S;
        double r;
        IVarianceProcessParameters paramss;
        CalibrationOutcome c;
        double error;

        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)
        {
            this.S = S; this.r = r;
            double[] TT = { 0 };
            InterfaceFill fill = new InterfaceFill(T, kappa, theta, sigma, rho,
                 v, numberTrials, numberTimeSteps, 0, TT, 100, 0);
            paramss = fill;
            this.c = c; this.error = error;
        }

        double IHestonModelParameters.InitialStockPrice => S;

        double IHestonModelParameters.RiskFreeRate => r;

        IVarianceProcessParameters IHestonModelParameters.VarianceParameters =>
             paramss;

        CalibrationOutcome ICalibrationResult.MinimizerStatus => c;

        double ICalibrationResult.PricingError => error;

    }
}
```

AnotherInterfaceFill.cs class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HestonModel.Interfaces;

namespace HestonModel.InterfaceImplement
{
    /// <summary>
    /// This class is used to implement the interface IHestonCalibrationResult.
    /// </summary>
    public class AnotherInterfaceFill : IHestonCalibrationResult
    {
        IHestonModelParameters paramms;
        double error;
        CalibrationOutcome c;

        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)
        {
            this.c = c; this.error = error;
            OtherInterfaceFill fill = new OtherInterfaceFill(T, kappa, theta,
                sigma, rho, v, numberTrials, numberTimeSteps, S, r, c, error);
            paramms = fill;
        }

        IHestonModelParameters IHestonCalibrationResult.Parameters => paramms;

        CalibrationOutcome ICalibrationResult.MinimizerStatus => c;

        double ICalibrationResult.PricingError => error;
    }
}
```

FinalInterfaceFill.cs class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HestonModel.Interfaces;

namespace HestonModel.InterfaceImplement
{
    /// <summary>
    /// This class is used to implement the interface IOptionMarketData when it
         takes IEuropeanOption.
    /// </summary>
    public class FinalInterfaceFill : IOptionMarketData<IEuropeanOption>
    {
        double price;
        IEuropeanOption option;

        public FinalInterfaceFill(double price, double T, PayoffType p, double
            K)
        {
            this.price = price;
            double[] TT = { 0 };
            InterfaceFill fill = new InterfaceFill(T, 0, 0, 0, 0, 0, 0, 0, p,
                TT, K, 0);
```

```
            option = fill;
        }

        IEuropeanOption IOptionMarketData<IEuropeanOption>.Option => option;

        double IOptionMarketData<IEuropeanOption>.Price => price;
    }
}
```

Heston.cs

```
using System;
using System.Collections.Generic;
using HestonModel.Interfaces;
using HestonCalibrationAndPricing;
using System.Linq;
using HestonModel.InterfaceImplement;

namespace HestonModel
{

    /// <summary>
    /// This class will be used for grading.
    /// Don't remove any of the methods and don't modify their signatures. Don't change the namespace.
    /// Your code should be implemented in other classes (or even projects if you wish), and the relevant functionality should only be called here and outputs returned.
    /// You don't need to implement the interfaces that have been provided if you don't want to.
    /// </summary>
    public static class Heston
    {
        /// <summary>
        /// Method for calibrating the heston model.
        /// </summary>
        /// <param name="guessModelParameters">Object implementing IHestonModelParameters interface containing the risk-free rate, initial stock price
        /// and initial guess parameters to be used in the calibration.</param>
        /// <param name="referenceData">A collection of objects implementing IOptionMarketData<IEuropeanOption> interface. These should contain the reference data used for calibration.</param>
        /// <param name="calibrationSettings">An object implementing ICalibrationSettings interface.</param>
        /// <returns>Object implementing IHestonCalibrationResult interface which contains calibrated model parameters and additional diagnostic information</returns>
        public static IHestonCalibrationResult CalibrateHestonParameters(IHestonModelParameters guessModelParameters, IEnumerable<IOptionMarketData<IEuropeanOption>> referenceData, ICalibrationSettings calibrationSettings)
        {
            // set up calibrator
            Calibrator cal = new Calibrator(guessModelParameters.RiskFreeRate, guessModelParameters.InitialStockPrice, calibrationSettings.MaximumNumberOfIterations, calibrationSettings.Accuracy);
            cal.SetGuessParameters(guessModelParameters.VarianceParameters.Kappa, guessModelParameters.VarianceParameters.Theta,
                guessModelParameters.VarianceParameters.Sigma, guessModelParameters.VarianceParameters.Rho, guessModelParameters.VarianceParameters.V0);

            // add market data
            foreach (IOptionMarketData<IEuropeanOption> data in referenceData)
            {
```

```
                cal.AddObservedOption(data.Option.StrikePrice, data.Option.Maturity, data.Price);
            }

            // calibrate, get error, outcome, parameters
            cal.Calibrate();
            double error = 0;
            HestonCalibrationAndPricing.CalibrationOutcome outcome = HestonCalibrationAndPricing.CalibrationOutcome.NotStarted;
            cal.GetCalibrationStatus(ref outcome, ref error);

            Options e = cal.GetCalibratedModel();
            double[] paramArray = e.ParamsAsArray();
            CalibrationOutcome outcome1 = (CalibrationOutcome)outcome;

            // implement and return IHestonCalibrationResult
            AnotherInterfaceFill fill = new AnotherInterfaceFill(0, paramArray[Options.kappaIndex], paramArray[Options.thetaIndex], paramArray[Options.sigmaIndex], paramArray[Options.rhoIndex], paramArray[Options.vIndex], 0, 0, 0, 0, outcome1, error);
            return fill;
        }

        /// <summary>
        /// Price a European option in the Heston model using the Heston formula. This should be accurate to 5 decimal places
        /// </summary>
        /// <param name="parameters">Object implementing IHestonModelParameters interface, containing model parameters.</param>
        /// <param name="europeanOption">Object implementing IEuropeanOption interface, containing the option parameters.</param>
        /// <returns>Option price</returns>
        public static double HestonEuropeanOptionPrice(IHestonModelParameters parameters, IEuropeanOption europeanOption)
        {
            Options eur = new Options(parameters.RiskFreeRate, parameters.InitialStockPrice,
               parameters.VarianceParameters.Kappa, parameters.VarianceParameters.Theta, parameters.VarianceParameters.Sigma,
               parameters.VarianceParameters.Rho, parameters.VarianceParameters.V0);

            if (europeanOption.Type == 0)
            {
                return eur.EuropeanCallPrice(europeanOption.Maturity, europeanOption.StrikePrice);
            }
            else
                return eur.EuropeanPutPrice(europeanOption.Maturity, europeanOption.StrikePrice);
        }

        /// <summary>
        /// Price a European option in the Heston model using the Monte-Carlo method. Accuracy will depend on number of time steps and samples
        /// </summary>
        /// <param name="parameters">Object implementing IHestonModelParameters interface, containing model parameters.</param>
        /// <param name="europeanOption">Object implementing IEuropeanOption interface, containing the option parameters.</param>
        /// <param name="monteCarloSimulationSettings">An object implementing IMonteCarloSettings object and containing simulation settings.</param>
        /// <returns>Option price</returns>
        public static double HestonEuropeanOptionPriceMC(IHestonModelParameters parameters, IEuropeanOption europeanOption, IMonteCarloSettings
```

```
            monteCarloSimulationSettings)
        {
            OptionsMC option = new OptionsMC(parameters.RiskFreeRate,
                europeanOption.StrikePrice, parameters.VarianceParameters.Kappa
                ,
                parameters.VarianceParameters.Theta, parameters.
                    VarianceParameters.Sigma, parameters.VarianceParameters.Rho
                    ,
                parameters.VarianceParameters.V0, parameters.InitialStockPrice)
                    ;

            if(europeanOption.Type == 0)
            {
                return option.EuropeanCallOptionPriceMCAnitheticParallel(
                    europeanOption.Maturity, monteCarloSimulationSettings.
                    NumberOfTimeSteps, monteCarloSimulationSettings.
                    NumberOfTrials);
            }
            else
                return option.EuropeanPutOptionPriceMCAnitheticParallel(
                    europeanOption.Maturity, monteCarloSimulationSettings.
                    NumberOfTimeSteps, monteCarloSimulationSettings.
                    NumberOfTrials);
        }

        /// <summary>
        /// Price a Asian option in the Heston model using the
        /// Monte-Carlo method. Accuracy will depend on number of time steps
            and samples</summary>
        /// <param name="parameters">Object implementing IHestonModelParameters
             interface, containing model parameters.</param>
        /// <param name="asianOption">Object implementing IAsian interface,
            containing the option parameters.</param>
        /// <param name="monteCarloSimulationSettings">An object implementing
            IMonteCarloSettings object and containing simulation settings.</
            param>
        /// <returns>Option price</returns>
        public static double HestonAsianOptionPriceMC(IHestonModelParameters
            parameters, IAsianOption asianOption, IMonteCarloSettings
            monteCarloSimulationSettings)
        {
            OptionsMC asian = new OptionsMC(parameters.RiskFreeRate,
                asianOption.StrikePrice, parameters.VarianceParameters.Kappa,
               parameters.VarianceParameters.Theta, parameters.
                   VarianceParameters.Sigma, parameters.VarianceParameters.Rho,
               parameters.VarianceParameters.V0, parameters.InitialStockPrice);

            if (asianOption.Type == 0)
            {
                return asian.PriceAsianCallMC(asianOption.MonitoringTimes.
                    ToArray(), asianOption.Maturity,
                monteCarloSimulationSettings.NumberOfTrials,
                    monteCarloSimulationSettings.NumberOfTimeSteps);
            }
            else
                return asian.PriceAsianPutMC(asianOption.MonitoringTimes.
                    ToArray(), asianOption.Maturity,
                monteCarloSimulationSettings.NumberOfTrials,
                    monteCarloSimulationSettings.NumberOfTimeSteps);
        }

        /// <summary>
        /// Price a lookback option in the Heston model using the
        /// a Monte-Carlo method. Accuracy will depend on number of time steps
            and samples </summary>
        /// <param name="parameters">Object implementing IHestonModelParameters
             interface, containing model parameters.</param>
```

```
        /// <param name="maturity">An object implementing IOption interface and containing option's maturity</param>
        /// <param name="monteCarloSimulationSettings">An object implementing IMonteCarloSettings object and containing simulation settings.</param>
        /// <returns>Option price</returns>
        public static double HestonLookbackOptionPriceMC(IHestonModelParameters parameters, IOption maturity, IMonteCarloSettings monteCarloSimulationSettings)
        {
            OptionsMC lookback = new OptionsMC(parameters.RiskFreeRate, parameters.VarianceParameters.Kappa,
                        parameters.VarianceParameters.Theta, parameters.VarianceParameters.Sigma, parameters.VarianceParameters.Rho,
                        parameters.VarianceParameters.V0, parameters.InitialStockPrice);

            return lookback.PriceLookbackCallMC(maturity.Maturity, monteCarloSimulationSettings.NumberOfTrials, monteCarloSimulationSettings.NumberOfTimeSteps);
        }
    }
}
```