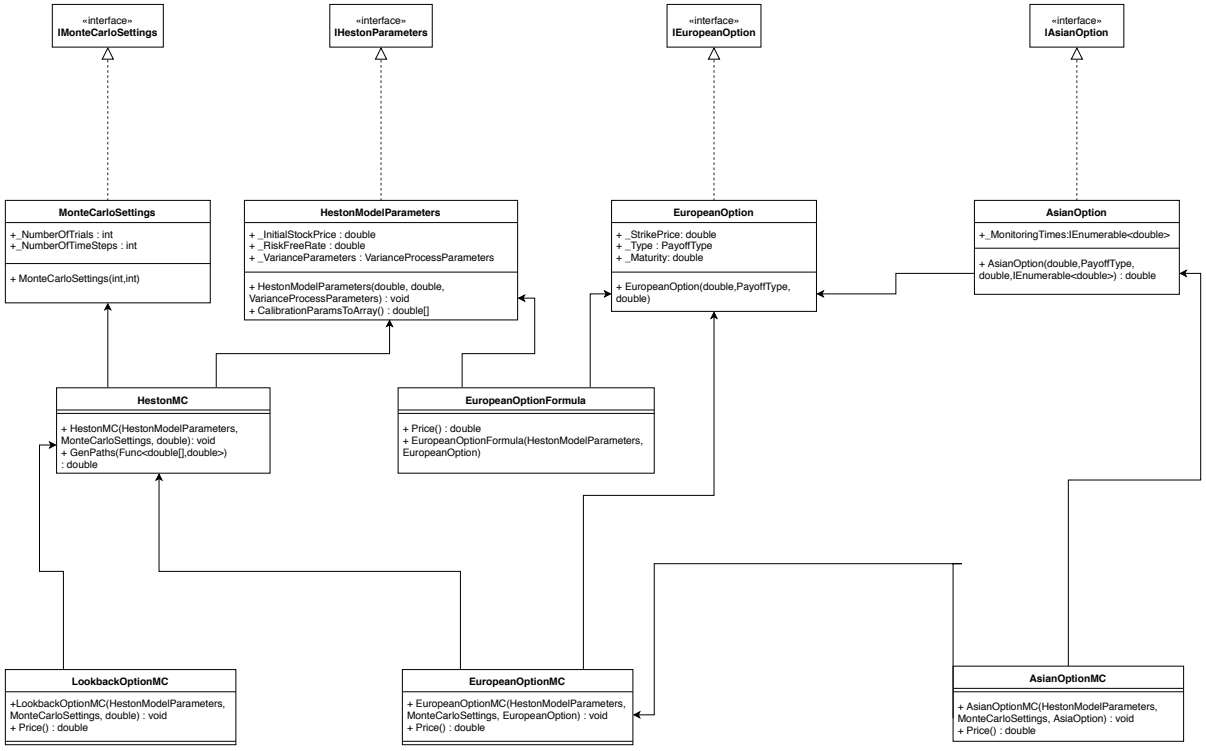
Object Oriented Programming with Applications

Kostas Mylonakis - s1824954

January 13, 2019

1 Class Diagram

The following diagram represents the main classes used in for pricing options. The bold arrows denotes inheritance relationship between classes, while the dashed arrows denotes the class implementation of an interface.



The class design adopted is almost identical with the one the given interfaces dictate. The reason behind this decision is that the inheritance relationships implied by the interfaces were very logical. The basis for all options is the **HestonModelParameters** class, which holds all parameters and methods of Heston Model and implements **IHestonParameters** interface. This way all functionality added to that class immediately applies for all options that are priced in Heston model, so we can avoid code duplication and change the functionality in one place when needed.

All the options that are priced using Monte Carlo method are derived by the HestonMC class which is derived by the HestonModelParameters. This design is efficient because the MC path generation procedure is the same for all options, therefore can be shared. Only payoff functions vary from option to option and this function can be passed as an argument to the path generation function of HestonMC class. The virtue of this design decision was notably clear when the I changed the Monte Carlo method

to work in parallel. The code changed only inside that class and all the option types worked correctly at once.

2 System Specification

The experiments presented in the rest of the report were run on a system with the following specifications:

- quad-core Intel Core i7 processor @ 2.2GHz
- 6MB L3 Cache
- 16GB of 1600MHz DDR3L

3 European Type Options

Heston Formula

For the implementation of the class for pricing European options using Heston Formula, section 2.4 of [1] is followed. Final part of those calculations consists of two integral calculations from 0 to infinity. It is straightforward to see that the lower bound should be something close to zero, like 1e-6, but not zero, because the integrad there is infinite. For the selection of the upper bound one have to choose a number where the integrad has decayed significantly so that the error is not large, but still a small number for efficiency. For that purpose I tried to gain some intuition by using MATLAB.

The integrad is a function of ten variables, namely ϕ, volatility, risk free rate, $\sigma, \rho, \kappa, \theta$, strike price, initial stock price and maturity, while the integral that we are interested in is over ϕ. The following figures illustrate the largest value of ϕ for which the integrad function gets value larger than a specific number, in our case 1e-3 and 1e-5. Consequently, if we choose our upper bound for our integration like this we can expect an analogous error.

Since we are not able to illustrate the whole parameter space, I have plotted the subspace of pairs of parameters in order to have a rough estimation of an appropriate upper bound.

We can see that regarding only the strike price and the initial stock price an upper bound near 20 would suffice for accuracy equal to 1e-3, while for accuracy 1e-5 we would need to integrate till $\phi = 60$.

We can see that regarding only the κ and θ an upper bound near 60 would suffice for accuracy equal to 1e-3, while for accuracy 1e-5 we would need to integrate till $\phi = 200$.

We can see that regarding only the ρ and risk free rate an upper bound near 1000 would suffice for accuracy equal to 1e-3, while for accuracy 1e-5 we would need to integrate till $\phi = 2000$.

In conclusion, a safe choice for accuracy and efficiency seems to be 1000, which is used for the results presented in this section.

In Table 1 are presented the prices for the European options of task 2.2 using Heston Formula.

Strike K	Option Exercise T	Price
100.0	1.0	7.274
100.0	2.0	11.74
100.0	3.0	15.48
100.0	4.0	18.77
100.0	15.0	43.17

Table 1: Prices of Call options in Heston Model using Heston Formula with parameters $r = 2.5\%$, $\theta = 3.98\%$, $\kappa = 157.68\%$, $\sigma = 57.51\%$, $\rho = -57.11\%$, $v = 1.75\%$, $S = 100$.

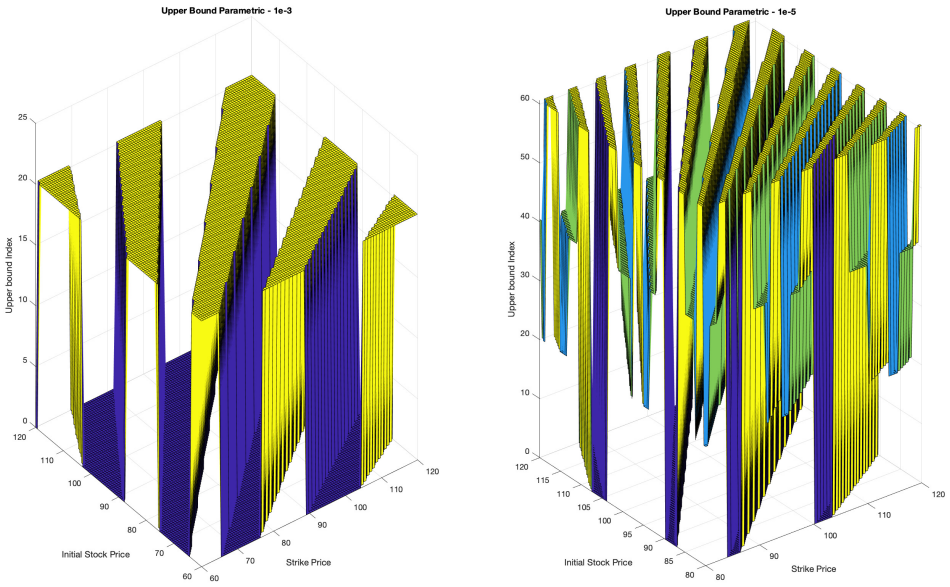


Figure 1: Upper Bound for fixed $\sigma, \rho, \kappa, \theta$ and maturity. Strike and Initial Stock price in range [60, 120].

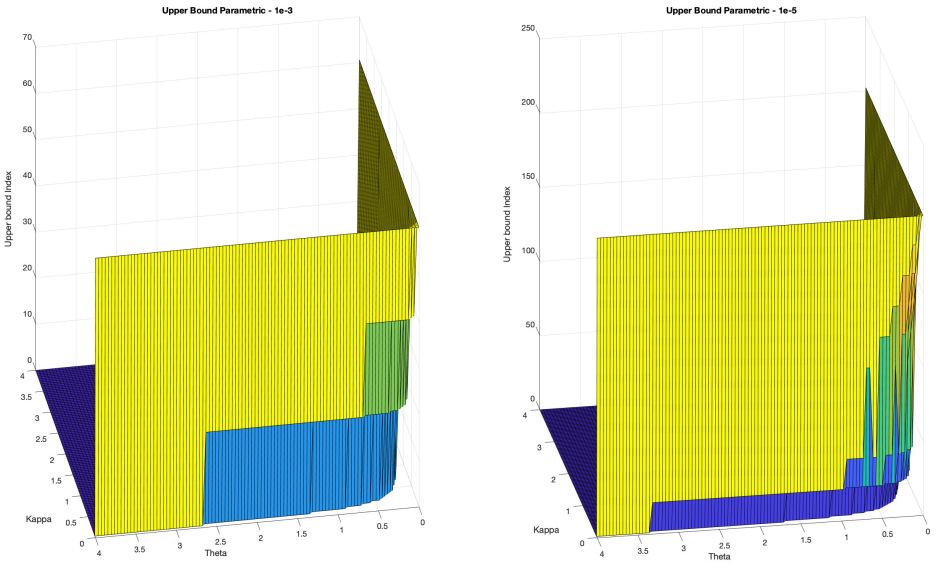


Figure 2: Upper Bound for fixed volatility, risk free rate,σ, ρ, strike price, initial stock price and maturity. κ and θ in range [0, 4].

The correctness of this implementation is check against MATLAB 2018 build-in function *optByHestonNI* which is part of Financial Toolbox.

Monte Carlo Pricing

In Table 2 are presented the prices for the European options of task 2.3 using Monte Carlo method.

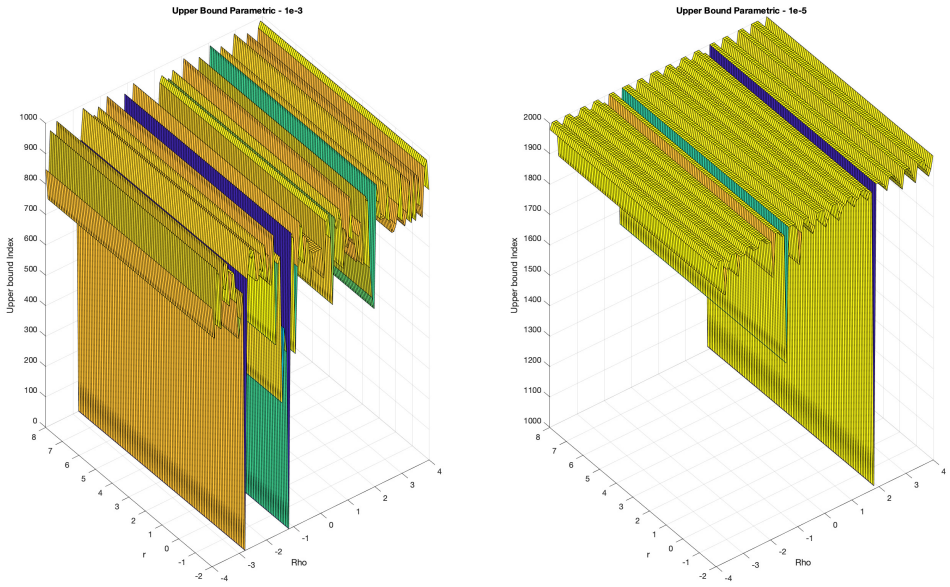


Figure 3: Upper Bound for fixed volatility, σ, κ, θ, strike price, initial stock price and maturity. Risk free rate in range [-2, 8] and ρ in range [-4, 4].

Strike K	Option Exercise T	Price
100.0	1.0	13.6
100.0	2.0	22.3
100.0	3.0	30.1
100.0	4.0	36.4
100.0	15.0	78.3

Table 2: Prices of European Call options in Heston Model using Monte Carlo Method with parameters $r = 10\%$, $\theta = 6\%$, $\kappa = 200\%$, $\sigma = 40\%$, $\rho = 50\%$, $v = 4\%$, $S = 100$. For the Monte Carlo simulation, 10^5 different paths was generated with 365 time steps per year.

Validation

In order to validate the correctness of the implementations of both methods, Heston formula and Monte Carlo, I price the options of task 2.2 with both methods and check the discrepancy between them. The results are presented in Table 3.

Strike K	Option Exercise T	Price	Reference Price	Relative Error
100.0	1.0	13.67	13.63	0.0029061
100.0	2.0	22.375	22.453	0.0034817
100.0	3.0	30.195	29.996	0.0066435
100.0	4.0	36.44	36.655	0.0058662
100.0	15.0	78.394	78.431	0.00046845

Table 3: Comparison between prices calculated using MC method against using Heston Formula. Heston parameters used in this experiment are $r = 10\%$, $\theta = 6\%$, $\kappa = 200\%$, $\sigma = 40\%$, $\rho = 50\%$, $v = 4\%$, $S = 100$. For the Monte Carlo simulation, 10^5 different paths was generated with 365 time steps per year.

Of course an other important aspect is the settings of the Monte Carlo simulation used and specifically

the way the number of trials and the number of time steps affect convergence.

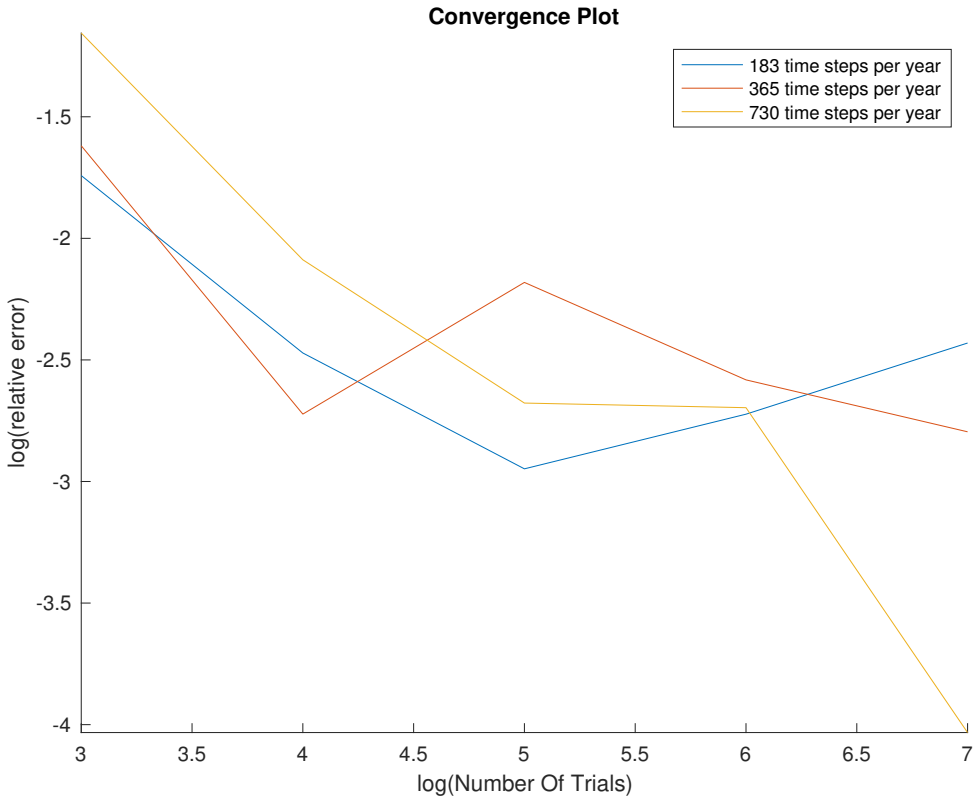


Figure 4: Relative error between prices using Heston formula and Monte Carlo for different number of trials, form 10^3 to 10^7 and for different time steps per year.

We can see that generally the error is reduced when we increase the number of trials, which is what we were expecting. For larger than 1 million trials also the number of time steps per year seems to have the same behaviour as expected, that is that the larger the time steps the smaller the error.

4 Heston Model Calibration

In order to set the model parameters a common practice is to calibrate it such that the price of some options match the observed prices of them in the market. This is essentially a minimization problem and for my implementation I have utilized BFGS algorithm form Alglib package. This function iteratively search for a set of parameters such that an user-defined error function satisfies one of the termination conditions given the desired accuracy. In my case this error function is the mean square error between the prices the model in the current iteration produces and the observed market prices. To limit the minimization time there is a limit of iterations that when is exceeded the algorithm finishes and return the last iteration result.

Therefore the possible minimization outcomes are:

- 1 - Relative function improvement is no more that desired accuracy
- 2 - Relative step is no more than desired accuracy
- 4 - Gradient norm is no more than desired accuracy
- 5 - Max iteration steps was taken

For the calibration of the model using the market data given in task 2.5, I have chosen as guess parameters $\kappa = 1.5768,\ \theta = 0.398,\ \sigma = 0.5751,\ v_0 = 1.0175,\ \rho = -0.5711$, the initial stock price equal to 100 , the risk-free-rate at 2.5% and $accuracy = 1e^{-3}$ and maximum iterations 1000. For this setup the calibrator finishes after 39 iterations, with result code 2, meaning the relative step is less than the accuracy set. The mean square error between the observed market prices and the model prices using the calibration result is 0.12008. This means that the algorithm was trapped in a local min.

Validation

To check the correctness of the calibration code I followed the following steps:

- Select a set of Heston parameters
- Price 5 or more options using those parameters
- Pick a set of parameters as guess parameters
- Try to calibrate model starting from the guess and see if I get the initial parameters used to price the options.

Therefore as an experiment I choose to price the options of Table 3 of task2.5 by using $\kappa = 1.5768,\ \theta = 0.398,\ \sigma = 0.5751,\ v_0 = 0.0175,\ \rho = -0.5711$, the initial stock price equal to 100 , the risk-free-rate at 2.5%

In Table 4 are presented some experiments to verify the correctness of the calibrator.

Guess	Accuracy	Iterations	Mean Square Error
$\kappa = 1.59,\ \theta = 0.48,\ \sigma = 0.599,\ v_0 = 0.03,\ \rho = -0.55$	$1e-3$	6	0.0002
$\kappa = 2.0,\ \theta = 0.88,\ \sigma = 2.599,\ v_0 = 1.3,\ \rho = 0.55$	$1e-3$	58	0.1078
$\kappa = 2.0,\ \theta = 0.88,\ \sigma = 2.599,\ v_0 = 1.3,\ \rho = 0.55$	$1e-5$	66	0.005845

Table 4: Experiments to validate calibrator. Guess stands for the initial guess parameters that are provided to the minimizer, accuracy and iterations are the minimization termination parameters and mean square error is the error between the observed market prices and the prices using the parameters of the calibration result.

For the first experiment I have selected an initial guess near the correct parameters and the calibrator after 6 iterations finishes and returns parameters very close to the correct ones, producing a mean square error of order $e-4$. For the second experiment I have selected a guess far from the correct answer and as we can see the calibrator was trapped in a local minimum. Finally, at the last experiment I have used again the distant guess but now I set a larger accuracy. At this case I just found an answer that decreases further the error but again it is a local minimum. The only case where the calibrator does not converge is when one of the parameter conditions is violated (negative volatility most frequently) so I throw an exception and terminate the calibration. I could handle this exception differently, for example I could give a valid value and let the minimization continue.

5 Asian Type Options

In Table 5 are presented the prices for the Asian options of task 2.7 using Monte Carlo method.

6 Lookback Type Options

In Table 6 are presented the prices for the Lookback options of task 2.8 using Monte Carlo method.

Strike K	Option Exercise T	$T_1, \cdots, T_M$	Price
100.0	1.0	0.75,1.00	11.99
100.0	2.0	0.25,0.50,0.75,1.00,1.25,1.50,1.75	11.45
100.0	3.0	1.00,2.00,3.00	19.7

Table 5: Prices of Asian Call options in Heston Model using Monte Carlo Method with parameters $r = 10\%$, $\theta = 6\%$, $\kappa = 200\%$, $\sigma = 40\%$, $\rho = 50\%$, $v = 4\%$, $S = 100$. For the Monte Carlo simulation, 10^5 different paths was generated with 365 time steps per year.

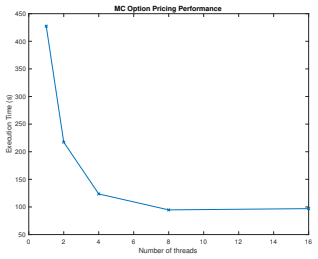
Option Exercise T	Price
1.0	19.08
3.0	37.61
5.0	50.15
7.0	60.48
9.0	68.17

Table 6: Prices of Lookback Call options in Heston Model using Monte Carlo Method with parameters $r = 10\%$, $\theta = 6\%$, $\kappa = 200\%$, $\sigma = 40\%$, $\rho = 50\%$, $v = 4\%$, $S = 100$. For the Monte Carlo simulation, 10^5 different paths was generated with 365 time steps per year.

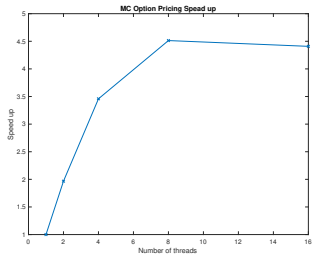
7 Parallel Monte Carlo Method

To accelerate the pricing of options I implemented a parallel version of the function that generates different paths for the Monte Carlo simulation. The code is implemented as follows:

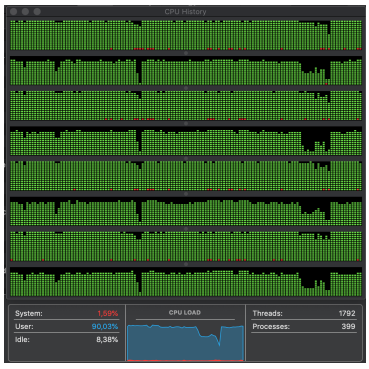
First a specific number of threads are created. Each thread has its own state (i.e memory) which is essential in order to avoid the use of mutex. The trials are divided equally among the threads and all threads start to do calculations concurrently. Each thread accumulates the results of its own trials and when all threads are finished all the thread sums are accumulated for the final result.



(a) Execution Time



(b) Speed Up



(c) CPU usage during experiment with 8 threads over 90%.

Figure 5: Performance of parallel Monte Carlo pricing code for different number of threads.

The performance seems to have a almost linear behaviour with the threads used till the 4 threads. This is expected, since the machine used for the experiments (see section 2) has 4 cores and the avoidance of mutex makes the threads to work fully parallel. Finally, due to hyperthreading the optimal number of threads for the machine used is 8.

Documentation

For documentation purposes I have used *Doxygen*. This open-source documentation tool can automatically create documentation for a project using the xml comments in the code. The documentation for my code is online under the following link `https://mylonakk.github.io/hestonModel/html/annotated.html`.

References

[1] David Siska. A note on the heston model. *OOPA course website*.

Appendix A - C# Code

AsianOption.cs

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

namespace HestonModel.Classes.InterfaceClasses
{
    /// <summary>
    /// Class Implementing Asian Option Interface.
    /// </summary>
    public class AsianOption : EuropeanOption, IAsianOption
    {
        // Properties
        private IEnumerable<double> _MonitoringTimes;

        // Interface implementation
        public IEnumerable<double> MonitoringTimes
        {
            get { return _MonitoringTimes; }
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.InterfaceClasses.AsianOption"/> class.
        /// </summary>
        /// <param name="StrikePrice">Strike price.</param>
        /// <param name="Type">Type (Put / Call)</param>
        /// <param name="Maturity">Maturity</param>
        /// <param name="MonitoringTimes">Monitoring times</param>
        public AsianOption(double StrikePrice, PayoffType Type,
                           double Maturity, IEnumerable<double>
                               MonitoringTimes)
            : base(StrikePrice, Type, Maturity)
        {
            // Sanity Check
            if (MonitoringTimes.Any())
            {
```

```
                double max = MonitoringTimes.Max();

                if (max > Maturity)
                {
                    throw new Exception("All monitoring time have to be
                        less of equal than the maturity time");
                }

                double min = MonitoringTimes.Min();
                if (min < 0)
                {
                    throw new Exception("All monitoring time have to be
                        nonnegative");
                }
            }
            else
            {
                throw new Exception("Provide at least one monitoring time
                    for the Asian option pricing");
            }

            _MonitoringTimes = MonitoringTimes;
        }
    }
}
```

AsianOptionMC.cs

```
  using System;
using HestonModel.Interfaces;
using System.Collections.Generic;
using HestonModel.Classes.InterfaceClasses;

namespace HestonModel.Classes
{
    /// <summary>
    /// Class to represent an Asian option which is priced using Monte
        Carlo
    ///  method in Heston model. This class inherits the HestonMC class and
         the
    ///  IAsianOption interface.
    /// </summary>
    public class AsianOptionMC : EuropeanOptionMC, IAsianOption
    {
        // Properties
        private IEnumerable<double> _MonitoringTimes;

        // Interface implementation
        public IEnumerable<double> MonitoringTimes
        {
            get { return _MonitoringTimes; }
        }

        /// <summary>
        /// Gets the monitored avg.
        /// </summary>
```

```
/// <returns>The monitored avg.</returns>
/// <param name="S">S.</param>
private double GetMonitoredAvg(double[] S)
{
    double tau = Maturity / NumberOfTimeSteps;
    double sum = 0;
    int count = 0;
    foreach (double t in MonitoringTimes)
    {
        sum += S[(int)Math.Floor(t / tau) - 1];
        count++;
    }
    return sum / count;
}

/// <summary>
/// Calls the payoff.
/// </summary>
/// <returns>The payoff.</returns>
/// <param name="S">S.</param>
private double CallPayoff(double[] S)
{
    return Math.Max(GetMonitoredAvg(S) - StrikePrice, 0);
}

/// <summary>
/// Puts the payoff.
/// </summary>
/// <returns>The payoff.</returns>
/// <param name="S">S.</param>
private double PutPayoff(double[] S)
{
    return Math.Max(0, GetMonitoredAvg(S) - StrikePrice);
}

/// <summary>
/// Initializes a new instance of the EuropeanOptionMC class.
/// </summary>
/// <param name="parameters">Interface holding Heston Model params
    .</param>
/// <param name="monteCarloSimulationSettings">Interface holding
    Monte carlo simulation settings.</param>
/// <param name="asianOption">Interface holding asian option.</
    param>
public AsianOptionMC(HestonModelParameters parameters,
                     MonteCarloSettings
                         monteCarloSimulationSettings,
                     AsianOption asianOption)
    : base(parameters, monteCarloSimulationSettings, asianOption)
{
    _MonitoringTimes = asianOption.MonitoringTimes;
}

/// <summary>
/// Price this instance.
/// </summary>
```

```
        /// <returns>The price.</returns>
        public new double Price(int workers)
        {
            if (Type == PayoffType.Call)
            {
                return GenPaths(CallPayoff, workers);
            }
            else
            {
                return GenPaths(PutPayoff, workers);
            }
        }
    }
}
```

CalibrationSetting.cs

```
  using System;
using HestonModel.Interfaces;

namespace HestonModel.Classes.InterfaceClasses
{
    /// <summary>
    /// Class representing calibration setting for Heston Model.
    /// </summary>
    public class CalibrationSettings : ICalibrationSettings
    {
        // Private Properties
        private double _Accuracy;
        private int _MaximumNumberOfIterations;

        // Interface Implementation
        public double Accuracy
        {
            get { return _Accuracy; }
        }
        public int MaximumNumberOfIterations
        {
            get { return _MaximumNumberOfIterations; }
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.InterfaceClasses.CalibrationSettings"/> class.
        /// </summary>
        /// <param name="accuracy">Accuracy.</param>
        /// <param name="maxIter">Max iterations</param>
        public CalibrationSettings(double accuracy, int maxIter)
        {
            // Sanity Check
            if (accuracy <= 0)
            {
                throw new Exception("The accuracy for the calibrator must
                    be positive.");
            }
            if (maxIter <= 0)
```

```
            {
                throw new Exception("The max number of iterations for the
                    calibrator must be a positive number.");
            }

            _Accuracy = accuracy;
            _MaximumNumberOfIterations = maxIter;
        }
    }
}
```

EuropeanOption.cs

```
  using System;
using HestonModel.Interfaces;

namespace HestonModel.Classes.InterfaceClasses
{
    public class EuropeanOption : IEuropeanOption
    {
        // Properties
        private double _StrikePrice;
        private PayoffType _Type;
        private double _Maturity;

        // Interface Implementation
        public double StrikePrice
        {
            get { return _StrikePrice; }
        }

        public PayoffType Type
        {
            get { return _Type; }
        }

        public double Maturity
        {
            get { return _Maturity; }
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.InterfaceClasses.EuropeanOption"/> class.
        /// </summary>
        /// <param name="StrikePrice">Strike price.</param>
        /// <param name="Type">Option Type (Put/Call).</param>
        /// <param name="Maturity">Maturity.</param>
        public EuropeanOption(double StrikePrice, PayoffType Type, double
            Maturity)
        {
            // Sanity Check
            if (Maturity <= 0)
            {
                throw new Exception("Maturity must be a positive number.");
            }
```

```
            if (StrikePrice <= 0)
            {
                throw new Exception("Strike price must be a positive number
                    .");
            }

            // Initialize private vars
            _StrikePrice = StrikePrice;
            _Type = Type;
            _Maturity = Maturity;
        }
    }
}
```

EuropeanOptionFormula.cs

```
  using System;
using System.Numerics;
using HestonModel.Interfaces;
using HestonModel.Classes.InterfaceClasses;
using MathNet.Numerics.Integration;

namespace HestonModel.Classes
{
    /// <summary>
    /// Class to represent an European Option in Heston model. This class
    /// inherits the HestonOption class and the IEuropeanOption interface.
    /// </summary>
    /// Todo: add exceptions, better documentation, put pricing"
    public class EuropeanOptionFormula : HestonModelParameters,
        IEuropeanOption
    {
        // Properties
        private double _StrikePrice;
        private PayoffType _Type;
        private double _Maturity;

        // Interface Implementation
        public double StrikePrice
        {
            get { return _StrikePrice; }
        }

        public PayoffType Type
        {
            get { return _Type; }
        }

        public double Maturity
        {
            get { return _Maturity; }
        }

        /// <summary>
        /// Initializes a new instance of the European option
        /// </summary>
```

```
/// <param name="parameters">Heston Model Parameter Object</param>
/// <param name="europeanOption">European option Parameter Object</
    param>
public EuropeanOptionFormula(HestonModelParameters parameters,
                             EuropeanOption europeanOption)
    : base(parameters.InitialStockPrice, parameters.RiskFreeRate,
           parameters.GetVariance())
{
    _StrikePrice = europeanOption.StrikePrice;
    _Type = europeanOption.Type;
    _Maturity = europeanOption.Maturity;
}

/// <summary>
/// G the specified j and phi.
/// </summary>
/// <returns>The g.</returns>
/// <param name="j">J.</param>
/// <param name="phi">Phi.</param>
private Complex g(int j, double phi)
{
    double Rho = _VarianceParameters.Rho;
    double Sigma = _VarianceParameters.Sigma;
    double[] b = _VarianceParameters.B;

    Complex A = new Complex(b[j - 1], - Rho * Sigma * phi) - d(j,
        phi);
    Complex B = new Complex(b[j - 1], - Rho * Sigma * phi) + d(j,
        phi);
    return A / B;
}

/// <summary>
/// D the specified j and phi.
/// </summary>
/// <returns>The d.</returns>
/// <param name="j">J.</param>
/// <param name="phi">Phi.</param>
private Complex d(int j, double phi)
{
    double Rho = _VarianceParameters.Rho;
    double Sigma = _VarianceParameters.Sigma;
    double[] b = _VarianceParameters.B;
    double[] u = _VarianceParameters.U;

    Complex A = new Complex(-b[j - 1], Rho * Sigma * phi);
    Complex B = Math.Pow(Sigma, 2) *
                new Complex(-Math.Pow(phi, 2), 2 * u[j - 1] * phi);

    return Complex.Sqrt(Complex.Pow(A, 2) - B);
}
/// <summary>
/// C the specified j, t and phi.
/// </summary>
/// <returns>The c.</returns>
/// <param name="j">J.</param>
```

```
/// <param name="phi">Phi.</param>
private Complex C(int j, double phi)
{

    double Rho = _VarianceParameters.Rho;
    double Sigma = _VarianceParameters.Sigma;
    double[] _b = _VarianceParameters.B;
    double alpha = _VarianceParameters.Alpha;
    double tau = _Maturity;

    Complex A = new Complex(0, _RiskFreeRate * phi * tau);

    // ln[\frac{1 - g_j (\phi)*exp^{\tau}}{-r* d_j(\phi)}{1 - g_j(\
        phi)}]
    Complex B2 = 2 * Complex.Log((1 - g(j, phi) *
                      Complex.Exp(-tau * d(j, phi))) /
                      (1 - g(j, phi)));

    Complex B = alpha / Math.Pow(Sigma, 2) *
                (new Complex(_b[j - 1], -Rho * Sigma * phi) * tau -
                 d(j, phi) * tau - B2);

    return A + B;
}

/// <summary>
/// D the specified j, t and phi.
/// </summary>
/// <returns>The d.</returns>
/// <param name="j">J.</param>
/// <param name="phi">Phi.</param>
private Complex D(int j, double phi)
{
    double Rho = _VarianceParameters.Rho;
    double Sigma = _VarianceParameters.Sigma;
    double[] _b = _VarianceParameters.B;
    double tau = _Maturity;

    Complex A = new Complex(_b[j-1], -Rho*Sigma*phi) - d(j, phi);
    Complex B1 = 1 - Complex.Exp(-tau * d(j, phi));
    Complex B2 = 1 - g(j, phi)*Complex.Exp(-tau * d(j, phi));
    return  A * (B1 / B2) / Math.Pow(Sigma, 2);
}

/// <summary>
/// Calculates the p.
/// </summary>
/// <returns>The p.</returns>
private double P(int j)
{
    double v = _VarianceParameters.V0;
    double x = Math.Log(_InitialStockPrice);

    // Integrand
    Func<double, double> integrad = phi =>
    ((Complex.Exp(new Complex(0, -phi * Math.Log(_StrikePrice))) *
```

```
            (Complex.Exp(C(j, phi) + D(j, phi) * v + new Complex(0, phi * x
                ))) /
            new Complex(0, phi)).Real;

            double integral = NewtonCotesTrapeziumRule.IntegrateAdaptive(
                integrad, 1e-6, 1000, 1e-5);

            return 0.5 + (integral / Math.PI);
        }

        /// <summary>
        /// Price this instance.
        /// </summary>
        /// <returns>The price.</returns>
        public double Price()
        {
            double A = _InitialStockPrice * P(1);
            double B = _StrikePrice * Math.Exp(-_RiskFreeRate * _Maturity )
                * P(2);

            if (Type == PayoffType.Call)
            {
                return A - B;
            }
            else
            {
                // Put - Call Parity
                return (A - B) - InitialStockPrice + StrikePrice *
                Math.Exp(-RiskFreeRate * Maturity);
            }

        }
    }
}
```

HestonCalibrationResult.cs

```
  using System;
using HestonModel.Interfaces;

namespace HestonModel.Classes.InterfaceClasses
{
    /// <summary>
    /// Class that holds the outcome details of the Heston calibration.
    /// </summary>
    public class HestonCalibrationResult : IHestonCalibrationResult
    {
        double _PricingError;
        CalibrationOutcome _MinimizerStatus;
        HestonModelParameters _Parameters;

        // Interface implementation
        public double PricingError { get { return _PricingError; } }
        public CalibrationOutcome MinimizerStatus { get { return
            _MinimizerStatus; } }
```

```
        public IHestonModelParameters Parameters { get { return (
            IHestonModelParameters)_Parameters; } }

        /// <summary>
        /// Initializes a new instance of the
        /// <see cref="T:HestonModel.Classes.InterfaceClasses.
            HestonCalibrationResult"/> class.
        /// </summary>
        /// <param name="pricingError">Pricing error.</param>
        /// <param name="status">Calibration Status.</param>
        /// <param name="parameters">Result Parameters.</param>
        public HestonCalibrationResult(double pricingError,
        CalibrationOutcome status, HestonModelParameters parameters)
        {
            _PricingError = pricingError;
            _MinimizerStatus = status;
            _Parameters = parameters;
        }
    }
}
```

HestonCalibrator

```
  using System;
using System.Collections.Generic;
using HestonModel.Classes.InterfaceClasses;

namespace HestonModel.Classes
{
    /// <summary>
    /// Class that implements the Calibration failed exception.
    /// </summary>
    public class CalibrationFailedException : Exception
    {
        public CalibrationFailedException()
        {
        }
        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.CalibrationFailedException"/> class.
        /// </summary>
        /// <param name="message">Message.</param>
        public CalibrationFailedException(string message)
            : base(message)
        {
        }
    }

    public class HestonCalibrator
    {
        // Model Params
        private double _InitialStockPrice;
        private double _RiskFreeRate;

        // Market Data
```

```
private LinkedList<OptionMarketData<EuropeanOption>>
    marketOptionsList;

// Calibration Settings
private const double defaultAccuracy = 10e-3;
private const int defaultMaxIterations = 500;
private double accuracy;
private int maxIterations;

// Calibration Vars
private CalibrationOutcome outcome;
private double[] calibratedParams;

/// <summary>
/// Initializes a new instance of the HestonCalibrator class.
/// </summary>
public HestonCalibrator()
{
    // Heston Model Params
    _RiskFreeRate = 0;
    _InitialStockPrice = 100;

    // Calibration Settings
    accuracy = defaultAccuracy;
    maxIterations = defaultMaxIterations;

    // Market Data
    marketOptionsList = new LinkedList<OptionMarketData<
        EuropeanOption>>();

    // Set default guess parameters
    calibratedParams = new double[] { 0.1, 0.05, 0.05, 0.5, 0.5 };
}

/// <summary>
/// Initializes a new instance of the HestonCalibrator class.
/// </summary>
/// <param name="parameters">Heston Model Parameters</param>
/// <param name="marketOptions">Observed Option prices</param>
/// <param name="calibrationSettings">Calibration settings</param>
public HestonCalibrator(HestonModelParameters parameters,
                        LinkedList<OptionMarketData<EuropeanOption
                            >> marketOptions,
                        CalibrationSettings calibrationSettings)
{
    // Heston Model Params
    _RiskFreeRate = parameters.RiskFreeRate;
    _InitialStockPrice = parameters.InitialStockPrice;

    // Copy market data
    marketOptionsList = new LinkedList<OptionMarketData<
        EuropeanOption>>(marketOptions);

    // Calibration Settings
    accuracy = calibrationSettings.Accuracy;
    maxIterations = calibrationSettings.MaximumNumberOfIterations;
```

```
        // Set default guess parameters
        calibratedParams = parameters.CalibrationParamsToArray();
    }

    /// <summary>
    /// Add new market data.
    /// </summary>
    /// <param name="euOption">European option.</param>
    /// <param name="price">European option Price</param>
    public void AddObservedOption(EuropeanOption euOption, double price
        )
    {
        OptionMarketData<EuropeanOption> newMarketOption = new
            OptionMarketData<EuropeanOption>(euOption, price);
        marketOptionsList.AddLast(newMarketOption);
    }

    /// <summary>
    /// Calculates the mean square error between model and market.
    /// </summary>
    /// <returns>The mean square error between model and market.</
        returns>
    /// <param name="m">Current heston parameters</param>
    public double CalcMeanSquareErrorBetweenModelAndMarket(
        HestonModelParameters m)
    {
        double meanSqErr = 0;
        foreach (OptionMarketData<EuropeanOption> marketData in
            marketOptionsList)
        {
            EuropeanOptionFormula euFormula = new EuropeanOptionFormula
                (m, marketData.Option);
            double modelPrice = euFormula.Price();

            double difference = modelPrice - marketData.Price;
            meanSqErr += difference * difference;
        }

        return meanSqErr;
    }

    /// <summary>
    /// Calibrations the objective function.
    /// </summary>
    /// <param name="paramsArray">Parameters array.</param>
    /// <param name="func">Objective Funtion.</param>
    /// <param name="obj">Object.</param>
    public void CalibrationObjectiveFunction(double[] paramsArray, ref
        double func, object obj)
    {
        VarianceProcessParameters varianceParams = new
            VarianceProcessParameters
        (paramsArray[0], paramsArray[1], paramsArray[2], paramsArray
            [3], paramsArray[4]);
        HestonModelParameters m = new HestonModelParameters(
```

```
        _InitialStockPrice, _RiskFreeRate, varianceParams);

    func = CalcMeanSquareErrorBetweenModelAndMarket(m);
}

/// <summary>
/// Calibration process.
/// </summary>
public void Calibrate()
{
    outcome = CalibrationOutcome.NotStarted;

    double[] initialParams = new double[calibratedParams.Length];
    calibratedParams.CopyTo(initialParams, 0);  // a reasonable
        starting guess

    double epsg = accuracy;
    double epsf = accuracy;
    double epsx = accuracy;
    double diffstep = 1e-6;
    int maxits = maxIterations;
    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);

    // this will do the work
    alglib.minlbfgsoptimize(state, CalibrationObjectiveFunction,
        null, null);
    double[] resultParams = new double[calibratedParams.Length];
    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             // relative function
        improvement is no more than EpsF.
        || rep.terminationtype == 2          // relative step is no
            more than EpsX.
        || rep.terminationtype == 4)
    {        // gradient norm is no more than EpsG
        outcome = CalibrationOutcome.FinishedOK;
        // we update the ''inital parameters''
        calibratedParams = resultParams;
    }
    else if (rep.terminationtype == 5)
    {   // MaxIts steps was taken
        outcome = CalibrationOutcome.FailedMaxItReached;
```

```
                // we update the ''inital parameters'' even in this case
                calibratedParams = resultParams;

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

        /// <summary>
        /// Gets the calibration status.
        /// </summary>
        /// <param name="calibOutcome">Calibration outcome.</param>
        /// <param name="pricingError">Pricing error.</param>
        public void GetCalibrationStatus(ref CalibrationOutcome
            calibOutcome, ref double pricingError)
        {
            calibOutcome = outcome;

            VarianceProcessParameters varianceParams = new
                VarianceProcessParameters
            (calibratedParams[0], calibratedParams[1], calibratedParams[2],
                 calibratedParams[3], calibratedParams[4]);
            HestonModelParameters m = new HestonModelParameters(
                _InitialStockPrice, _RiskFreeRate, varianceParams);

            pricingError = CalcMeanSquareErrorBetweenModelAndMarket(m);
        }

        /// <summary>
        /// Gets the calibrated model.
        /// </summary>
        /// <returns>The calibrated model.</returns>
        public HestonModelParameters GetCalibratedModel()
        {
            VarianceProcessParameters varianceParams = new
                VarianceProcessParameters
            (calibratedParams[0], calibratedParams[1], calibratedParams[2],
                 calibratedParams[3], calibratedParams[4]);
            HestonModelParameters m = new HestonModelParameters(
                _InitialStockPrice, _RiskFreeRate, varianceParams);
            return m;
        }
    }
}
```

HestonMC.cs

```
  using System;
using HestonModel.Interfaces;
using HestonModel.Classes.InterfaceClasses;
using MathNet.Numerics.Distributions;
using System.Threading;
```

```
namespace HestonModel.Classes
{
    /// <summary>
    /// Class that implements a tread with state used in the parallel Monte
        Carlo.
    /// </summary>
    public class WorkerMC
    {
        private int _NumberOfTrials;
        private int _NumberOfTimeSteps;

        private double sum;
        private Func<double[], double> _payoffFunc;

        private VarianceProcessParameters _VarianceParameters;

        private double _Maturity;
        private double _InitialStockPrice;
        private double _RiskFreeRate;

        public double getSum() { return sum; }

        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.WorkerMC"/> class.
        /// </summary>
        /// <param name="NumberOfTrials">Number of trials.</param>
        /// <param name="NumberOfTimeSteps">Number of time steps.</param>
        /// <param name="payoffFunc">Payoff function.</param>
        /// <param name="VarianceParameters">Variance parameters.</param>
        /// <param name="Maturity">Maturity.</param>
        /// <param name="InitialStockPrice">Initial stock price.</param>
        /// <param name="RiskFreeRate">Risk free rate.</param>
        public WorkerMC(int NumberOfTrials, int NumberOfTimeSteps, Func<
            double[], double> payoffFunc,
        VarianceProcessParameters VarianceParameters, double Maturity,
            double InitialStockPrice,
        double RiskFreeRate)
        {
            // Init MC settings
            _NumberOfTrials = NumberOfTrials;
            _NumberOfTimeSteps = NumberOfTimeSteps;
            // Init sum
            sum = 0;
            // Init Payoff function
            _payoffFunc = payoffFunc;
            // Init Variance Params
            _VarianceParameters = VarianceParameters;
            // Init Option Params
            _Maturity = Maturity;
            _InitialStockPrice = InitialStockPrice;
            _RiskFreeRate = RiskFreeRate;

        }
```

```
/// <summary>
/// Generates paths for the Monte Carlo simulation depending on the
/// state of the thread.
/// </summary>
public void ThreadGenPaths()
{
    double dz1;
    double dz2;

    // Hesto Model vars
    double rho = _VarianceParameters.Rho;
    double kappa = _VarianceParameters.Kappa;
    double theta = _VarianceParameters.Theta;
    double sigma = _VarianceParameters.Sigma;

    // Time-step
    double tau = _Maturity / _NumberOfTimeSteps;
    // MC samples for 1st Wiener process
    double[] x1 = new double[_NumberOfTimeSteps];
    // MC samples for 2nd Wiener process
    double[] x2 = new double[_NumberOfTimeSteps];

    // MC Constants
    double alpha = (4 * kappa * theta - Math.Pow(sigma, 2)) / 8;
    double beta = -kappa / 2;
    double gamma = sigma / 2;

    for (int i = 0; i < _NumberOfTrials; i++)
    {
        double[] s = new double[_NumberOfTimeSteps];
        double[] y = new double[_NumberOfTimeSteps];
        s[0] = _InitialStockPrice;
        y[0] = Math.Sqrt(_VarianceParameters.V0);

        // Samples for current path
        Normal.Samples(x1, 0, 1);
        Normal.Samples(x2, 0, 1);

        for (int j = 1; j < _NumberOfTimeSteps; j++)
        {
            dz1 = Math.Sqrt(tau) * x1[j];
            dz2 = Math.Sqrt(tau) * (rho * x1[j] +
                  Math.Sqrt(1 - Math.Pow(rho, 2)) * x2[j]);

            // Y increment
            y[j] = (y[j - 1] + gamma * dz2) / (2 - 2 * beta * tau)
                +
                      Math.Sqrt(Math.Pow(y[j - 1] + gamma * dz2, 2)
                          /
                      (4 * Math.Pow(1 - beta * tau, 2)) +
                      alpha * tau / (1 - beta * tau));

            s[j] = s[j - 1] + _RiskFreeRate * s[j - 1] * tau + y[j
                - 1] * s[j - 1] * dz1;
        }
```

```
            // Accumulate payoff for each path
            sum += _payoffFunc(s);
        }
    }

}

/// <summary>
/// Class to represent an option which is priced using Monte Carlo
    method
/// in Heston model that . This class inherits the HestonOption class
    and
/// the IMonteCarloSettings interface.
/// </summary>
public class HestonMC : HestonModelParameters, IMonteCarloSettings,
    IOption
{

    private int _NumberOfTrials;
    private int _NumberOfTimeSteps;
    private double _Maturity;

    // Implementation of Interfaces
    public int NumberOfTrials
    {
        get { return _NumberOfTrials; }
    }

    public int NumberOfTimeSteps
    {
        get { return _NumberOfTimeSteps; }
    }
    public double Maturity
    {
        get { return _Maturity; }
    }

    /// <summary>
    /// Initializes a new instance of Heston MC class.
    /// </summary>
    /// <param name="parameters">Interface holding Heston Model params
        .</param>
    /// <param name="monteCarloSimulationSettings">Interface holding
        Monte carlo simulation settings.</param>
    public HestonMC(HestonModelParameters parameters,
                    MonteCarloSettings monteCarloSimulationSettings,
                    double Maturity) :
                    base(parameters.InitialStockPrice,
                         parameters.RiskFreeRate,
                         parameters.GetVariance())
    {
        _NumberOfTrials = monteCarloSimulationSettings.NumberOfTrials;
        _NumberOfTimeSteps = monteCarloSimulationSettings.
            NumberOfTimeSteps;
        _Maturity = Maturity;
    }
```

```
/// <summary>
/// Gens the paths.
/// </summary>
/// <returns>Initializes and launches the threads to produce the
    Monte
/// Carlo paths.</returns>
/// <param name="payoffFunc">Payoff function.</param>
/// <param name="workers">Number of threads.</param>
protected double GenPaths(Func<double[], double> payoffFunc, int
    workers)
{
    // Hesto Model vars
    double rho = _VarianceParameters.Rho;
    double kappa = _VarianceParameters.Kappa;
    double theta = _VarianceParameters.Theta;
    double sigma = _VarianceParameters.Sigma;

    // Check Feller Condition
    if (2 * kappa * theta <= Math.Pow(sigma, 2))
    {
        throw new Exception("Feller Condition unsatisfied.");
    }

    // Create Thread Pool
    WorkerMC[] workerArr = new WorkerMC[workers];
    ThreadStart[] threadStartArr = new ThreadStart[workers];
    Thread[] threadArr = new Thread[workers];
    for (int k = 0; k < workers; k++)
    {
        // Create Thread with state
        workerArr[k] = new WorkerMC(NumberOfTrials / workers,
            NumberOfTimeSteps, payoffFunc,
        _VarianceParameters, Maturity, InitialStockPrice,
            RiskFreeRate);
        threadStartArr[k] = new ThreadStart(workerArr[k].
            ThreadGenPaths);
        threadArr[k] = new Thread(threadStartArr[k]);
        // Start the thread
        threadArr[k].Start();
    }

    // Join all thread with Main thread
    for (int k = 0; k < workers; k++)
    {
        threadArr[k].Join();
    }

    // Accumulate all sums
    double sum = 0;
    for (int k = 0; k < workers; k++)
    {
        sum += workerArr[k].getSum();
    }
```

```
            // Return price
            return Math.Exp(-_RiskFreeRate * _Maturity) * (sum /
                NumberOfTrials);
        }

    }

}
```

LookbackOptionMC.cs

```
  using System;
using HestonModel.Interfaces;
using HestonModel.Classes.InterfaceClasses;

namespace HestonModel.Classes
{
    /// <summary>
    /// Class to represent a Lookback option which is priced using Monte
        Carlo
    ///  method in Heston model. This class inherits the HestonMC class and
         the
    ///  IOption interface.
    /// </summary>
    public class LookbackOptionMC : HestonMC
    {

        /// <summary>
        /// Initializes a new instance of the LookbackOptionMC class.
        /// </summary>
        /// <param name="parameters">Interface holding Heston Model params
            .</param>
        /// <param name="monteCarloSimulationSettings">Interface holding
            Monte carlo simulation settings.</param>
        /// <param name="maturity">Interface holding maturity.</param>
        public LookbackOptionMC(HestonModelParameters parameters,
                                MonteCarloSettings
                                    monteCarloSimulationSettings,
                                double maturity)
            : base(parameters, monteCarloSimulationSettings, maturity)
        {}

        /// <summary>
        /// Payoff Function for the Lookback option.
        /// </summary>
        /// <returns>The payoff.</returns>
        /// <param name="S">S.</param>
        private double payoff(double[] S)
        {
            double min = double.MaxValue;

            for (int i = 0; i < S.Length; i++)
            {
                if (S[i] < min)
                {
                    min = S[i];
```

```
                }
            }
            return S[S.Length - 1] - min;
        }

        /// <summary>
        /// Price Lookback option..
        /// </summary>
        /// <returns>The price.</returns>
        /// <param name="workers">Number of threads.</param>
        public double Price(int workers)
        {
            return GenPaths(payoff, workers);
        }
    }
}
```

MonteCarloSettings.cs

```
  using System;
using HestonModel.Interfaces;

namespace HestonModel.Classes.InterfaceClasses
{
    /// <summary>
    /// Class representing settings for Monte Carlo
    /// </summary>
    public class MonteCarloSettings : IMonteCarloSettings
    {
        // Class Properties
        private int _NumberOfTrials;
        private int _NumberOfTimeSteps;

        // Implementation of Interface
        public int NumberOfTrials
        {
            get { return _NumberOfTrials; }
        }

        public int NumberOfTimeSteps
        {
            get { return _NumberOfTimeSteps; }
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.InterfaceClasses.MonteCarloSettings"/> class.
        /// </summary>
        /// <param name="NumberOfTrials">Number of trials.</param>
        /// <param name="NumberOfTimeSteps">Number of time steps.</param>
        public MonteCarloSettings(int NumberOfTrials, int NumberOfTimeSteps
            )
        {
            // Sanity Check
            if (NumberOfTrials <= 0)
            {
```

```
                throw new Exception("Number of trials of MC method must be
                    a positive number.");
            }
            if (NumberOfTimeSteps <= 0)
            {
                throw new Exception("Number of time steps of MC method must
                    be a positive number.");
            }

            _NumberOfTrials = NumberOfTrials;
            _NumberOfTimeSteps = NumberOfTimeSteps;
        }
    }
}
```

Option.cs

```
  using System;
using HestonModel.Interfaces;

namespace HestonModel.Classes.InterfaceClasses
{
    public class Option : IOption
    {
        private double _Maturity;

        // Interface Implementation
        public double Maturity
        {
            get { return _Maturity; }
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.InterfaceClasses.Option"/> class.
        /// </summary>
        /// <param name="Maturity">Maturity.</param>
        public Option(double Maturity)
        {
            // Sanity Check
            if (Maturity <= 0)
            {
                throw new Exception("Maturity must be a positive number.");
            }
            _Maturity = Maturity;
        }
    }
}
```

OptionMarketData.cs

```
  using System;
using HestonModel.Interfaces;

namespace HestonModel.Classes.InterfaceClasses
{
```

```
    /// <summary>
    /// Generic Class representing data set of options and treir price in
        the market.
    /// </summary>
    public class OptionMarketData<T> : IOptionMarketData<T> where T :
        IEuropeanOption
    {
        private T _Option;
        private double _Price;

        public double Price
        {
            get { return _Price; }
        }
        public T Option
        {
            get { return _Option; }
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="T:HestonModel.
            Classes.InterfaceClasses.OptionMarketData`1"/> class.
        /// </summary>
        /// <param name="option">Option.</param>
        /// <param name="price">Price.</param>
        public OptionMarketData(T option, double price)
        {
            _Option = option;
            _Price = price;
        }
    }
}
```

VarianceProcessParameters.cs

```
  using System;
using HestonModel.Interfaces;

namespace HestonModel.Classes.InterfaceClasses
{
    /// <summary>
    /// Class representing variance paramenters for Heston Model
    /// </summary>
    public class VarianceProcessParameters : IVarianceProcessParameters
    {
        // Variance Process Parameters
        protected double _Kappa;
        protected double _Theta;
        protected double _Sigma;
        protected double _V0;
        protected double _Rho;

        // Equation Vars
        protected double[] _u = new double[2];
        protected double _alpha;
        protected double[] _b = new double[2];
```

```
public double Kappa
{
    get { return _Kappa; }
}

public double Theta
{
    get { return _Theta; }
}

public double Sigma
{
    get { return _Sigma; }
}

public double V0
{
    get { return _V0; }
}

public double Rho
{
    get { return _Rho; }
}

public double Alpha
{
    get { return _alpha; }
}

public double[] U
{
    get { return _u; }
}

public double[] B
{
    get { return _b; }
}

/// <summary>
/// Initializes a new instance of the
/// <see cref="T:HestonModel.Classes.InterfaceClasses.
    VarianceProcessParameters"/> class.
/// </summary>
/// <param name="Kappa">Kappa.</param>
/// <param name="Theta">Theta.</param>
/// <param name="Sigma">Sigma.</param>
/// <param name="V0">V0.</param>
/// <param name="Rho">Rho.</param>
public VarianceProcessParameters(double Kappa, double Theta,
                                 double Sigma, double V0,
                                 double Rho)
{
    // Check Sanity for parameters
```

```
            if (Sigma < 0)
            {
                throw new Exception("Parameter sigma cannot be a negative
                    number.");
            }
            if (V0 < 0)
            {
                throw new Exception("Volatility cannot be a negative number
                    .");
            }

            // Initialize Heston
            _Kappa = Kappa;
            _Theta = Theta;
            _Sigma = Sigma;
            _V0 = V0;
            _Rho = Rho;

            // Initialize Equation helping vars
            _alpha = _Kappa * _Theta;
            _b[0] = _Kappa - _Rho * _Sigma;
            _b[1] = _Kappa;
            _u[0] = 0.5;
            _u[1] = -0.5;
        }
    }
}
```