Object oOriented Programming with Applications
Final Assignment Report

Thomas Zacharis
UUN: s1774194

January 16, 2018

1 Project structure

The project is organized into three projects as follows:

- `Alglib`: Used in the Heston model calibration.
- `HestonModel`: This is the main project. The option pricing and calibration algorithms are implemented in this project. Building the project produces a shared library. The project contains the following files:
  - `Heston.cs`: Defines the static methods wrapping the functionality provided the rest of the project.
  - `HestonFormula.cs`: In this file one can the `HestonFormula` class whih implements European call and put pricing using the Heston formula. The class `HestonFormula2` implements the same formula, but this time using the branch correction described in Section 2.4 of the accompanying paper. Neither implementation provided me with satisfactory results, in the sense that for certain parameter ranges the two varied significantly.
  - `HestonMC.cs`: In this file European, Asian and lookback options are priced using a Monte Carlo algorithm.
  - `PathGenerator.cs`: This file is used by the Monter Carlo algorithms to generate sample paths of the underlying stochastic process.
  - `HestonCalibrator.cs`: Here I implement parameter calibration using `Alglib`.
- `HestonXL`: The Excel interface. Builds into a shared library and an Excel plugin. The functions in `HestonXLInterface.cs` call into `Heston.cs`. The excel file `HestonExample.xlsx` is also included, showcasing the exported functions. Allow a few moments when opening it, since the Monte Carlo calculations and the calibration may take a while.
- `test`: Contains only a `Main` function used for testing and generating the numerical data contained in this report.

2 Implementing the Heston formula

I implemented the Heston formula twice, one time using the formulae in Section 2.3 of the accompanying paper and one more time using Section

2.4. The pricing function in `Heston.cs` uses the first implementation. The results where disappointed. In many cases the two implementations return totally different results, sometimes one of them will fail with `Nan` of return improbable values. My first suspicion was that this behavior has something to do with the integration algorithm used in the final step, but after trying different algorithms and varying the parameters no improvement was made.

For example, take

$$r = 0.025 \qquad \kappa^* = 1.5768 \qquad \theta^* = 0.0398 \qquad \sigma = 0.5751$$

$$\rho = -0.5711 \qquad \upsilon = 0.0175$$

and fill the table requested in the task for $S = 100$:

Stike K	T	Price 1	Price 2
100	1	7.27	5.44
100	2	13.45	8.53
100	3	23.16	11.33
100	4	16.18	13.93
100	15	`NaN`	35.86

Price 1 indicated result returned by the `HestonFormula` class, while Price 2 indicated the value returned by `HestonFormula2`. As we can see, not even close and even got a `NaN`, probably because on some division by zero. If we however increase the risk free rate to $r = 0.2$ then we get he following result($S = 100$ again):

Stike K	T	Price 1	Price 2
100	1	19.54	18.9
100	2	34.62	33.45
100	3	46.60	45.45
100	4	63.37	63.33
100	10	89.27	94.92

Both implementations here give different answer again, but at least they are somewhat similar. It seems that the larger the risk free rate the closer the two formulas are.

For put options I used the put-call parity. However, the results are off again with negative and non-sensical numbers being returned in many cases. I think this indicates that both implementations of the formula are wrong.

3 Monte Carlo implementation

I had better luck with the Monte Carlo algorithm. The class implementing it is called `HestoMC`. For the discretization of υ I use the Dereich, Neuenkirch and Szpruch scheme.

The path generation is delegated to the `PathGenerator` class and is done using a simple `arallel.For` loop. On my hardware this is almost 5x faster that a simple loop.

Numerical data for call options

$$r = 0.1 \qquad \kappa^* = 2 \qquad \theta^* = 0.06 \qquad \sigma = 0.4$$

$$\rho = 0.5 \qquad \upsilon = 0.04$$

are as following (the Heston formula implementation used is the first one):

S	K	T	Heston formula	Monte Carlo
100	100	1	13.61	13.79
100	100	2	22.40	22.28
100	100	3	29.55	29.86
100	100	4	34.23	36.70
100	100	10	68.23	77.53

For small maturity times the two values are almost identical but the difference between them grows with T. Consider the discussion in the previous section, this has to be because of the defect in my implenentation of the formula.

I was tasked with pricing Asian arithmetic options and lookback options. For the Asian options, I modified the path generating function to `yield return` the paths upon hitting a monitoring time giving the caller a chance to perform operations on then and beofre resuming. Please note that the final time T is not automatically considered a monitoring time.

Numerical data for Asian calls:

S	K	T	$T_1, \cdots, T_M$	Call
100	100	1	0.75, 1.00	6.7
100	100	2	0.25,0.50,0.75,1.0,1.25,1.5,1.75	2.5
100	100	3	1.0, 2.0, 3.0	9.1

Lookback options where straightforward to implement. I had to modify the path generator to keep track of the mininum value of each path and return it along the paths. I got the following results:

S	K	T	Lookback Call
100	100	1	19.3
100	100	3	37.4
100	100	5	51.8
100	100	7	59.6
100	100	9	68

4 Calibration

For the calibration class in `HestonCalibration.cs` I took the code from the last workshop and adjusted it for the project. Because I am certain that my Heston formula is not correct, the following numerical data are probably meaningless. Despite this, I gave the calibrator the following guess:

$$\kappa^* = 2 \qquad \theta^* = 0.2 \qquad \sigma = 0.5$$

$$\rho = -0.3 \qquad \upsilon = 0.02$$

The calibrator finished the minimization without exceeding the maximum number of iterations and returned

$$\kappa^* = 1.9727 \qquad \theta^* = 0.115 \qquad \sigma = 0.7134$$

$$\rho = -0.0674 \qquad \upsilon = 0.1288$$

The formula, with the calibrated parameters gives us, for the same options we have market data for:

S	K	T	Calibrated model call
100	80	1	24.44
100	90	1	19.19
100	80	2	30.17
100	100	2	20.39
100	100	1.5	17.10

The model is fairly close to the market data.

The calibration process is sensitive to and extreme initial guess. For example, in the above initial guess set $\rho = -2.0$ and keep everything else fixed. The calibrated parameters then are

$$\kappa^* = 1.99 \qquad \theta^* = -0.05 \qquad \sigma = 0.7134$$

$$\rho = -2.0 \qquad \upsilon = 0.02$$

The calibrator returns, almost instantly, indicating it must have hit a local minimum. We also see that θ^* is negative, and the other parameters have barely changed. Not suprisingly, the model is completely off.

S	K	T	Calibrated model call
100	80	1	22.00
100	90	1	12.22
100	80	2	23.90
100	100	2	4.87
100	100	1.5	3.68

5 Code

File: `Heston.cs`

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

namespace HestonModel
{
    public interface ICalibrationResult
    {
        CalibrationOutcome MinimizerStatus { get; }
        double PricingError { get; } // what's this supposed to be set to?
    }

    public interface IHestonCalibrationResult : ICalibrationResult
    {
        IHestonParameters Parameters { get; }
    }

    public interface IHestonParameters
    {
        double RiskFreeRate { get; } // Continuously compounded risk free rate
        double Kappa { get; } //  Mean reversion speed in Heston model
        double Theta { get; } // The long-term mean in Heston model
        double Sigma { get; } // The vol of vol in Heston model
        double Rho { get; } // The correlation between asset price and vol of vol in Heston model
        double V0 { get; } // Initial variance in Heston model
    }

    public interface IOption
    {
        double Maturity { get; }  //Option maturity as a year fraction (i.e. 1 means one year)
    }

    public interface IEuropeanOption : IOption
    {
        double Strike { get; }
        PayoffType Type { get; }
    }

    public interface IOptionMarketData<T> where T : IOption
    {
        T Option { get; }
        double Price { get; }
    }

    public enum PayoffType { Call, Put };

    public enum CalibrationOutcome { NotStarted, FinishedOK, FailedMaxItReached, FailedOtherReason };


    public struct HestonCalibrationResult : IHestonCalibrationResult
    {
        public CalibrationOutcome MinimizerStatus { get; set; }
        public double PricingError { get; set; }
```

```
54          public IHestonParameters Parameters { get; set; }
55      }
56
57      public struct HestonParameters : IHestonParameters
58      {
59          public double RiskFreeRate { get; set; }
60          public double Kappa { get; set; }
61          public double Theta { get; set; }
62          public double Sigma { get; set; }
63          public double Rho { get; set; }
64          public double V0 { get; set; }
65      }
66
67      public struct EuropeanOption : IEuropeanOption
68      {
69          public EuropeanOption(PayoffType _type, double T, double K)
70          {
71              Type = _type;
72              Maturity = T;
73              Strike = K;
74          }
75
76          public PayoffType Type { get; set; }
77          public double Maturity { get; set; }
78          public double Strike { get; set; }
79      }
80
81      public class OptionMarketData<T> : IOptionMarketData<T> where T : IOption
82      {
83          public T Option { get; set; }
84          public double Price { get; set; }
85      }
86
87      /// <summary>
88      /// This class will be used for grading. Please keep it within "HestonModel" namespace,
89      /// feel free to add to it as needed, but don't remove the existing methods or modify their signatures
90      /// </summary>
91      public static class Heston
92      {
93          /// <summary>
94          /// Method for calibrating the heston model parameters.
95          /// </summary>
96          /// <param name="underlying">The current stock price</param>
97          /// <param name="guessModelParameters">Object implementing IHestonParameters interface containing the risk-free
            ↪  rate
98          /// and initial guess parameters to be used in calibration.</param>
99          /// <param name="referenceData">A colection of objects implementing IOptionMarketData<IEuropeanOption> interface
            .
100         /// These should contain the reference data used for calibration.</param>
101         /// <param name="accuracy">A parameter influencing the accuracy the minimization algorithm is trying to achieve.
            ↪  Note that we are
102         /// allowing more options than parameters so we don't necessarily expect to be able to re-price all the
            ↪  options.</param>
103         /// <param name="maxIterations">The maximum number of iterations you allow the minimization algorithm to use.
            ↪  Note that even 10 iterations
104         /// can take more than a few seconds!</param>
105         /// <returns>Object implementing IHestonCalibrationResult interface which contains calibrated model parameters
            ↪  and addtional diagnostic information</returns>
106         public static IHestonCalibrationResult CalibrateHestonParameters(double underlying, IHestonParameters
            ↪  guessModelParameters, IEnumerable<IOptionMarketData<IEuropeanOption>> referenceData, double accuracy, int
            ↪  maxIterations)
107         {
108             if(accuracy <= 0.0 || maxIterations <= 0 || underlying <= 0 || !referenceData.Any())
109                 throw new ArgumentException("invalid arguments in calibration");
110
111             var cal = new HestonCalibrator(guessModelParameters.RiskFreeRate, new HestonParams
112             {
113                 kappa = guessModelParameters.Kappa,
114                 theta = guessModelParameters.Theta,
115                 sigma = guessModelParameters.Sigma,
116                 rho   = guessModelParameters.Rho,
117                 v0    = guessModelParameters.V0
118             });
119
120             foreach(var option in referenceData)
121             {
122                 cal.market_data.Add(new MarketDataEntry
123                 {
124                     type = option.Option.Type,
125                     S = underlying,
126                     K = option.Option.Strike,
127                     T = option.Option.Maturity,
128                     price = option.Price
```

```
129                 });
130             }
131
132             cal.Calibrate(accuracy, maxIterations);
133
134             var p = new HestonParameters
135             {
136                 RiskFreeRate = guessModelParameters.RiskFreeRate,
137                 Kappa = cal.calibrated_params.kappa,
138                 Theta = cal.calibrated_params.theta,
139                 Sigma = cal.calibrated_params.sigma,
140                 Rho = cal.calibrated_params.rho,
141                 V0 = cal.calibrated_params.v0
142             };
143
144             var res = new HestonCalibrationResult
145             {
146                 MinimizerStatus = cal.outcome,
147                 Parameters = p,
148                 PricingError = cal.MeanSquareError(new HestonFormula(p.RiskFreeRate, p.V0, p.Kappa, p.Theta, p.Sigma,
                    ↪  p.Rho))
149             };
150
151             return res;
152         }
153
154         /// <summary>
155         /// Price a call or put European option in the Heston model using the
156         /// Heston formula. This should be accurate to 5 decimal places
157         /// </summary>
158         /// <param name="underlying">The current stock price</param>
159         /// <param name="parameters">Object implementing IHestonParameters interface containing the risk-free rate
160         /// and the Heston model parameters.</param>
161         /// <param name="option">Object implementing IEuropeanOption interface, containing the option parameters.</param>
162         /// <returns>Option price</returns>
163         public static double HestonOneOptionPrice(double underlyingPrice, IHestonParameters parameters, IEuropeanOption
            ↪  option)
164         {
165             if(underlyingPrice <= 0.0 || option.Maturity < 0.0 || option.Strike < 0.0)
166                 throw new ArgumentException();
167
168             var hf = new HestonFormula(parameters.RiskFreeRate, parameters.V0, parameters.Kappa, parameters.Theta,
                ↪  parameters.Sigma, parameters.Rho);
169
170             return (option.Type == PayoffType.Call) ? hf.PriceEuropeanCallOption(underlyingPrice, option.Strike,
                ↪  option.Maturity) :
171                             hf.PriceEuropeanPutOption(underlyingPrice, option.Strike, option.Maturity);
172
173         }
174
175         /// <summary>
176         /// Price a call or put European option in the Heston model using the
177         /// Monte-Carlo method. Accuracy will depend on number of time steps and samples
178         /// </summary>
179         /// <param name="underlying">The current stock price</param>
180         /// <param name="parameters">Object implementing IHestonParameters interface containing the risk-free rate
181         /// and the Heston model parameters.</param>
182         /// <param name="option">Object implementing IEuropeanOption interface, containing the option parameters.</param>
183         /// <param name="numSamplePaths">The number of sample paths generated for Monte-Carlo valuation</param>
184         /// <param name="numSteps">The number of time steps for each path</param>
185         /// <returns>Option price</returns>
186         public static double HestonOneOptionPriceMC(double underlying, IHestonParameters parameters, IEuropeanOption
            ↪  option, int numSamplePaths, int numSteps)
187         {
188             // feller condition is checked in HestonMC ctor
189             if(underlying <= 0.0 || option.Maturity < 0.0 || option.Strike < 0.0 || numSamplePaths <= 0 || numSteps <= 0)
190                 throw new ArgumentException();
191
192             var mc = new HestonMC(parameters.RiskFreeRate, parameters.V0, parameters.Kappa, parameters.Theta,
                ↪  parameters.Sigma, parameters.Rho);
193
194             return (option.Type == PayoffType.Call) ? mc.GetCallOptionPrice(underlying, option.Strike, option.Maturity,
                ↪  numSamplePaths, numSteps) :
195                 mc.GetPutOptionPrice(underlying, option.Strike, option.Maturity, numSamplePaths, numSteps);
196         }
197
198         /// <summary>
199         /// Price a call or put Asian option in the Heston model using the
200         /// Monte-Carlo method. Accuracy will depend on number of time steps and samples</summary>
201         /// <param name="underlying">The current stock price</param>
202         /// <param name="parameters">Object implementing IHestonParameters interface containing the risk-free rate
203         /// and the Heston model parameters.</param>
204         /// <param name="maturity">Option maturity</param>
```

```
205         /// <param name="strike">Option strike</param>
206         /// <param name="monitoringTimes">Collection of times (expressed as year fraction)
207         /// denoting the times over which the average is calculated.</param>
208         /// <param name="payoffType">Payoff type</param>
209         /// <param name="numSamplePaths">The number of sample paths generated for Monte-Carlo valuation</param>
210         /// <param name="numSteps">The number of time steps for each path</param>
211         /// <returns>Option price</returns>
212         public static double HestonAsianOptionPriceMC(double underlying, IHestonParameters parameters, double maturity,
            ↪ double strike, IEnumerable<double> monitoringTimes, PayoffType payoffType, int numSamplePaths, int numSteps)
213         {
214             // feller condition is checked in HestonMC ctor
215             if(underlying <= 0.0 || maturity < 0.0 || strike < 0.0 || numSamplePaths <= 0 || numSteps <= 0)
216                 throw new ArgumentException();
217
218             // CAREFUL: I do not consider the maturity as a monitoring time (unless it is included in the monitoring
                ↪ times)
219             // see HestonMC.cs too
220             // this is in contrast to the paper on Monter Carlo in the course website where it is considered
                ↪ automatically in the
221             // monitoring times
222
223             var observe_times = monitoringTimes.ToArray();
224             Array.Sort(observe_times);
225
226             if(observe_times.Max() > maturity || observe_times.Min() < 0.0)
227                 throw new ArgumentException("fail monitoring times ");
228
229             var mc = new HestonMC(parameters.RiskFreeRate, parameters.V0, parameters.Kappa, parameters.Theta,
                ↪ parameters.Sigma, parameters.Rho);
230
231             return (payoffType == PayoffType.Call) ? mc.GetAsianCallOptionPrice(underlying, strike, maturity,
                ↪ observe_times, numSamplePaths, numSteps) :
232                 mc.GetAsianPutOptionPrice(underlying, strike, maturity, observe_times, numSamplePaths, numSteps);
233         }
234
235         /// <summary>
236         /// Price a lookback option in the Heston model using the
237         /// a Monte-Carlo method. Accuracy will depend on number of time steps and samples </summary>
238         /// <param name="underlying">The current stock price</param>
239         /// <param name="parameters">Object implementing IHestonParameters interface containing the risk-free rate
240         /// and the Heston model parameters.</param>
241         /// <param name="maturity">Option maturity</param>
242         /// <param name="numSamplePaths">The number of sample paths generated for Monte-Carlo valuation</param>
243         /// <param name="numSteps">The number of time steps for each path</param>
244         /// <returns>Option price</returns>
245         public static double HestonLookbackOptionPriceMC(double underlying, IHestonParameters parameters, double
            ↪ maturity, int numSamplePaths, int numSteps)
246         {
247             // feller condition is checked in HestonMC ctor
248             if(underlying <= 0.0 || maturity < 0.0 || numSamplePaths <= 0 || numSteps <= 0)
249                 throw new ArgumentException();
250
251             var mc = new HestonMC(parameters.RiskFreeRate, parameters.V0, parameters.Kappa, parameters.Theta,
                ↪ parameters.Sigma, parameters.Rho);
252
253             return mc.GetLookbackOptionPrice(underlying, maturity, numSamplePaths, numSteps);
254         }
255     }
256 }
```

File: HestonFormula.cs

```
1   /*
2    * 10/01/2018
3    * ~ thwmakos ~
4    */
5
6   using System;
7   using System.Numerics;
8   using System.Linq;
9
10  using MathNet.Numerics;
11  using MathNet.Numerics.Integration;
12
13  namespace HestonModel
14  {
15      // this is with formulas from 2.4
16      public class HestonFormula2
17      {
18          public HestonFormula2(double _r, double _v0, double _kappa, double _theta, double _sigma, double _rho)
```

```
19          {
20              kappa = _kappa;
21              theta = _theta;
22              sigma = _sigma;
23              rho = _rho;
24              r = _r;
25              v0 = _v0;
26
27              a = kappa * theta;
28              b1 = kappa - rho * sigma;
29              b2 = kappa;
30          }
31
32          public double PriceEuropeanCallOption(double S, double K, double T)
33          {
34              var P1 = Pj(1, 0.0, Math.Log(K), v0, K, T, r);
35              var P2 = Pj(2, 0.0, Math.Log(K), v0, K, T, r);
36
37              return S * P1 - K * Math.Exp(-r * T) * P2;
38          }
39
40          // put / call parity used for put options
41          public double PriceEuropeanPutOption(double S, double K, double T)
42          {
43              var t1 = PriceEuropeanCallOption(S, K, T);
44              var t2 = Math.Exp(-r * T) * K;
45
46              //return PriceEuropeanCallOption(S, K, T) + (Math.Exp(-r * T) * K) - S;
47              return t1 + t2 - S;
48          }
49
50          private Complex dj(double phi, double bj, double uj)
51          {
52
53              var c1 = rho * sigma * phi * Complex.ImaginaryOne - bj;
54              c1 = c1 * c1; // square it
55
56              var c2 = (sigma * sigma) * (2.0 * uj * phi * Complex.ImaginaryOne - phi * phi);
57
58              return Complex.Sqrt(c1 - c2);
59          }
60
61          private Complex gj(double phi, double bj, double uj)
62          {
63              var deej = dj(phi, bj, uj);
64              var c1 = bj - rho * sigma * phi * Complex.ImaginaryOne;
65
66              return (c1 - deej) / (c1 + deej);
67          }
68
69          private Complex Cj(double tau, double phi, double r, double bj, double uj)
70          {
71              var deej = dj(phi, bj, uj);
72              var geej = gj(phi, bj, uj);
73
74              var c1 = (bj - (rho * sigma * phi) * Complex.ImaginaryOne - deej) * tau;
75              var c2 = 2.0 * Complex.Log((1.0 - geej * Complex.Exp(-tau * deej)) / (1.0 - geej));
76
77              return r * phi * tau * Complex.ImaginaryOne + (a / sigma * sigma) * (c1 - c2);
78          }
79
80          private Complex Dj(double tau, double phi, double bj, double uj)
81          {
82              var deej = dj(phi, bj, uj);
83              var geej = gj(phi, bj, uj);
84
85              var c1 = (bj - rho * sigma * phi * Complex.ImaginaryOne - deej) / (sigma * sigma);
86              var c2 = (1.0 - Complex.Exp(-tau * deej)) / (1.0 - geej * Complex.Exp(-tau * deej));
87
88              return c1 * c2;
89          }
90
91          private Complex Phij(double time, double x, double v, double phi, double T, double r, double bj, double uj)
92          {
93              return Complex.Exp(Cj(T - time, phi, r, bj, uj) + Dj(T - time, phi, bj, uj) * v + phi * x *
                ↪  Complex.ImaginaryOne);
94          }
95
96          private double Pj(int j, double time, double x, double v, double K, double T, double r)
97          {
98              System.Diagnostics.Debug.Assert(j == 1 || j == 2);
99
100             var bj = (j == 1) ? b1 : b2;
```

```
101             var uj = (j == 1) ? 0.5 : -0.5;
102
103             // you can actually have function in function ?
104             double integrand(double phi)
105             {
106                 var c1 = Complex.Exp(-phi * Math.Log(K) * Complex.ImaginaryOne);
107
108                 var temp = ((c1 * Phij(time, x, v, phi, T, r, bj, uj)) / (phi * Complex.ImaginaryOne)).Real;
109
110                 return temp;
111             }
112
113             // in Heston's original paper he says that the integral decays fast so integrating from 0 to 200 should be
                ↪  fine
114             // I actually start from 0.1 to avoid divisio by zero
115             var integral = SimpsonRule.IntegrateComposite(integrand, 0.001, 100.0, 200);
116
117             //var integral = MathNet.Numerics.Integration.SimpsonRule.IntegrateComposite(integrand, 0.1, 10000.0, 10000);
118
119             return 0.5 + (1.0 / Math.PI) * integral;
120         }
121
122         private readonly double kappa;
123         private readonly double theta;
124         private readonly double sigma;
125         private readonly double rho;
126         private readonly double r;
127         private readonly double v0;
128
129         private readonly double a;
130         private readonly double b1;
131         private readonly double b2;
132
133     }
134
135     // actually this is the third rewrite
136     // I could not get the formula from the paper work for small maturity times
137     // when the risk free rate is <0.2 so
138     // I translated the MATLAB code from http://www.hec.unil.ch/matlabcodes/option_pricing.html
139     // to C# (CF_SVj.m and HestonCall.m). I think it does not use the Little Heston trap
140     // (section 2.3)
141     public class HestonFormula
142     {
143         public HestonFormula(double _r, double _v0, double _kappa, double _theta, double _sigma, double _rho)
144         {
145             kappa = _kappa;
146             theta = _theta;
147             sigma = _sigma;
148             rho   = _rho;
149             r     = _r;
150             v0    = _v0;
151
152             a  = kappa * theta;
153             b1 = kappa - rho * sigma;
154             b2 = kappa;
155         }
156
157         public double PriceEuropeanCallOption(double S, double K, double T)
158         {
159             var P1 = Pj(Math.Log(S), T, b1, 0.5);
160             var P2 = Pj(Math.Log(S), T, b2, -0.5);
161
162             return S * P1 - K * Math.Exp(-r * T) * P2;
163         }
164
165         // put call parity
166         public double PriceEuropeanPutOption(double S, double K, double T)
167         {
168             return PriceEuropeanCallOption(S, K, T) + Math.Exp(-r * T) * K - S;
169         }
170
171         private double Pj(double x, double tau, double bj, double uj)
172         {
173             double integrand(double phi) // you can have functions within function wow
174             {
175                 var f = Phi_j(x, tau, bj, uj, phi);
176
177                 return (Complex.Exp(-phi * x * Complex.ImaginaryOne) * f / (phi * Complex.ImaginaryOne)).Real;
178             }
179
180             // start from 0.01 to avoid division by zero
181             // supposedly the integrand decays quickly so integrating from 0 to 100
182             // should suffice
```

```
183             var intgrl = SimpsonRule.IntegrateComposite(integrand, 0.01, 100.0, 200);
184
185             return 0.5 + (1.0 / Math.PI) * intgrl;
186         }
187
188         private Complex Phi_j(double x, double tau, double bj, double uj, double phi)
189         {
190             var xj = bj - rho * sigma * phi * Complex.ImaginaryOne;
191             var dj = Complex.Sqrt(xj * xj - (sigma * sigma) * (2 * uj * phi * Complex.ImaginaryOne - phi * phi));
192             var gj = (xj + dj) / (xj - dj);
193             var D = (xj + dj) / (sigma * sigma) * (1.0 - Complex.Exp(dj * tau)) / (1.0 - gj * Complex.Exp(dj * tau));
194             var xx = (1.0 - gj * Complex.Exp(dj * tau)) / (1.0 - gj);
195             var C = r * phi * tau * Complex.ImaginaryOne + a / (sigma * sigma) * ((xj + dj) * tau - 2.0 *
                ↪  Complex.Log(xx));
196
197             return Complex.Exp(C + D * v0 + phi * x * Complex.ImaginaryOne);
198         }
199
200         private readonly double kappa;
201         private readonly double theta;
202         private readonly double sigma;
203         private readonly double rho;
204         private readonly double r;
205         private readonly double v0;
206
207         private readonly double a;
208         private readonly double b1;
209         private readonly double b2;
210     }
211 }
```

File: HestonMC.cs

```
1   /*
2    * HestonMC.cs - 13/01/2018
3    */
4
5   using System;
6   using System.Linq;
7
8   namespace HestonModel
9   {
10      // option pricing using monte carlo
11      public class HestonMC
12      {
13          public HestonMC(double _r, double _v0, double _kappa, double _theta, double _sigma, double _rho)
14          {
15              kappa = _kappa;
16              theta = _theta;
17              sigma = _sigma;
18              rho = _rho;
19              r = _r;
20              v0 = _v0;
21
22              if(2.0 * kappa * theta <= sigma * sigma)
23                  throw new ArgumentException("Feller condition violated!");
24
25              pg = new PathGenerator(r, v0, kappa, theta, sigma, rho);
26          }
27
28          public double GetCallOptionPrice(double S, // curr price
29                                           double K, // strike
30                                           double T, // maturity
31                                           int num_paths = 5000,
32                                           int num_timesteps = 1000)
33          {
34
35              var paths = GenEuropeanOptionSample(S, K, T, num_paths, num_timesteps);
36
37              // LINQ is actually usefull
38              // I hope it doens't just add all numbers together
39              // and then divide because this will cause terrible
40              // fp inaccuracies for large numbers of paths or it
41              // may cause overflow
42              // FIXME: exception is thrown in case of overflow?
43              var temp = paths.Average((x) => Math.Max(x - K, 0.0));
44
45              return Math.Exp(-r * T) * temp;
46          }
47
```

```
48          public double GetPutOptionPrice(double S,
49                                          double K,
50                                          double T,
51                                          int num_paths = 5000,
52                                          int num_timesteps = 1000)
53          {
54              var paths = GenEuropeanOptionSample(S, K, T, num_paths, num_timesteps);
55
56              var temp = paths.Average(x => Math.Max(K - x, 0.0));
57
58              return Math.Exp(-r * T) * temp;
59          }
60
61          // arithmetic asian call option
62          public double GetAsianCallOptionPrice(double S,
63                                                double K,
64                                                double T,
65                                                double[] observe_times,
66                                                int num_paths = 5000,
67                                                int num_timesteps = 1000)
68          {
69              var samples = GenAsianOptionSample(S, K, T, observe_times, num_paths, num_timesteps);
70
71              return Math.Exp(-r * T) * samples.Average((x) => Math.Max(x - K, 0));
72          }
73
74          // arithmetic asian put
75          public double GetAsianPutOptionPrice(double S,
76                                               double K,
77                                               double T,
78                                               double[] observe_times,
79                                               int num_paths = 5000,
80                                               int num_timesteps = 100)
81          {
82              var samples = GenAsianOptionSample(S, K, T, observe_times, num_paths, num_timesteps);
83
84              return Math.Exp(-r * T) * samples.Average(x => Math.Max(K - x, 0));
85          }
86
87          // lookback option
88          public double GetLookbackOptionPrice(double S,
89                                               double T,
90                                               int num_paths = 5000,
91                                               int num_timesteps = 1000)
92          {
93              var t = pg.GenPaths(num_paths, num_timesteps, T, S, true);
94
95              var samples = t.Item1;
96              var mins = t.Item2;
97
98              var diffs = Enumerable.Zip(samples, mins, (x, y) => x - y);
99
100             return Math.Exp(-r * T) * diffs.Average();
101         }
102
103         // this simply generates sample paths for use in European call/put
104         // pricing
105         private double[] GenEuropeanOptionSample(double S,
106                                                  double K,
107                                                  double T,
108                                                  int num_paths,
109                                                  int num_timesteps)
110         {
111             return pg.GenPaths(num_paths, num_timesteps, T, S).Item1;
112         }
113
114         // same as above, but for asian put and call
115         // generate the sums 1/M  S(T_m)
116         private double[] GenAsianOptionSample(double S,
117                                               double K,
118                                               double T,
119                                               double[] observe_times,
120                                               int num_paths,
121                                               int num_timesteps)
122         {
123             int num_timesteps_per_year = (int)(num_timesteps / T);
124
125             // sums of S(T_m) for all m for each path
126             var sums = new double[num_paths];
127
128             for(var i = 0; i < num_paths; i++)
129                 sums[i] = 0;
130
```

```
131
132             foreach(var paths in pg.GenPaths(num_paths, num_timesteps_per_year, T, S, observe_times))
133             {
134                 for(var i = 0; i < num_paths; i++)
135                     sums[i] += paths[i];
136             }
137
138             for(var i = 0; i < num_paths; i++)
139                 sums[i] /= observe_times.Length; // divide by M
140
141             //var averages = sums.Select(x => x / observe_times.Length);
142
143             return sums;
144         }
145
146         // class state, immutable if you want to change these
147         // just create a new instance
148         private readonly double kappa;
149         private readonly double theta;
150         private readonly double sigma;
151         private readonly double rho;
152         private readonly double r;
153         private readonly double v0;
154
155         private readonly PathGenerator pg;
156
157     }
158 }
```

File: `PathGenerator.cs`

```
1   /*
2    * 11/01/2018
3    * ~thwmakos~
4    */
5
6   using System;
7   using System.Threading.Tasks;
8   using System.Collections.Generic;
9   using System.Diagnostics;
10
11  using MathNet.Numerics.Distributions;
12
13  namespace HestonModel
14  {
15      public class PathGenerator
16      {
17          // simply copy the arguments and calculate some extra parameters needed for
18          // the discretization of the volatility
19          public PathGenerator(double _r, double _v0, double _kappa, double _theta, double _sigma, double _rho)
20          {
21              r     = _r;
22              v0    = _v0;
23              kappa = _kappa;
24              theta = _theta;
25              sigma = _sigma;
26              rho   = _rho;
27
28              alpha = (4 * kappa * theta - sigma * sigma) / 8.0;
29              beta  = -kappa / 2.0;
30              gamma = sigma / 2.0;
31
32              one_minus_sqrt_rho = Math.Sqrt(1 - rho * rho);
33          }
34
35          // get S(T) for num_paths number of paths with initial condition S(0) = 0, v(0)=v0
36          // used for european options
37          // set the bool flag to true to keep the minimum of the
38          // approximation sequence (used in lookback options)
39          // returns a tuple, first item is the value of the paths,second is the minimum of each path
40          // if ret_mins == false the second items of the tuple is null
41          public Tuple<double[], double[]> GenPaths(int num_paths, int num_timesteps, double T, double S0, bool ret_mins =
            ↪  false)
42          {
43              Debug.Assert(num_paths > 0);
44              Debug.Assert(num_timesteps > 0);
45              Debug.Assert(v0 > 0.0);
46
47              var paths = new double[num_paths];
48              double[] mins = null;
```

```
49
50              if(ret_mins == true)
51                  mins = new double[num_paths];
52
53              var step = T / num_timesteps;
54              var sqrt_step = Math.Sqrt(step);
55              var sqrt_v0 = Math.Sqrt(v0);
56
57              Parallel.For(0, num_paths, (i) =>
58              {
59                  var S = S0;
60                  var y = sqrt_v0;
61                  var min = S0;
62
63                  for(int j = 0; j < num_timesteps; j++)
64                  {
65                      var next = NextStep(S, y, step, sqrt_step);
66
67                      S = next.Item1;
68                      y = next.Item2;
69
70                      if(min >= S)
71                          min = S;
72                  }
73
74                  paths[i] = S;
75
76                  if(ret_mins == true)
77                      mins[i] = min;
78              });
79
80              return Tuple.Create(paths, mins);
81          }
82
83          // this is used in asian arithmetic options
84          // for each time T_m we want to average on the function will yield return the current state of the paths to the
            ↪  caller
85          // to do whatever, then control is reverted to this function to continue evolving the paths
86          // the function DOES NOT return the paths at time T
87          // BE CAREFUL: second param here number of timesteps per year(per 1.0 T) and NOT total number of timesteps as in
            ↪  the above overload
88          public IEnumerable<double[]> GenPaths(int num_paths, int num_timesteps_per_year, double T, double S0, double[]
            ↪  observe_times)
89          {
90              var m = observe_times.Length;
91
92              var num_timesteps = new int[m];
93
94              // uncomment this to include path states at time T in the returned values
95              //var num_timesteps = new int[m + 1];
96
97              // how many timesteps between consecutive T_m's ?
98              // i.e. the number of timesteps between the numbers 0, T_1, T_2, ..., T_m, T
99              num_timesteps[0] = (int)(num_timesteps_per_year * observe_times[0]);
100             for(var i = 1; i < m; i++)
101                 num_timesteps[i] = (int)(num_timesteps_per_year * (observe_times[i] - observe_times[i - 1]));
102
103             // uncomment this to include time T
104             // T - T_M
105             //num_timesteps[m] = (int)(num_timesteps_per_year * (T - observe_times[m - 1]));
106
107             var paths = new double[num_paths];
108
109             var step = 1.0 / num_timesteps_per_year; // time step
110             var sqrt_step = Math.Sqrt(step);         // precalc to avoid calling sqrt all the time
111             var sqrt_v0 = Math.Sqrt(v0);             // same
112
113             for(var i = 0; i < num_timesteps.Length; i++)
114             {
115                 Parallel.For(0, num_paths, j =>
116                 {
117                     var S = S0;
118                     var y = sqrt_v0;
119
120                     // evolve the path until T_i
121                     for(var k = 0; k < num_timesteps[i]; k++)
122                     {
123                         var next = NextStep(S, y, step, sqrt_step);
124
125                         S = next.Item1;
126                         y = next.Item2;
127                     }
128
```

```
129                     paths[j] = S;
130                 });
131
132                 yield return paths; // let the caller see what we have so far before continuing
133             }
134         }
135
136         // push S and y step time forward
137         private Tuple<double, double> NextStep(double S, double y, double step, double sqrt_step)
138         {
139             var one_minus_beta_step = 1.0 - beta * step;
140             var one_minus_beta_step_square = one_minus_beta_step * one_minus_beta_step;
141             var constant = alpha * step / one_minus_beta_step;
142
143             // FIXME: we call Sample() in parallel
144             // is it thread safe or required any locks?
145             var x1 = Normal.Sample(0.0, 1.0);
146             var x2 = Normal.Sample(0.0, 1.0);
147
148             var z1 = sqrt_step * x1;
149             var z2 = sqrt_step * (rho * x1 + one_minus_sqrt_rho * x2);
150
151             var next_S = S + r * S * step + y * S * z1;
152             var temp = ((y + gamma * z2) * (y + gamma * z2)) / (4.0 * one_minus_beta_step_square) + constant;
153             var next_y = (y + gamma * z2) / (2.0 * one_minus_beta_step) + Math.Sqrt(temp);
154
155             return Tuple.Create(next_S, next_y);
156             //return new Tuple<double, double>(next_S, next_y);
157         }
158
159         private readonly double r;
160         private readonly double v0;
161         private readonly double kappa;
162         private readonly double theta;
163         private readonly double sigma;
164         private readonly double rho;
165
166         private readonly double alpha;
167         private readonly double beta;
168         private readonly double gamma;
169
170         private readonly double one_minus_sqrt_rho;
171     }
172 }
```

File: HestonCalibrator.cs

```
1   /*
2    *
3    * HestonCalibrator.cs
4    *
5    * 14/01/2018
6    *
7    * ~thwmakos~
8    */
9
10  using System;
11  using System.Collections.Generic;
12  using System.Linq;
13  using System.Text;
14  using System.Threading.Tasks;
15
16  namespace HestonModel
17  {
18      public class HestonParams
19      {
20          public double kappa;
21          public double theta;
22          public double sigma;
23          public double rho;
24          public double v0;
25
26          public const int NumParams = 5;
27      }
28
29      public struct MarketDataEntry
30      {
31          public MarketDataEntry(PayoffType _type, double _S, double _K, double _T,  double _price)
32          {
33              type  = _type;
```

```
34              S     = _S;
35              K     = _K;
36              T     = _T;
37              price = _price;
38          }
39
40          public PayoffType type;
41
42          public double S;
43          public double K;
44          public double T;
45          public double price;
46      }
47
48      // adjusted from the vasicek model calibrator class
49      public class HestonCalibrator
50      {
51
52          public HestonCalibrator(double _r0, HestonParams initial)
53          {
54              r0 = _r0;
55
56              market_data = new List<MarketDataEntry>();
57              outcome = CalibrationOutcome.NotStarted;
58
59              // this does not copy but w/e
60              initial_guess = calibrated_params = initial;
61
62          }
63
64          public double MeanSquareError(HestonFormula hf)
65          {
66              var mean_sq_error = 0.0;
67
68              foreach(var option in market_data)
69              {
70                  var model_price = (option.type == PayoffType.Call) ?
71                      hf.PriceEuropeanCallOption(option.S, option.K, option.T) : hf.PriceEuropeanPutOption(option.S,
                        ↪  option.K, option.T);
72
73                  var diff = model_price - option.price;
74                  mean_sq_error += diff * diff;
75              }
76
77              return mean_sq_error;
78          }
79
80          public void ObjectiveFunction(double[] parameters, ref double func, object obj)
81          {
82              // parameters are in the following order:
83              // kappa, theta, sigma, rho, v0
84              var hf = new HestonFormula(r0, parameters[4], parameters[0], parameters[1],
85                  parameters[2], parameters[3]);
86
87              func = MeanSquareError(hf);
88          }
89
90          public void Calibrate(double accuracy = 1.0e-2, int max_iterations = 100)
91          {
92              outcome = CalibrationOutcome.NotStarted;
93
94              double[] initial_params = new double[HestonParams.NumParams]
95              {
96                  initial_guess.kappa,
97                  initial_guess.theta,
98                  initial_guess.sigma,
99                  initial_guess.rho,
100                 initial_guess.v0
101             };
102
103             double differentation_step = 1.0e-4;
104             double stpmax = 0.5;
105
106             alglib.minlbfgsstate  state;
107             alglib.minlbfgsreport report;
108
109             // create and set up the optimizer
110             alglib.minlbfgscreatef(1, initial_params, differentation_step, out state);
111             alglib.minlbfgssetcond(state, accuracy, accuracy, accuracy, max_iterations);
112             alglib.minlbfgssetstpmax(state, stpmax);
113
114             // actually do optimize and retrieve results
115             alglib.minlbfgsoptimize(state, ObjectiveFunction, null, null);
```

```
116
117             var result_params = new double[HestonParams.NumParams];
118
119             alglib.minlbfgsresults(state, out result_params, out report);
120
121             // copy, but dont use before checking exit flag
122             calibrated_params.kappa = result_params[0]; // no memcpy in C#
123             calibrated_params.theta = result_params[1];
124             calibrated_params.sigma = result_params[2];
125             calibrated_params.rho   = result_params[3];
126             calibrated_params.v0    = result_params[4];
127             if((new int[] { 1, 2, 4}).Contains(report.terminationtype))
128             {
129                 outcome = CalibrationOutcome.FinishedOK;
130             }
131             else if ((new int[] { 5 }).Contains(report.terminationtype))
132             {
133                 outcome = CalibrationOutcome.FailedMaxItReached;
134             }
135             else
136             {
137                 outcome = CalibrationOutcome.FailedOtherReason;
138
139                 throw new ArithmeticException("Calibration failed :(");
140             }
141
142         }
143
144
145         public List<MarketDataEntry> market_data; // public so ppl can add stuff
146
147         public CalibrationOutcome outcome { get; private set; }
148         public HestonParams calibrated_params { get; private set; }
149
150         private HestonParams initial_guess;
151         private double r0; // risk free rate, not calibrated
152
153     }
154 }
```

File: HestonXLInterface.cs

```
1   using System;
2   using System.Linq;
3   using System.Collections.Generic;
4   using ExcelDna.Integration;
5   using HestonModel;
6
7   namespace HestonXL
8   {
9
10      public class HestonXLInterface
11      {
12
13          static LinkedList<string> errorMessages;
14
15
16          static HestonXLInterface()
17          {
18              errorMessages = new LinkedList<string>();
19          }
20
21          [ExcelFunction(Description = "About HestonXL function")]
22          public static string AboutHestonXL()
23          {
24              return "Heston Excel Interface for OOP with Applications 2017/18.";
25          }
26
27          [ExcelFunction(Description = "Display Error Messages for HestonXL.")]
28          public static object[,] GetLatestErrors(int number)
29          {
30              if(number <= 0)
31              {
32                  string[,] toDisplay = new string[1, 1];
33                  toDisplay[0, 0] = "GetLatestErrors: You must enter a positive number.";
34                  return toDisplay;
35              }
36              else
37              {
38                  string[,] toDisplay = new string[number, 1];
```

```
            int msgIdx = 0;
            foreach(string errorMsg in errorMessages)
            {
                toDisplay[msgIdx, 0] = errorMsg;
                ++msgIdx;
                if(msgIdx >= number)
                    break;
            }

            for(; msgIdx < number; ++msgIdx)
            {
                toDisplay[msgIdx, 0] = "";
            }
            return toDisplay;
        }
    }

    public static object HestonOneOptionPrice(
        double underlying,
        double riskFreeRate,
        double kappa,
        double theta,
        double sigma,
        double rho,
        double v0,
        double maturity,
        double strike,
        string type)
    {
        if(ExcelDnaUtil.IsInFunctionWizard()) return null;

        try
        {

            var prm = new HestonParameters
            {
                RiskFreeRate = riskFreeRate,
                V0 = v0,
                Kappa = kappa,
                Theta = theta,
                Sigma = sigma,
                Rho = rho
            };

            PayoffType call_or_put;

            if(type == "call")
                call_or_put = PayoffType.Call;
            else if(type == "put")
                call_or_put = PayoffType.Put;
            else
                throw new ArgumentException("invalid payoff");

            var opt = new EuropeanOption
            {
                Type = call_or_put,
                Maturity = maturity,
                Strike = strike
            };

            return Heston.HestonOneOptionPrice(underlying, prm, opt);

        }
        catch(Exception e)
        {
            errorMessages.AddFirst("HestonOneOptionPrice: unknown error: " + e.Message);
            return null;
        }
    }

    public static object HestonOneOptionPriceMC(double underlying,
                                        double riskFreeRate,
                                        double kappa,
                                        double theta,
                                        double sigma,
                                        double rho,
                                        double v0,
                                        double maturity,
                                        double strike,
                                        string type,
                                        int numSamplePaths,
                                        int numSteps)
    {
```

```
122
123             if(ExcelDnaUtil.IsInFunctionWizard()) return null;
124
125             try
126             {
127                 var prm = new HestonParameters
128                 {
129                     RiskFreeRate = riskFreeRate,
130                     V0 = v0,
131                     Kappa = kappa,
132                     Theta = theta,
133                     Sigma = sigma,
134                     Rho = rho
135                 };
136
137                 PayoffType call_or_put;
138
139                 if(type == "call")
140                     call_or_put = PayoffType.Call;
141                 else if(type == "put")
142                     call_or_put = PayoffType.Put;
143                 else
144                     throw new ArgumentException("invalid payoff");
145
146                 var opt = new EuropeanOption
147                 {
148                     Type = call_or_put,
149                     Maturity = maturity,
150                     Strike = strike
151                 };
152
153                 return Heston.HestonOneOptionPriceMC(underlying, prm, opt, 5000, 200 * (int)maturity);
154             }
155             catch(Exception e)
156             {
157                 errorMessages.AddFirst("HestonOneOptionPriceMC: unknown error: " + e.Message);
158                 return null;
159             }
160         }
161
162         public static object[,] CalibrateHestonParameters(object guessModelParameters,
163                                                           double riskFreeRate,
164                                                           double underlyingPrice,
165                                                           object strikes,
166                                                           object maturities,
167                                                           object type,
168                                                           object observedPrices,
169                                                           double accuracy,
170                                                           int maxIterations)
171         {
172             if(ExcelDnaUtil.IsInFunctionWizard()) return null;
173             try
174             {
175                 var strikes_array = ConvertToArray<double>(strikes);
176                 var maturities_array = ConvertToArray<double>(maturities);
177                 var types_array = ConvertToArray<string>(type).Select(t =>
178                 {
179                     switch(t)
180                     {
181                         case "put":
182                             return PayoffType.Put;
183                         case "call":
184                             return PayoffType.Call;
185                         default:
186                             throw new ArgumentException("invalid payoof type");
187                     }
188                 }).ToArray();
189
190                 var observed_prices_array = ConvertToArray<double>(observedPrices);
191
192                 if(strikes_array.Length != maturities_array.Length ||
193                     maturities_array.Length != types_array.Length ||
194                     types_array.Length != observed_prices_array.Length)
195                 {
196                     errorMessages.AddFirst("market data lengths do not match");
197                     return null;
198                 }
199
200                 var param_pairs = ConvertToKeyValuePairs(guessModelParameters);
201                 if(param_pairs.Length != 5)
202                 {
203                     errorMessages.AddFirst("guess model must be 5 key-value pairs ");
204                     return null;
```

```
205                 }
206
207                 var guess_params = new HestonParameters
208                 {
209                     RiskFreeRate = riskFreeRate
210                 };
211
212                 for(var i = 0; i < param_pairs.Length; i++)
213                 {
214                     var pair = param_pairs[i];
215
216                     if(pair.Key.Equals("kappa"))
217                         guess_params.Kappa = pair.Value;
218                     else if(pair.Key.Equals("theta"))
219                         guess_params.Theta = pair.Value;
220                     else if(pair.Key.Equals("sigma"))
221                         guess_params.Sigma = pair.Value;
222                     else if(pair.Key.Equals("rho"))
223                         guess_params.Rho = pair.Value;
224                     else if(pair.Key.Equals("v0"))
225                         guess_params.V0 = pair.Value;
226                     else
227                     {
228                         errorMessages.AddFirst("invalid pair key");
229                         return null;
230                     }
231                 }
232
233                 var data = new List<IOptionMarketData<IEuropeanOption>>();
234
235                 for(var i = 0; i < strikes_array.Length; i++)
236                 {
237                     IEuropeanOption option = new EuropeanOption
238                     {
239                         Strike = strikes_array[i],
240                         Maturity = maturities_array[i],
241                         Type = types_array[i]
242                     };
243
244                     IOptionMarketData<IEuropeanOption> d = new OptionMarketData<IEuropeanOption>
245                     {
246                         Option = option,
247                         Price = observed_prices_array[i]
248                     };
249
250                     data.Add(d);
251                 }
252
253                 var cal_result = Heston.CalibrateHestonParameters(underlyingPrice, guess_params, data, accuracy,
                    ↪  maxIterations);
254
255                 var ret = new object[7, 2];
256
257                 ret[0, 0] = "kappa"; ret[0, 1] = cal_result.Parameters.Kappa;
258                 ret[1, 0] = "theta"; ret[1, 1] = cal_result.Parameters.Theta;
259                 ret[2, 0] = "sigma"; ret[2, 1] = cal_result.Parameters.Sigma;
260                 ret[3, 0] = "rho"; ret[3, 1] = cal_result.Parameters.Rho;
261                 ret[4, 0] = "v0"; ret[4, 1] = cal_result.Parameters.V0;
262                 ret[5, 0] = "minimizer status";
263
264                 switch(cal_result.MinimizerStatus)
265                 {
266                     case CalibrationOutcome.FinishedOK:
267                         ret[5, 1] = "ok";
268                         break;
269                     case CalibrationOutcome.FailedMaxItReached:
270                         ret[5, 1] = "max iters reached";
271                         break;
272                     case CalibrationOutcome.FailedOtherReason:
273                         ret[5, 1] = "unknown fail";
274                         break;
275                 }
276
277                 ret[6, 0] = "pricing error"; ret[6, 1] = cal_result.PricingError;
278
279                 return ret;
280             }
281             catch(Exception e)
282             {
283                 errorMessages.AddFirst("CalibrateHestonParameters: unknown error: " + e.Message);
284             }
285             return null;
286         }
```

```
287
288         public static object HestonAsianOptionPriceMC(
289             double underlying,
290             double riskFreeRate,
291             double kappa,
292             double theta,
293             double sigma,
294             double rho,
295             double v0,
296             double maturity,
297             double strike,
298             object monitoringTimes,
299             string type,
300             int numSamplePaths,
301             int numSteps)
302         {
303             if(ExcelDnaUtil.IsInFunctionWizard()) return null;
304
305             try
306             {
307                 double[] observe_times = ConvertToArray<double>(monitoringTimes);
308
309                 var prm = new HestonParameters
310                 {
311                     RiskFreeRate = riskFreeRate,
312                     V0 = v0,
313                     Kappa = kappa,
314                     Theta = theta,
315                     Sigma = sigma,
316                     Rho = rho
317                 };
318
319                 PayoffType call_or_put;
320
321                 if(type == "call")
322                     call_or_put = PayoffType.Call;
323                 else if(type == "put")
324                     call_or_put = PayoffType.Put;
325                 else
326                     throw new ArgumentException("invalid payoff");
327
328                 return Heston.HestonAsianOptionPriceMC(underlying, prm, maturity, strike, observe_times, call_or_put,
                    ↪  numSamplePaths, numSteps);
329             }
330             catch(Exception e)
331             {
332                 errorMessages.AddFirst("HestonAsianOptionPriceMC error: " + e.Message);
333             }
334             return null;
335         }
336
337         public static object HestonLookbackOptionPriceMC(
338             double underlying,
339             double riskFreeRate,
340             double kappa,
341             double theta,
342             double sigma,
343             double rho,
344             double v0,
345             double maturity,
346             int numSamplePaths,
347             int numSteps)
348         {
349             if(ExcelDnaUtil.IsInFunctionWizard()) return null;
350
351             try
352             {
353                 var prm = new HestonParameters
354                 {
355                     RiskFreeRate = riskFreeRate,
356                     V0 = v0,
357                     Kappa = kappa,
358                     Theta = theta,
359                     Sigma = sigma,
360                     Rho = rho
361                 };
362
363                 return Heston.HestonLookbackOptionPriceMC(underlying, prm, maturity, 5000, 200 * (int)maturity);
364             }
365             catch(Exception e)
366             {
367                 errorMessages.AddFirst("HestonLookbackOptionPriceMC error: " + e.Message);
368             }
```

```
369             return null;
370         }
371 
372         // I copied these two funnctions for VasicekXLInterface.cs
373 
374         // Helper function: try to convert input object
375         // into specific type
376         private static T ConvertTo<T>(object In)
377         {
378             try
379             {
380                 return (T)In;
381             }
382             catch (Exception e)
383             {
384                 errorMessages.AddFirst("Could not convert object to " + typeof(T).ToString());
385                 throw e;
386             }
387         }
388         // Helper function: try to convert input object
389         // into array of specific type
390         // if the input is a 2D array then it only returns
391         // the first column
392         private static T[] ConvertToArray<T>(object In)
393         {
394             T[] V;
395             try
396             {
397                 object[] InVec;
398                 if(In.GetType() == typeof(object[]))
399                 {
400                     InVec = (object[])In;
401                     int length = InVec.GetLength(0);
402                     V = new T[length];
403                     for(int i = 0; i < length; i++)
404                     {
405                         V[i] = ConvertTo<T>(InVec[i]);
406                     }
407                     return V;
408                 }
409                 else if(In.GetType() == typeof(object[,]))
410                 {
411                     object[,] InM = (object[,])In;
412                     int rows = InM.GetLength(0);
413                     V = new T[rows];
414                     for(int i = 0; i < rows; i++)
415                     {
416                         V[i] = ConvertTo<T>(InM[i, 0]);
417                     }
418                     return V;
419                 }
420                 else
421                 {
422                     errorMessages.AddFirst("Could not convert input to array of type " + typeof(T).ToString());
423                     return null;
424                 }
425 
426 
427             }
428             catch(Exception)
429             {
430                 errorMessages.AddFirst("Could not convert input to array of type " + typeof(T).ToString());
431                 return null;
432             }
433         }
434 
435         // Helper function: try to convert input object
436         // into a key value pair of string and double
437         private static KeyValuePair<string, double>[] ConvertToKeyValuePairs(object In)
438         {
439         KeyValuePair<string, double>[] keyValPairs;
440         try
441         {
442             object[,] In2D = (object[,])In;
443             int rows = In2D.GetLength(0);
444             int cols = In2D.GetLength(1);
445             if (cols != 2)
446             {
447                 Console.WriteLine("Need two colums!");
448                 return null;
449             }
450 
451             keyValPairs = new KeyValuePair<string, double>[rows];
```

```
452
453             for (int i = 0; i < rows; i++)
454             {
455                 string key = ConvertTo<string>(In2D[i, 0]);
456                 double value = ConvertTo<double>(In2D[i, 1]);
457                 KeyValuePair<string, double> pair = new KeyValuePair<string, double>(key, value);
458                 keyValPairs[i] = pair;
459             }
460
461             return keyValPairs;
462         }
463         catch (Exception)
464         {
465             errorMessages.AddFirst("Could not create key - value pair.");
466             return null;
467         }
468
469         }
470     }
471 }
```