Report on option pricing using Heston stochastic volatility model

Tomasz Dubiel-Teleszynski

2014

Abstract

The idea behind this report is to shed light on option pricing under stochastic volatility using Heston model. However, it is not our aim to either present a complete summary of all results related to this model which are available in the literature and this is vast, or let alone implement all these available solutions. What we merely attempt to do here is to describe model dynamics to the extent necessary to understand option pricing first. Then, we show how to price plain vanilla options, such as European call option by means of Monte Carlo simulation. Next, we move on to calibrate model parameters using market data and optimization techniques under different criteria. Subsequently, we use these calibration results to price exotic options, in particular down-and-out European call option and arithmetic Asian call option.

Contents

List of Tables

List of Figures

1 Intro to Heston model

Following Veraart (2014), we use Heston model formulation under risk-neutral probability measure $\mathbb{Q}$ where dynamics for the asset price process S is following

$$\begin{aligned} dS_t &= rS_t dt + \sqrt{v_t} S_t dW_t^S \\ dv_t &= \alpha(\beta - v_t)dt + \gamma\sqrt{v_t} dW_t^v \end{aligned} \tag{1}$$

for $t \in [0, T]$, while W^S, W^v are Brownian motions with

$$dW_t^S dW_t^v = \rho dt \tag{2}$$

and $T > 0$ is maturity, $r \geq 0$ is interest rate, $v_0 > 0$ is initial squared volatility, $\alpha \geq 0$ is mean reversion rate of the squared volatility, $\beta \geq 0$ is the long-run volatility, $\gamma > 0$ is called volatility of volatility and $\rho \in [-1, 1]$ is denoted as leverage parameter.

Detailed exposition of the Heston model under real world probability measure $\mathbb{P}$ which begins with assumption that volatility $\sqrt{v_t}$ follows an Ornstein-Uhlenbeck process can be found in Heston (1993).

1.1 Square root process

In the Heston model squared volatility is a CIR or square root process (Cox et al., 1985) which is also a Feller process. Squared volatility v thus follows a homogeneous, mean reverting diffusion equation and has a non-central chi-squared distribution (Feller, 1951). We recall that

$$dv_t = \alpha(\beta - v_t)dt + \gamma\sqrt{v_t} dW_t^v \tag{3}$$

for $t \in [0, T]$, while W^v is Brownian motion.

As in (Gapeev, 2014), Feller condition states that if $2\alpha\beta > \gamma^2$ in (3) then $v_t > 0$, $\mathbb{P} - a.s.$, $\forall t \geq 0$. Furthermore, mean of squared volatility is

$$\mathbb{E}[v_t] = \beta + (v_0 - \beta)e^{-\alpha t} \tag{4}$$

and its variance

$$\mathbb{V}[v_t] = \frac{\gamma^2\beta}{2\alpha}(1 - e^{-\alpha t})^2 + \frac{\gamma^2}{\alpha} v_0 e^{\alpha t}(1 - e^{-\alpha t}) \tag{5}$$

however for small t it holds that $\mathbb{V}[v_t] \simeq \gamma^2 t v_0$, whereas for large t variance $\mathbb{V}[v_t]$ converges to

$\frac{\gamma^2\beta}{2\alpha}$.

1.2 Euler approximation

As in (Veraart, 2014), we adapt full truncation Euler scheme for log-asset price in the Heston model. Let $\log(\hat{S})$ and $\hat{v}$ denote approximations of $\log(S)$ and v respectively, then for some $h > 0$ we consider the following approximation to (1)

$$\begin{aligned} \log(\hat{S}_{t+h}) &= \log(\hat{S}_t) + (r - \tfrac{1}{2}\hat{v}_t^+)h + \sqrt{\hat{v}_t^+}Z_S\sqrt{h} \\ \hat{v}_{t+h} &= \hat{v}_t + \alpha(\beta - \hat{v}_t^+)h + \gamma\sqrt{\hat{v}_t^+}Z_v\sqrt{h} \end{aligned} \tag{6}$$

where $\log$ is natural logarithm, Z_S and Z_v are correlated, standard normally distributed random variables and $\hat{v}_t^+ = \max(\hat{v}_t, 0)$.

We simulate Z_S and Z_v sampling two independent standard normally distributed random variables Z_1 and Z_2, setting $Z_S = \rho Z_1 + \sqrt{1-\rho^2}Z_2$ and $Z_v = Z_1$, by means of a normal sampler provided by Veraart (2014).

1.3 European call option

Monte Carlo simulation serves here to obtain time 0 price $C_0(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho)$ of a European call option under stochastic volatility using Heston model. European call option has payoff $(S_T - K)^+$ which depends on asset price S_T at time T and strike price K.

According to theory of risk-neutral option pricing

$$C_0(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho) = e^{-rT}\mathbb{E}[(S_T - K)^+] \tag{7}$$

where expectation $\mathbb{E}[\cdot]$ is calculated under risk-neutral probability measure $\mathbb{Q}$ (Lokka and Zervos, 2013).

In order to obtain Monte Carlo estimator of (7) we need asset price S at time T from the Heston model. Using Euler scheme in (6) with m steps, what means that $h = \frac{T}{m+1}$ in (6), we can simulate n sample paths and only store terminal values (Glasserman, 2004).

Monte Carlo estimator of a European call option price is of the form

$$\hat{C}_0(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; m, n) = e^{-rT}\frac{1}{n}\sum_{i=1}^{n}(\hat{S}_T^i - K)^+ \tag{8}$$

We choose $m = 1000$ and $n = 100000$ and for $r = 0$, $S_0 = 10$, $K = 10$, $T = 1$, $v_0 = 0.04$, $\alpha = 2$, $\beta = 0.04$, $\gamma = 0.2$ and $\rho = -0.2$ we obtain following time 0 price of European call option $\hat{C}_0(0, 10, 10, 1; 0.04, 2, 0.04, 0.2, -0.2; 1000, 100000) = 0.7859$, with asymptotic 95% confidence interval $[0.7759, 0.7959]$.

2 Calibration to market

Next, we consider the problem of calibrating Heston model to market data, which involves $l \in \mathbb{N}$ empirical prices p_1^m, ..., p_l^m of traded European call options with maturities T_1, ..., T_l and strike prices K_1, ..., K_l.

2.1 Optimization criteria

Veraart (2014) suggests to calibrate Heston model to market data minimizing over v_0, α, β, γ and ρ the following function

$$\sum_{j=1}^{l} \left[p_j^m - C_0(r, S_0, K_j, T_j; v_0, \alpha, \beta, \gamma, \rho)\right]^2 \tag{9}$$

Instead of sum of squared errors we also propose to minimize sum of absolute deviations

$$\sum_{j=1}^{l} \left|p_j^m - C_0(r, S_0, K_j, T_j; v_0, \alpha, \beta, \gamma, \rho)\right| \tag{10}$$

similarly over v_0, α, β, γ and ρ what leads to more robustness to outliers among other things. It is especially of interest if outliers should be effectively and can be safely ignored. On the other hand, solution resulting from minimizing sum of squared errors is more stable. It means that for small data adjustment solution also moves only slightly, unlike in case of using sum of absolute deviations (Schlossmacher, 1973).

For instance, when certain European call options with particular maturities are not liquid, empirical market prices may not precisely reflect their actual value. In such case we would like their influence on calibration to market to be minimal. If it is not convenient to identify these options and remove them from the data set manually since the latter is large, choosing sum of absolute deviations as minimization criterion is a better idea than sum of squared errors.

Mikhailov and Nogel (2003) suggest to calibrate Heston model to market data minimizing over v_0, α, β, γ and ρ criterion of the form

$$\sum_{j=1}^{l} w_j \left[p_j^m - C_0(r, S_0, K_j, T_j; v_0, \alpha, \beta, \gamma, \rho)\right]^2 \tag{11}$$

where we deliberately ignored penalty term, discussed later.

We can also introduce weights w_j, $j = 1, ..., l$ into (10) to obtain

$$\sum_{j=1}^{l} w_j \left|p_j^m - C_0(r, S_0, K_j, T_j; v_0, \alpha, \beta, \gamma, \rho)\right| \tag{12}$$

Although these weights can be used to control influence of potentially outlying prices of non liquid options on calibration results, their suitable choice is essential for satisfying performance. Since we do not have any prior knowledge about weights w_j, $j = 1, ..., l$, we do not pursue using these two criteria any further.

2.2 Parameter constraints

Heston model parameters over which minimization is carried out, in particular v_0, α, β, γ and ρ of the square root process, are bounded by constraints of the form

- $v_0 > 0$
- $\alpha \geq 0$
- $\beta \geq 0$
- $\gamma > 0$
- $\rho \in [-1, 1]$

We tackle first four constraints with exponential transformation

$$y = e^x \quad \Leftrightarrow \quad x = \log(y) \tag{13}$$

and the last one with modified logistic transformation found for example on Wikipedia

$$y = \frac{1 - e^{-x}}{1 + e^{-x}} \quad \Leftrightarrow \quad x = -\log\left(\frac{1 - y}{1 + y}\right) \tag{14}$$

what necessitates no further explanation.

Apart from these constraints above we deal with Feller condition by means of regularization which introduces additional stability into calibration process (Mikhailov and Nogel, 2003). Namely, we enrich objective functions (9) and (10) with a penalty term in the following way

$$\sum_{j=1}^{l} \left[p_j^m - C_0(r, S_0, K_j, T_j; v_0, \alpha, \beta, \gamma, \rho)\right]^2 + \kappa(\gamma^2 - 2\alpha\beta)^+ \tag{15}$$

and

$$\sum_{j=1}^{l} \left|p_j^m - C_0(r, S_0, K_j, T_j; v_0, \alpha, \beta, \gamma, \rho)\right| + \kappa(\gamma^2 - 2\alpha\beta)^+ \tag{16}$$

where κ is a constant penalty term multiplier which we arbitrarily choose to be very large, in particular $\kappa = 10^6$, a usual move in regularization.

2.3 Simple(x) procedures

We then choose to minimize objective functions (15) and (16) using simplex algorithm by Nelder and Mead (1965) which is available in GNU Scientific Library (Galassi et al., 2013), henceforth GSL, suggested by Veraart (2014).

Although there are different strategies to perform minimization of an arbitrary k-dimensional function, embedded in GSL, we use this particular algorithm since it does not require gradient of the objective function, which is not immediately available, if at all.

What theoretically Nelder-Mead simplex algorithm does is following. Namely, it maintains $k + 1$ trial parameter vectors in form of vertices of a k-dimensional simplex. In each iteration it attempts to improve the worst vertex of the simplex via geometrical operations. Iterations are carried forward until overall size of the simplex is below specified tolerance level tol, which we arbitrarily choose to be $tol = 10^{-6}$.

From the practical viewpoint, selected algorithm uses general framework provided for minimization procedures available in GSL. User provides input to individual functions necessary for each of the steps in the algorithm. Three main stages of single iteration are distinguished

- user initializes minimizer state s for algorithm T
- state s is updated by iterating algorithm T
- convergence test on state s against tol

and iteration is then repeated if necessary (Galassi et al., 2013). These phases are reflected in the C/C++ code appended to this project report.

Most evident caveat of multidimensional minimization methods which are to be found in GSL, including Nelder-Mead simplex algorithm, is that they only search for one local minimum at a time and if several local minima exist in the search region the first identified minimum is returned and no procedure exists to find out if this is global minimum of the objective function being minimized (Galassi et al., 2013).

In order to cope with this disadvantage we try different sets of starting parameter values. After some experimentation, we decide to continue with parameter values provided by Veraart (2014) to initially price European call option above, as our starting parameter values, cap number of minimizer's iterations to 2000 and set initial step size for all parameters equal to 1.

We also identify problem-specific caveat of chosen algorithm and Monte Carlo pricing in general. In particular, we observe that using Monte Carlo estimator of European call option price $\hat{C}_0(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; m, n)$ in (8) as input to objective function (15) or (16), minimization becomes computationally prohibitive for large m and n, that is if it is to be carried out at an

acceptable level of numerical accuracy and we do pick $m = 1000$ and $n = 100000$ in the end. We thus refrain from using method of dependent sampling or quasi-Monte Carlo suggested by Eichler et al. (2011).

To overcome this disadvantage we resort to closed-form solution to the Heston model for European call option price (Heston, 1993; Mikhailov and Nogel, 2003). We implement equations of this solution one by one in C++ as they are stipulated in Heston (1993), but omit reproducing them here since they are both well reflected in appended code and also provide no novel insights at this stage. Infinite integral necessary to compute on the way to closed-form solution is tackled by numerical integration following commonly used trapezoidal rule.

This way of overcoming our computational difficulty introduces new parameter to the problem at hand, namely volatility risk premium λ. Although we could for simplicity set it fixed equal to zero as in Mikhailov and Nogel (2003), we proceed differently since it is claimed to be nonzero for instance for equity options (Lamoureux and Lastrapes, 1993) and decide to retrieve it from European call option price obtained via Monte Carlo estimator (8) with parameter values provided by Veraart (2014).

Using bisection method, details of which are omitted here, we arrive at a negative estimate of volatility risk premium $\hat{\lambda} = -0.0083$ and keep it fixed when pricing all other derivatives throughout the rest of this exposition. Such a negative value of λ is expected in practice and justified in the literature (Heston, 1993).

Now we are in a position to ultimately reformulate our objective functions (15) and (16) to their final form

$$\sum_{j=1}^{l} \left[p_j^m - \bar{C}_0(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; \hat{\lambda}) \right]^2 + \kappa(\gamma^2 - 2\alpha\beta)^+ \tag{17}$$

labeled SSE and

$$\sum_{j=1}^{l} \left| p_j^m - \bar{C}_0(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; \hat{\lambda}) \right| + \kappa(\gamma^2 - 2\alpha\beta)^+ \tag{18}$$

labeled SAD, where in both cases $\bar{C}_0(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; \hat{\lambda})$ is time 0 price of European call option under Heston model resulting from closed-form solution.

2.4 Empirical calibration

In tables 1 and 2, for $r = 0$, $T = 1$ and $S_0 = 14.97$ we present calibration results for parameters v_0, α, β, γ and ρ in the Heston model using market data provided by Veraart (2014). We also report after how many iterations (#) Nelder-Mead simplex algorithm converges for a given criterion, as well as show values of minimized objective functions, that is SSE and SAD respectively.

#	$\hat{v}_0$	$\hat{\alpha}$	$\hat{\beta}$	$\hat{\gamma}$	$\hat{\rho}$	SSE
787	**0.1082**	**1.0953**	**0.1344**	**0.5424**	**-0.5679**	0.015302

Table 1: Calibration to market using SSE

#	$\hat{v}_0$	$\hat{\alpha}$	$\hat{\beta}$	$\hat{\gamma}$	$\hat{\rho}$	SAD
280	**0.0616**	**0.8210**	**0.2340**	**0.6199**	**-0.4814**	0.135118

Table 2: Calibration to market using SAD

We observe that SAD criterion leads to faster convergence of chosen minimization algorithm in our problem formulation, that is after 280 iterations compared to 787 iterations in case of SSE, however minimized value of objective function is higher in the former case.

None of the sets of parameter estimates violates Feller condition. Although each criterion delivers visibly different parameter calibration results, Heston model fit to market data expressed via root-mean-square error ($RMSE$) between European call option market prices p^m and corresponding prices $\hat{C}_0$ based on calibrated parameters, is comparable for both objective functions. Respective prices and their asymptotic 95% confidence intervals are shown in tables 3 and 4.

In particular, we obtain $RMSE_{SSE} = 0.0458$ and $RMSE_{SAD} = 0.0456$ what only marginally lets us be in favor of criterion based on sum of absolute deviations, when neglecting speed of convergence for the moment. In passing, this difference of 0.0002 in root-mean-square errors results from misalignment vis-a-vis market prices of European call option prices after calibration using SSE which are out-of-the money at time 0.

K	10.0	12.5	15.0	17.5	20.0	22.5	25.0
$95\%U$	5.4319	3.4499	1.9300	0.9297	0.4071	0.1660	0.0642
p^m	5.4300	3.3500	1.8500	0.9000	0.4000	0.1800	0.0900
$\hat{C}_0$	**5.3184**	**3.3663**	**1.8773**	**0.9022**	**0.3940**	**0.1603**	**0.0619**
$95\%L$	5.2048	3.2828	1.8246	0.8747	0.3810	0.1547	0.0596

Table 3: European call option prices after SSE calibration

K	10.0	12.5	15.0	17.5	20.0	22.5	25.0
$95\%U$	5.4249	3.4166	1.9076	0.9216	0.4173	0.1835	0.0862
p^m	5.4300	3.3500	1.8500	0.9000	0.4000	0.1800	0.0900
$\hat{C}_0$	**5.3114**	**3.3314**	**1.8514**	**0.8915**	**0.4018**	**0.1762**	**0.0824**
$95\%L$	5.1979	3.2462	1.7952	0.8613	0.3863	0.1690	0.0786

Table 4: European call option prices after SAD calibration

Closer look at asymptotic 95% confidence intervals reported in tables 3 and 4 provides one more

argument, beside higher speed of convergence of Nelder-Mead simplex algorithm and better Heston model fit to market data, seemingly in favor of SAD criterion. Namely, in SAD case market prices p^m fall into these intervals for European call options which are non-negligibly further in-the-money and out-of-the-money at time 0 compared to what we observe in SSE case.

Graphical reinvestigation of calibration results from tables 3 and 4 can also be done for time 0 "almost"-at-the-money and in-the-money European call options by viewing figure 1 and for time 0 "almost"-at-the-money and out-of-the-money European call options by viewing figure 2, in case of SSE, as well as for time 0 "almost"-at-the-money and in-the-money European call options by viewing figure 3 and for time 0 "almost"-at-the-money and out-of-the-money European call options by viewing figure 4, in case of SAD, respectively.

Here "almost"-at-the-money is a reference to the case when $K = 15$ and we have $S_0 = 14.97$.

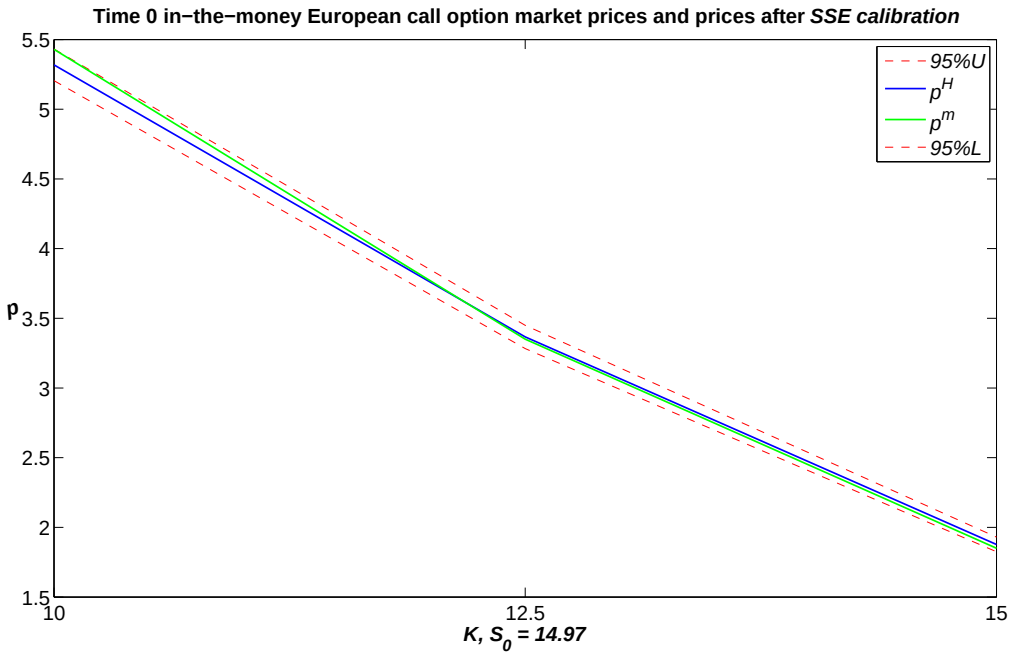

Figure 1: In-the-money European call option market prices and prices after SSE calibration

3 Pricing exotic options

In the following we focus on Monte Carlo pricing of barrier and Asian options. In particular, we concentrate our attention on down-and-out European call option and on arithmetic Asian call option.

After stating their payoffs and describing Monte Carlo estimators for their respective time 0 prices under stochastic volatility using Heston model, we apply parameter calibration results obtained above for European call options with market data provided by Veraart (2014) and these two exotics. We keep $m = 1000$ and $n = 100000$ fixed as stipulated before.

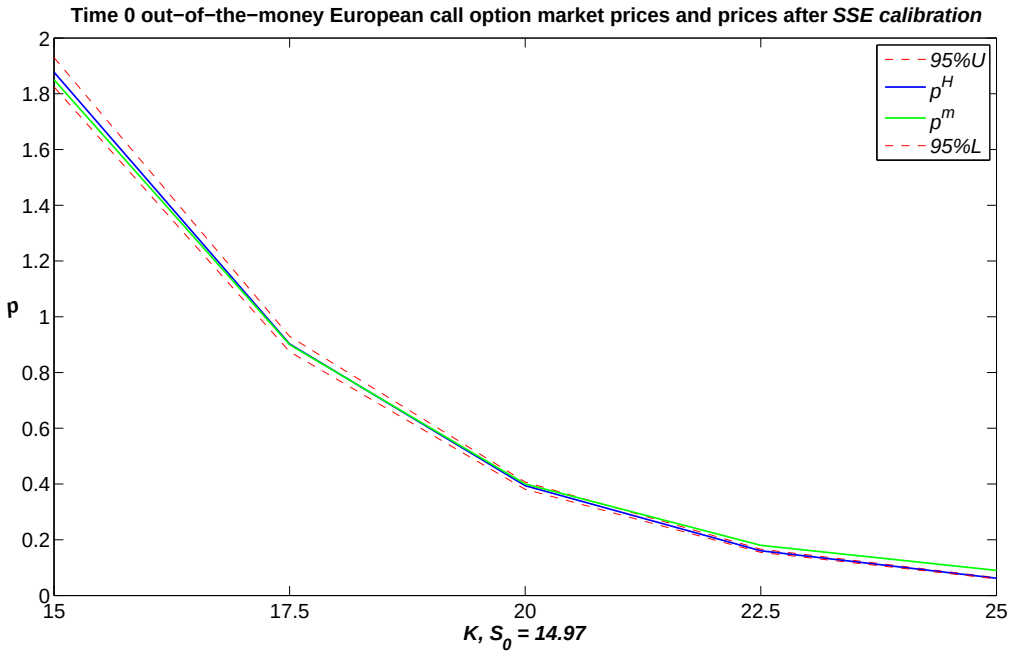

Figure 2: Out-of-the-money European call option market prices and prices after SSE calibration

3.1 Down & out

Monte Carlo simulation serves to obtain time 0 price $C_0^{D\&O}(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; B)$ of a down-&-out European call option with barrier B under stochastic volatility using Heston model.

Down-&-out European call option with barrier B has payoff $I_{(\underline{S}_T > B)} \times (S_T - K)^+$ which not only depends on asset price S at time T and strike price K but also on barrier B and running minimum $\underline{S}_T = \min_{0 \leq t \leq T} S_t$, hence it depends on entire path of S.

According to risk-neutral option pricing theory

$$C_0^{D\&O}(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; B) = e^{-rT} \mathbb{E}[I_{(\underline{S}_T > B)} \times (S_T - K)^+] \tag{19}$$

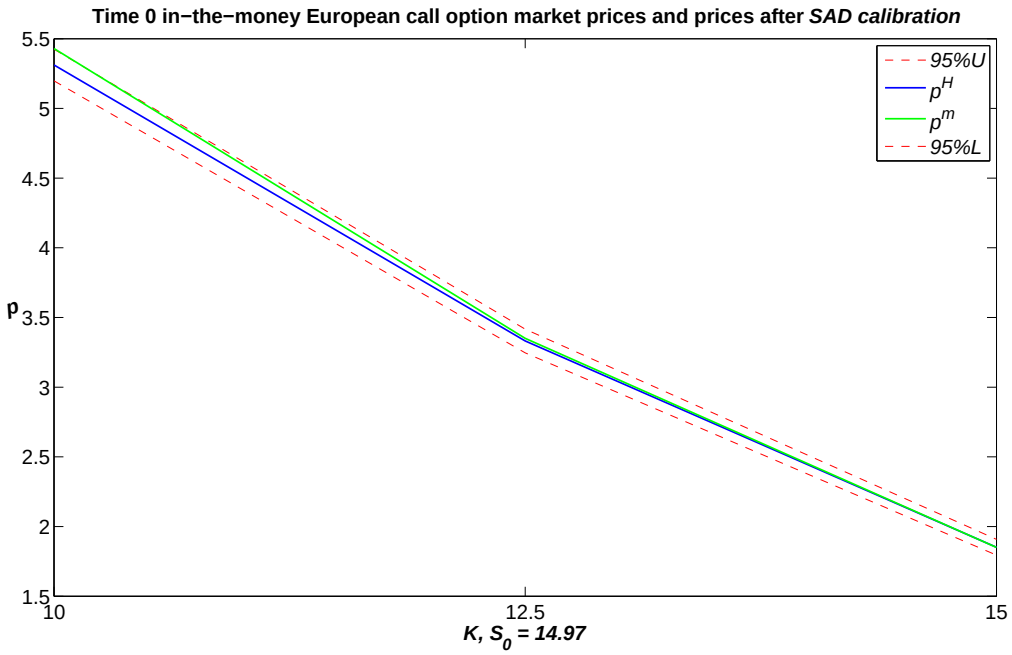

Figure 3: In-the-money European call option market prices and prices after SAD calibration

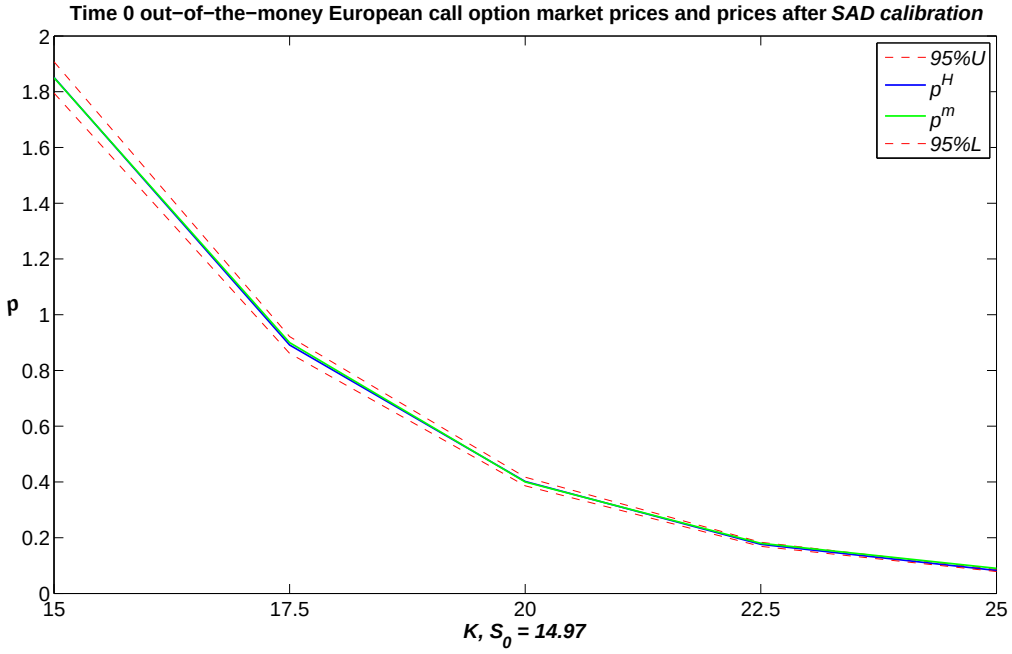

Figure 4: Out-of-the-money European call option market prices and prices after SAD calibration

where $I_{(\cdot)}$ is usual indicator function and expectation $\mathbb{E}[\cdot]$ is calculated under risk-neutral probability measure $\mathbb{Q}$ (Lokka and Zervos, 2013).

So as to obtain Monte Carlo estimator of (19) we need the entire path of asset price S from the Heston model. Using Euler scheme with m steps we can simulate n sample paths and store them.

Monte Carlo estimator of time 0 price of down-&-out European call option price with barrier B is then following

$$\hat{C}_0^{D\&O}(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; B; m, n) = e^{-rT} \frac{1}{n} \sum_{i=1}^{n} (\hat{\underline{S}}_T^i - B)^+ \times (\hat{S}_T^i - K)^+ \qquad (20)$$

With parameter calibration results at hand, for $r = 0$, $T = 1$ and $S_0 = 14.97$ and strikes stated by Veraart (2014) we arrive at time 0 prices of down-&-out European call options with barrier arbitrarily chosen to be $B = 9$, which we present in tables 5 and 6 for SSE and SAD cases respectively. On top of option prices asymptotic 95% confidence intervals are reported as well.

K	10.0	12.5	15.0	17.5	20.0	22.5	25.0
$95\%U$	5.3159	3.4290	1.9243	0.9214	0.3924	0.1632	0.0683
$\hat{C}_0^{D\&O}$	**5.1968**	**3.3437**	**1.8719**	**0.8948**	**0.3801**	**0.1575**	**0.0658**
$95\%L$	5.0777	3.2584	1.8195	0.8683	0.3679	0.1519	0.0633

Table 5: Down-&-out European call option prices after SSE calibration

K	10.0	12.5	15.0	17.5	20.0	22.5	25.0
$95\%U$	5.3092	3.3989	1.9309	0.9231	0.4028	0.1587	0.0687
$\hat{C}_0^{D\&O}$	**5.1907**	**3.3152**	**1.8783**	**0.8957**	**0.3901**	**0.1536**	**0.0662**
$95\%L$	5.0722	3.2316	1.8257	0.8683	0.3775	0.1484	0.0637

Table 6: Down-&-out European call option prices after SAD calibration

3.2 Asian option

We use Monte Carlo simulation to obtain time 0 price $C_0^A(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho)$ of an arithmetic Asian call option under Heston stochastic volatility model.

Since we consider discrete monitoring case, arithmetic Asian call option has payoff $(\frac{1}{m}\sum_{j=1}^{m} S_{t_j} - K)^+$ which depends on S_{t_j}, $j = 0, ..., m$, that is on m points from the entire path of S.

According to risk-neutral option pricing theory

$$C_0^A(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho) = e^{-rT}\mathbb{E}[(\frac{1}{m}\sum_{j=1}^{m} S_{t_j} - K)^+] \tag{21}$$

where expectation $\mathbb{E}[\cdot]$ is calculated under risk-neutral probability measure $\mathbb{Q}$ (Veraart, 2014).

To obtain Monte Carlo estimator of (21) we thus need entire path of asset price S from the Heston model. Using Euler scheme with m steps we simulate n sample paths again and store them.

Monte Carlo estimator of time 0 price of an arithmetic Asian call option price has the form as in

$$\hat{C}_0^A(r, S_0, K, T; v_0, \alpha, \beta, \gamma, \rho; m, n) = e^{-rT}\frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{m}\sum_{j=1}^{m}\hat{S}_{t_j}^i - K\right)^+ \tag{22}$$

With parameter calibration results available, for $r = 0$, $T = 1$ and $S_0 = 14.97$ and strikes specified by Veraart (2014) we obtain time 0 prices of arithmetic Asian call options which we present in tables 7 and 8 for SSE and SAD criteria respectively. Furthermore, we report asymptotic 95% confidence intervals for these prices as well.

K	10.0	12.5	15.0	17.5	20.0	22.5	25.0
$95\%U$	5.0444	2.7926	1.1124	0.2792	0.0458	0.0067	0.0011
$\hat{C}_0^A$	**4.9991**	**2.7584**	**1.0959**	**0.2749**	**0.0451**	**0.0065**	**0.0011**
$95\%L$	4.9537	2.7243	1.0795	0.2706	0.0443	0.0064	0.0011

Table 7: Asian call option prices after SSE calibration

K	10.0	12.5	15.0	17.5	20.0	22.5	25.0
$95\%U$	5.0315	2.7159	0.9840	0.2001	0.0320	0.0061	0.0017
$\hat{C}_0^A$	**4.9941**	**2.6871**	**0.9707**	**0.1971**	**0.0315**	**0.0060**	**0.0017**
$95\%L$	4.9567	2.6584	0.9573	0.1940	0.0309	0.0059	0.0016

Table 8: Asian call option prices after SAD calibration

References

Cox, J. C., Ingersoll Jr, J. E. and Ross, S. A. (1985). A theory of the term structure of interest rates. Econometrica: Journal of the Econometric Society, 385-407.

Eichler, A., Leobacher, G. and Zellinger, H. (2011). Calibration of financial models using quasi-Monte Carlo. Monte Carlo Methods and Applications, 17(2), 99-131.

Feller, W. (1951). Two singular diffusion problems. Annals of Mathematics, 173-182.

Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., Rossi, F. and Ulerich, R. (2013). GNU Scientific Library Reference Manual. Version 1.16, Network Theory Ltd.

Gapeev, P. (2014). MA416 lecture notes and class materials, LSE.

Glasserman, P. (2004). Monte Carlo methods in financial engineering (Vol. 53). Springer.

Heston, S. L. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options. Review of Financial Studies, 6(2), 327-343.

Lamoureux, C. G. and Lastrapes, W. D. (1993). Forecasting stock-return variance: Toward an understanding of stochastic implied volatilities. Review of Financial Studies, 6(2), 293-326.

Lokka, A. and Zervos, M. (2013). MA415 lecture notes and class materials, LSE.

Mikhailov, S. and Nogel, U. (2003). Heston's Stochastic Volatilty Model: Implementation, Calibration, and some Extensions. Wilmott Magazine, July, 74-79.

Nelder, J. A. and Mead, R. (1965). A simplex method for function minimization. Computer Journal, 7(4), 308-313.

Schlossmacher, E. J. (1973). An iterative technique for absolute deviations curve fitting. Journal of the American Statistical Association, 68(344), 857-859.

Veraart, L. A. M. (2014). MA417 lecture notes, class materials and lab codes, LSE.

4 C++ code

We omit here C++ code used to obtain normal samples provided by Veraart (2014) and begin with the main.cpp file. Any particular ordering of files which may result is not intended. To test the entire C++ code found below please open **Project.dev** file and after decreasing the value of variable m to $m = 100$ at the very beginning of the main.cpp file to save time run it, however doing so numerical results, though consistent with these here, will be different.

main.cpp

```
//////////////////////////////////////////////////

// paths at the LSE ---------------------------------

//Q:/A20010_BLOODSHED_4992/Dev-Cpp/lib/libgsl.a
//Q:/A20010_BLOODSHED_4992/Dev-Cpp/lib/libgslcblas.a

//////////////////////////////////////////////////

//////////////////////////////////////////////////

// paths  at home ----------------------------------

//C:/Dev-Cpp/lib/libgsl.a
//C:/Dev-Cpp/lib/libgslcblas.a

//////////////////////////////////////////////////

#include <cstdlib>                // EXIT_SUCCESS, EXIT_FAILURE
#include <vector>                 // vector
#include <iostream>               // cout
#include <stdio.h>                // writing output file
#include <cmath>                  // exp, fabs
#include <gsl/gsl_multimin.h>     // Simplex algorithm of Nelder and Mead
#include "EuropeanCall_SV_MC.h" // European call option under Heston
    stochastic
                                  // volatility model priced using Monte
                                     Carlo
#include "EuropeanCallDownAndOut_SV_MC.h" // Down-and-out European call
    option
                                  // under Heston stochastic volatility model
                                  // priced using Monte Carlo
#include "AsianCall_SV_MC.h"      // Asian call option under Heston
    stochastic
                                  // volatility model priced using Monte
                                     Carlo
#include "VolRiskClass.h"         // volatility of risk base class
#include "VolRiskPrice.h"         // volatility of risk price derived class

using namespace std;

// declare functions used here which are defined in functions.cpp
int minf(vector<double>& calvals, double *fpar, double *stvals);
int Feller(double alpha, double beta, double gamma);
double rmsef(double *prices, double *pricesCAL, int n);
double Bisection(VolRiskClass *FCptr, double a, double b, double c, double
    eps);
```

```
//////////////////////////////////////////////////////////////////////////////
// TO TEST ENTIRE CODE DECREASE "m" (FIRST VARIABLE BELOW) TO "m = 100"
    FOR ////
// SPEED, HOWEVER IT WILL NOT AFFECT SPEED OF NUMERICAL INTEGRATION WITHIN
    /////
// MINIMIZATION ONLY SIMULATING PATHS THUS OPTIMIZATION TIME REMAINS SAME
    //////
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////
// MAIN PROGRAM
    ////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
int main(void)
{
  //////////////////////////////////////////////////////////////////////////////
  // PROBLEM 1
      ////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////////////////////

  // declare and specify Euler scheme and Monte Carlo simulation parameters
  // number of Euler scheme steps
  int m = 1000;
  // number of Monte Carlo simulations
  int n = 100000;

  // declare and specify general parameters for pricing European call
      option
  double r  =  0.0;
  double S0 = 10.0;
  double K  = 10.0;
  double T  =  1.0;  // equivalent to 1 year

  // declare and specify parameters for Heston stochastic volatility model
  double v0    =  0.04;
  double alpha =  2.0;
  double beta  =  0.04;
  double gamma =  0.2;
  double rho   = -0.2;
  // check Feller condition for above start values
  if(Feller(alpha,beta,gamma))
  {
      cout << "Feller condition not met by specified parameters!" <<  endl;
      system("PAUSE");
      return EXIT_FAILURE;
  }

  // instantiate European call option under stochastic volatility using
      Monte
  // Carlo pricer class
  EuropeanCall_SV_MC ecmcsv(n);

  // European call option price
  double *ecmcsvp = ecmcsv.getprice(r,alpha,beta,gamma,rho,v0,S0,K,T,m);

  // open file
  FILE *pricingEC = fopen("PricingEC.txt","w");
```

```
// screen output
printf("European call option price:\n");
printf("LCIB: %.4f PRICE: %.4f UCIB: %.4f\n",ecmcsvp[0],ecmcsvp[1],
    ecmcsvp[2]);
// file output
fprintf(pricingEC,"European call option price:\n");
fprintf(pricingEC,"LCIB: %.4f PRICE: %.4f UCIB:
    %.4f\n",ecmcsvp[0],ecmcsvp[1],
    ecmcsvp[2]);
// close file
fclose(pricingEC);

////////////////////////////////////////////////////////////////////////////
// PROBLEM 3
    ////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

// problem resultant from the fact that later we used closed form
    solution
// to Heston stochastic volatility model for European call option price
// hence we need to calculate price of volatility risk (volatility risk
// premium) using bisection method and reference price obtained in
    problem 1
double refprice = ecmcsvp[1];
// VolRiskPrice class instantiation
VolRiskPrice vrp(r,alpha,beta,gamma,rho,v0,S0,K,T);
VolRiskPrice *VRPptr = &vrp;
// specify tolerance
double eps = 0.0000000000000001;
// specify bounds for bisection method
double lbound = -5;
double ubound =  5;
// calculate volatility risk premium using bisection method
double lambda =  Bisection(VRPptr,lbound,ubound,refprice,eps);

// open file
FILE *volriskpr = fopen("VolRiskPremium.txt","w");
// terminal output ----------------------
printf("Price of volatility risk (volatility risk premium): 
    %.4f\n",lambda);
// file output
fprintf(volriskpr,"Price of volatility risk (volatility risk premium): 
    %.4f\n"
    ,lambda);
// close file
fclose(volriskpr);

// specify initial stock value for pricing European call option which is
// fixed in minimization and differs from this specified above in
    problem 1
S0 = 14.97;

// specify minimization criterion (1 for SSE and 0 for SAD)
double cr = 1;

// collect general parameters for pricing European call option which are
// fixed in minimization and minimization criterion in double array
double fpar[7] = {T, S0, r, (double)m, (double)n, lambda, cr};
```

```
// declare and specify start values for Heston stochastic volatility
    model,
// apply logarithmic transormation due to exponentiation in objective
    function
// for convenience choose corresponding values from problem 1
double v00    =  log(v0);
double alpha0 =  log(alpha);
double beta0  =  log(beta);
double gamma0 =  log(gamma);
double rho0   = -log(2.0/(rho+1.0)-1);
// check Feller condition for above start values
if(Feller(exp(alpha0), exp(beta0), exp(gamma0)))
{
    cout << "Feller condition not met by start values!" <<  endl;
    system("PAUSE");
    return EXIT_FAILURE;
}

//collect above start values in a double array
double stvals[5] = {v00,alpha0,beta0,gamma0,rho0};

// calibrate to MARKET DATA using SSE criterion
// define double vector to store SSE calibration results
vector<double> calvalsSSE(5);
// begin SSE minimization
int retSSE = minf(calvalsSSE,fpar,stvals);
// check status of SSE minimization
if(retSSE)
{
   cout << "SSE minimization failed!" <<  endl;
   system("PAUSE");
   return EXIT_FAILURE;
}

// calibrate to MARKET DATA using SAD criterion
// change minimization criterion (1 for SSE and 0 for SAD)
cr = 0;
// inset change from SSE to SAD into double array of fixed parameters
fpar[6] = cr;
// define double vector to store SAD calibration results
vector<double> calvalsSAD(5);
// begin SAD minimization
int retSAD = minf(calvalsSAD,fpar,stvals);
// check status of SAD minimization
if(retSAD)
{
   cout << "SAD minimization failed!" <<  endl;
   system("PAUSE");
   return EXIT_FAILURE;
}

// calculate post-SSE-calibration European call option prices
// assign SSE-calibrated parameter values to variables
double _v0    = calvalsSSE[0];
double _alpha = calvalsSSE[1];
double _beta  = calvalsSSE[2];
double _gamma = calvalsSSE[3];
```

```
double _rho    = calvalsSSE[4];
// check Feller condition for above SSE-calibrated parameter values
if(Feller(_alpha, _beta, _gamma))
{
    cout << "Feller condition not met by SSE-calibrated values!" <<
        endl;
    system("PAUSE");
    return EXIT_FAILURE;
}

// MARKET DATA ------------------------------------------------
// constant number of market data observations
const int nStrikes = 7;
// given strikes
double strikes[nStrikes] = {10.0, 12.5, 15.0, 17.5, 20.0, 22.5, 25.0};
// given market prices
double prices[nStrikes] = {5.43, 3.35, 1.85, 0.9, 0.4, 0.18, 0.09};
// ----------------------------------------------------------

// define double array to store post-SSE-calibration European call
// option prices
double pricesSSE[nStrikes];

// open file
FILE *pricingECSSE = fopen("PricingECSSE.txt","w");
// terminal output ----------------------
printf("Post-SSE-calibration European call option prices:\n");
// file output
fprintf(pricingECSSE,"Post-SSE-calibration European call option
    prices:\n");
// loop over strikes
for(int i = 0; i<nStrikes; i++)
{
    // post-SSE-calibration European call option price
    ecmcsvp = ecmcsv.getprice(r,_alpha,_beta,_gamma,_rho,_v0,S0,
        strikes[i],T,m);
    pricesSSE[i] = ecmcsvp[1];

    // terminal output
    printf("LCIB: %.4f PRICE: %.4f UCIB: %.4f\n",ecmcsvp[0],ecmcsvp[1],
        ecmcsvp[2]);
    // file output
    fprintf(pricingECSSE,"LCIB: %.4f PRICE: %.4f UCIB:
        %.4f\n",ecmcsvp[0],
        ecmcsvp[1],ecmcsvp[2]);
}
// calculate RMSE for post-SSE-calibration prices
double rmseSSE = rmsef(prices, pricesSSE, nStrikes);
 // terminal output
printf(
    "RMSE for market and post-SSE-calibration European call option
        prices: %.4f\n",
    rmseSSE);
// file output
fprintf(pricingECSSE,
    "RMSE for market and post-SSE-calibration European call option
        prices: %.4f\n",
    rmseSSE);
```

```
// close file
fclose(pricingECSSE);

// calculate post-SAD-calibration European call option prices
// assign SAD-calibrated parameter values to variables
double _v0_    = calvalsSAD[0];
double _alpha_ = calvalsSAD[1];
double _beta_  = calvalsSAD[2];
double _gamma_ = calvalsSAD[3];
double _rho_   = calvalsSAD[4];
// check Feller condition for above SAD-calibrated parameter values
if(Feller(_alpha_,_beta_,_gamma_))
{
    cout << "Feller condition not met by SAD-calibrated values!" <<
        endl;
    system("PAUSE");
    return EXIT_FAILURE;
}

// define double array to store post-SAD-calibration European call
// option prices
double pricesSAD[nStrikes];

// open file
FILE *pricingECSAD = fopen("PricingECSAD.txt","w");
// terminal output
printf("Post-SAD-calibration European call option prices:\n");
// file output
fprintf(pricingECSAD,"Post-SAD-calibration European call option
    prices:\n");
// loop over strikes
for(int i = 0; i<nStrikes; i++)
{
    // post-SAD-calibration European call option price
    ecmcsvp = ecmcsv.getprice(r,_alpha_,_beta_,_gamma_,_rho_,_v0_,S0,
        strikes[i],T,m);
    pricesSAD[i] = ecmcsvp[1];

    // terminal output
    printf("LCIB: %.4f PRICE: %.4f UCIB: %.4f\n",ecmcsvp[0],ecmcsvp[1],
        ecmcsvp[2]);
    // file output
    fprintf(pricingECSAD,"LCIB: %.4f PRICE: %.4f UCIB:
        %.4f\n",ecmcsvp[0],
        ecmcsvp[1],ecmcsvp[2]);
}
 // calculate RMSE for post-SAD-calibration prices
double rmseSAD = rmsef(prices, pricesSAD, nStrikes);
 // terminal output
printf(
    "RMSE for market and post-SAD-calibration European call option
        prices: %.4f\n",
    rmseSAD);
// file output
fprintf(pricingECSAD,
    "RMSE for market and post-SAD-calibration European call option
        prices: %.4f\n",
    rmseSAD);
```

```
// close file
fclose(pricingECSAD);

//////////////////////////////////////////////////////////////////////////////
// PROBLEM 4
    ////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

// calculate post-SSE-calibration down-and-out European call option
    prices
// instantiate down-and-out European call option under stochastic
    volatility
// using Monte Carlo pricer class
EuropeanCallDownAndOut_SV_MC ecdaomcsv(n);
// define pointer to price from above pricer
double *ecdaomcsvp;
// define price barrier
double barr = 9;

// define double array to store post-SSE-calibration down-and-out
    European
// call option prices
double pricesSSEDAO[nStrikes];

// open file
FILE *pricingSSEDAO = fopen("PricingSSEDAO.txt","w");
// terminal output
printf("Post-SSE-calibration down-and-out European call option
    prices:\n");
// file output
fprintf(pricingSSEDAO,
    "Post-SSE-calibration down-and-out European call option prices:\n");
// loop over strikes
for(int i = 0; i<nStrikes; i++)
{
    // post-SSE-calibration down-and-out European call option price
    ecdaomcsvp =
        ecdaomcsv.getprice(barr,r,_alpha,_beta,_gamma,_rho,_v0,S0,
         strikes[i],T,m);
    pricesSSEDAO[i] = ecdaomcsvp[1];

    // terminal output
    printf("LCIB: %.4f PRICE: %.4f UCIB:
        %.4f\n",ecdaomcsvp[0],ecdaomcsvp[1],
         ecdaomcsvp[2]);
    // file output
    fprintf(pricingSSEDAO,"LCIB: %.4f PRICE: %.4f UCIB:
        %.4f\n",ecdaomcsvp[0],
         ecdaomcsvp[1],ecdaomcsvp[2]);
}
// close file
fclose(pricingSSEDAO);

// calculate post-SAD-calibration down-and-out European call option
    prices
// define double array to store post-SAD-calibration down-and-out
    European
// call option prices
```

```
double pricesSADDAO[nStrikes];

 // open file
FILE *pricingSADDAO = fopen("PricingSADDAO.txt","w");
// terminal output
printf("Post-SAD-calibration down-and-out European call option
    prices:\n");
// file output
fprintf(pricingSADDAO,
    "Post-SAD-calibration down-and-out European call option prices:\n");
// loop over strikes
for(int i = 0; i<nStrikes; i++)
{
    // post-SAD-calibration down-and-out European call option price
    ecdaomcsvp =
        ecdaomcsv.getprice(barr,r,_alpha,_beta,_gamma,_rho,_v0,S0,
         strikes[i],T,m);
    pricesSADDAO[i] = ecdaomcsvp[1];

    // terminal output
    printf("LCIB: %.4f PRICE: %.4f UCIB:
        %.4f\n",ecdaomcsvp[0],ecdaomcsvp[1],
         ecdaomcsvp[2]);
    // file output
    fprintf(pricingSADDAO,"LCIB: %.4f PRICE: %.4f UCIB: %.4f\n",
         ecdaomcsvp[0],ecdaomcsvp[1],ecdaomcsvp[2]);
}
// close file
fclose(pricingSADDAO);

// calculate post-SSE-calibration Asian call option prices
// instantiate Asian call option under stochastic volatility using Monte
// Carlo pricer class
AsianCall_SV_MC acmcsv(n);
// define pointer to price from above pricer
double *acmcsvp;

// define double array to store post-SSE-calibration Asian call option
    prices
double pricesSSEAC[nStrikes];

// open file
FILE *pricingSSEAC = fopen("PricingSSEAC.txt","w");
// terminal output
printf("Post-SSE-calibration Asian call option prices:\n");
// file output
fprintf(pricingSSEAC,"Post-SSE-calibration Asian call option prices:\n");
// loop over strikes
for(int i = 0; i<nStrikes; i++)
{
    // post-SSE-calibration Asian call option price
    acmcsvp = acmcsv.getprice(r,_alpha,_beta,_gamma,_rho,_v0,S0,
         strikes[i],T,m);
    pricesSSEAC[i] = acmcsvp[1];

    // terminal output
    printf("LCIB: %.4f PRICE: %.4f UCIB: %.4f\n",acmcsvp[0],acmcsvp[1],
         acmcsvp[2]);
```

```
        // file output
        fprintf(pricingSSEAC,"LCIB:■%.4f■PRICE:■%.4f■UCIB:■
            %.4f\n",acmcsvp[0],
             acmcsvp[1],acmcsvp[2]);
    }
    // close file
    fclose(pricingSSEAC);

    // calculate post-SAD-calibration Asian call option prices
    // define double array to store post-SAD-calibration Asian call option
        prices
    double pricesSADAC[nStrikes];

    // open file
    FILE *pricingSADAC = fopen("PricingSADAC.txt","w");
    // terminal output
    printf("Post-SAD-calibration■Asian■call■option■prices:\n");
    // file output
    fprintf(pricingSADAC,"Post-SAD-calibration■Asian■call■option■prices:\n");
    // loop over strikes
    for(int i = 0; i<nStrikes; i++)
    {
        // post-SAD-calibration Asian call option price
        acmcsvp = acmcsv.getprice(r,_alpha_,_beta_,_gamma_,_rho_,_v0_,S0,
            strikes[i],T,m);
        pricesSADAC[i] = acmcsvp[1];

        // terminal output
        printf("LCIB:■%.4f■PRICE:■%.4f■UCIB:■%.4f\n",acmcsvp[0],acmcsvp[1],
            acmcsvp[2]);
        // file output
        fprintf(pricingSADAC,"LCIB:■%.4f■PRICE:■%.4f■UCIB:■
            %.4f\n",acmcsvp[0],
             acmcsvp[1],acmcsvp[2]);
    }
    // close file
    fclose(pricingSADAC);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

functions.cpp

```
#include <complex> // complex
#include <vector> // vector
#include <cmath> // exp, fabs
#include <gsl/gsl_multimin.h> // Simplex algorithm of Nelder and Mead
#include "EuropeanCall_SV_MC.h" // European call option under stochastic
                                // volatility priced using Monte Carlo
                                // simulation
#include "VolRiskPrice.h"       // volatility of risk price class

using namespace std;

// declare functions called here by functions used then in main.cpp
double realf(double phi, double alpha, double beta, double gamma,
```

```
    double rho, double lambda, double v0, double S0, double K, double r,
        double T,
    int fNum);
double trapz(vector<double> X, vector<double> Y);
double objf (const gsl_vector *v, void *params);
double getpriceCF(double r, double alpha, double beta, double gamma,
    double rho, double lambda, double v0, double S0, double K, double T);
double Bisection(VolRiskClass *FCptr, double a, double b, double c, double
    epsilon);

//
    ------------------------------------------------------------------------
// Bisection method to calculate price of volatility risk
    ----------------------
//
    ------------------------------------------------------------------------
double Bisection(VolRiskClass *FCptr, double a, double b, double c, double
    eps)
{
    double l = a;
    double r = b;
    while((r-l)>eps)
    {
        if( ( FCptr->Value(l)-c ) * ( FCptr->Value(0.5*(l+r))-c ) <= 0 )
        {
            l = l;
            r = 0.5*(l+r);
        }
        else
        {
            l = 0.5*(l+r);
            r = r;
        }
    }
    return l;
}

//
    ------------------------------------------------------------------------
// European call option price from Heston model in closed
    form------------------
//
    ------------------------------------------------------------------------
double getpriceCF(double r, double alpha, double beta, double gamma,
    double rho, double lambda, double v0, double S0, double K, double T)
{
    const double pi = 3.14159265358979323846264 3;

    // create partition
        double infty = 100.0;
        double step  =   0.01;
    int nPoints  = int(infty/step);

        // declare complex vectors
        vector<double> PHI(nPoints);
        vector<double>  F1(nPoints);
        vector<double>  F2(nPoints);
```

```
        // grid evaluation
    for (int j=0; j<=nPoints-1; j++)
    {
                if(j==0)
        {
            PHI[j] = 0.0000001;
        }
        else
        {
                    PHI[j]  = double(j)*step;
        }
        F1[j] = realf(PHI[j], alpha, beta, gamma, rho, lambda, v0, S0, K,
            r, T, 1);
                F2[j] = realf(PHI[j], alpha, beta, gamma, rho, lambda, v0,
                    S0, K, r, T, 2);
        }

        // integration
    double int1 = trapz(PHI, F1);
        double int2 = trapz(PHI, F2);

        // probabilities
    double P1 = 0.5 + int1/pi;
        double P2 = 0.5 + int2/pi;

        // just in case
    if(P1<0){P1=0;}
        if(P1>1){P1=1;}
    if(P2<0){P2=0;}
    if(P2>1){P2=1;}

        // return Heston price in closed form
    return S0*P1 - K*exp(-r*T)*P2;
}

//
   ------------------------------------------------------------------------
// real part of characteristic function
   ---------------------------------------
//
   ------------------------------------------------------------------------
double realf(double phi, double alpha, double beta, double gamma,
    double rho, double lambda, double v0, double S0, double K, double r,
       double T,
    int fNum)
{
    // create complex 1 and i
    complex<double>  one(1.0, 0.0);
        complex<double>    i(0.0, 1.0);

        // declare double variables
    double a = alpha*beta;
    double u;
    double b;

    // declare complex variables
        complex<double> d;
        complex<double> g;
```

```
        complex<double> C;
        complex<double> D;

        // make distinction
        if (fNum==1)
        {
        u = 0.5;
        b = alpha + lambda - rho*gamma;
    }
        else
        {
            u = -0.5;
        b = alpha + lambda;
    }

    // calculate elements of characteristic function
    d = sqrt( pow(rho*gamma*phi*i-b,2) - pow(gamma,2)*( 2*u*phi*i -
       pow(phi,2) ) );

    g = (b-rho*gamma*phi*i+d)/(b-rho*gamma*phi*i-d);

    C = r*phi*i*T + alpha*beta/pow(gamma,2)*( (b-rho*gamma*phi*i+d)*T -
        log( ((one-g*exp(d*T))/(one-g))*((one-g*exp(d*T))/(one-g)) ) );

    D =
       (b-rho*gamma*phi*i+d)/pow(gamma,2)*((one-exp(d*T))/(one-g*exp(d*T)));

    // return real part
    return real( exp( -i*phi*log(K) ) * exp( C + D*v0 + i*phi*log(S0) ) /
       (i*phi) );
}

//
   ------------------------------------------------------------------------
// numerical integration by trapeizoidal rule
   ---------------------------------
//
   ------------------------------------------------------------------------
double trapz(vector<double> X, vector<double> Y)
{
    int n = X.size();
        double sum = 0;
        for (int i=1; i<=n-1; i++)
                sum += 0.5*(X[i] - X[i-1])*(Y[i-1] + Y[i]);
        return sum;
}

//
   ------------------------------------------------------------------------
// root mean-squared error
   ------------------------------------------------
//
   ------------------------------------------------------------------------
double rmsef(double * prices, double * pricesCAL, int n)
{
    double rmse = 0.0;
    for(int i = 0; i<n; i++)
    {
```

```
        rmse += (prices[i]-pricesCAL[i])*(prices[i]-pricesCAL[i]);
    }
    rmse = rmse/n;
    return sqrt(rmse);
}

//
   ------------------------------------------------------------------------
// Feller condition check
   ------------------------------------------------
//
   ------------------------------------------------------------------------
int Feller(double alpha, double beta, double gamma)
{
    if(2*alpha*beta-gamma*gamma<0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

//
   ------------------------------------------------------------------------
// minimization objective function
   ------------------------------------------
//
   ------------------------------------------------------------------------
double objf(const gsl_vector *v, void *params)
{
  // MARKET DATA ----------------------------------------------
  // given strikes ////////////////////////////////////////
  double strikes[7] = {10.0, 12.5, 15.0, 17.5, 20.0, 22.5, 25.0};
  // given market prices //////////////////////////////////
  double prices[7] = {5.43, 3.35, 1.85, 0.9, 0.4, 0.18, 0.09};

  // cast void to double
  double *par = (double *)params;

  // allocate fixed parameter values to variables
  double T      =      par[0];
  double S0     =      par[1];
  double r      =      par[2];
     int m      = (int)par[3];
     int n      = (int)par[4];
  double lambda =      par[5];
     int cr     = (int)par[6];

  // allocate iterated parameter valies to variables
  double v0, alpha, beta, gamma, rho;
  v0    = exp(gsl_vector_get(v, 0));
  alpha = exp(gsl_vector_get(v, 1));
  beta  = exp(gsl_vector_get(v, 2));
  gamma = exp(gsl_vector_get(v, 3));
  rho   = 2.0/(1.0+exp(-gsl_vector_get(v, 4)))-1.0;
```

```
  double p;
  double crit = 0;
  for(int i =0; i<7; i++)
  {
      // calculate  closed form European call option price
      p = getpriceCF(r,alpha,beta,gamma,rho,lambda,v0,S0,strikes[i],T);

      if(cr)
      {
          // use SSE criterion
          crit += (prices[i]-p)*(prices[i]-p);
      }
      else
      {
          // use SAD criterion
          crit += fabs(prices[i]-p);
      }
  }

  // penalty term multiplier
  double penmult = 1e6;

  // penalty term
  double penalty = max(0.0,gamma*gamma-2.0*alpha*beta);

  return crit + penmult * penalty;
}

//
   ------------------------------------------------------------------------
// minimizing procedure
   ----------------------------------------------------
//
   ------------------------------------------------------------------------
int minf(vector<double>& calvals, double * fpar, double * stvals)
{
    // Simplex algorithm of Nelder and Mead instantiation
    const gsl_multimin_fminimizer_type *SNM =
        gsl_multimin_fminimizer_nmsimplex;
    gsl_multimin_fminimizer *s = NULL;
    gsl_vector *ss, *x;
    gsl_multimin_function minex_func;

    // allocate start values to variables
    double v00    = stvals[0];
    double alpha0 = stvals[1];
    double beta0  = stvals[2];
    double gamma0 = stvals[3];
    double rho0   = stvals[4];

    // collect variables with start values into GSL vector
    x = gsl_vector_alloc(5);
    gsl_vector_set(x,0,v00);
    gsl_vector_set(x,1,alpha0);
    gsl_vector_set(x,2,beta0);
    gsl_vector_set(x,3,gamma0);
    gsl_vector_set(x,4,rho0);
```

```
// set initial step size to 1 for all
ss = gsl_vector_alloc(5);
gsl_vector_set_all(ss,1);

// set up minimizer
minex_func.n = 5;
minex_func.f = objf;
minex_func.params = fpar;
s = gsl_multimin_fminimizer_alloc(SNM, 5);
gsl_multimin_fminimizer_set(s, &minex_func, x, ss);

// create output file --------------------------
FILE *calibration;
if(fpar[6])
{
    calibration = fopen("CalibrationSSE.txt","w");
    // terminal output -------------
    printf("SSE calibration:\n");
    // file output
    fprintf(calibration,"SSE calibration:\n");
    // -----------------------------
}
else
{
    calibration = fopen("CalibrationSAD.txt","w");
    // terminal output -------------
    printf("SAD calibration:\n");
    // file output
    fprintf(calibration,"SAD calibration:\n");
    // -----------------------------
}
// ---------------------------------------------

// auxiliaries
size_t iter = 0;
int status;
double size;

do
{
    // iterating
    iter++;

    // check status
    status = gsl_multimin_fminimizer_iterate(s);
    if (status)
        break;

    // check termination criterion
    size = gsl_multimin_fminimizer_size(s);
    status = gsl_multimin_test_size(size, 1e-6);

    // check status again
    if (status == GSL_SUCCESS)
    {
        // terminal output ------------------------
        printf ("converged to minimum at\n");
        // file output
```

```
            fprintf(calibration,"converged to minimum at\n");
            // ---------------------------------------
        }

        // terminal output ------------------------------------------
        printf ("%5d %.4f %.4f %.4f %.4f %.4f f() = %.6f size = %.6f\n",
                iter,
                exp(gsl_vector_get(s->x, 0)),
                exp(gsl_vector_get(s->x, 1)),
                exp(gsl_vector_get(s->x, 2)),
                exp(gsl_vector_get(s->x, 3)),
                2.0/(1+exp(-gsl_vector_get(s->x, 4)))-1,
                s->fval,
                size);
        // file output
        fprintf(calibration,
                "%5d %.4f %.4f %.4f %.4f %.4f f() = %.6f size = %.6f\n",
                iter,
                exp(gsl_vector_get(s->x, 0)),
                exp(gsl_vector_get(s->x, 1)),
                exp(gsl_vector_get(s->x, 2)),
                exp(gsl_vector_get(s->x, 3)),
                2.0/(1+exp(-gsl_vector_get(s->x, 4)))-1,
                s->fval,
                size);
        // ------------------------------------------------------------
    } // set maximum number of iterations below to 2000
    while (status == GSL_CONTINUE && iter < 2001);

    // close output file -
    fclose(calibration);
    // -------------------

    // assign calibrated parameter values to output vector
    calvals.at(0) = exp(gsl_vector_get(s->x, 0));              // v0
    calvals.at(1) = exp(gsl_vector_get(s->x, 1));              // alpha
    calvals.at(2) = exp(gsl_vector_get(s->x, 2));              // beta
    calvals.at(3) = exp(gsl_vector_get(s->x, 3));              // gamma;
    calvals.at(4) = 2.0/(1+exp(-gsl_vector_get(s->x, 4)))-1;   // rho

    // release GSL vectors and minimizer
    gsl_vector_free(x);
    gsl_vector_free(ss);
    gsl_multimin_fminimizer_free(s);

    // report final status
    return status;
}
```

Pricer_SV.h

```
#ifndef PRICER_SV_H
#define PRICER_SV_H

class Pricer_SV{

      public:
            virtual double * getprice(double r, double alpha, double beta,
```

```
                double gamma, double rho, double v0, double S0, double K,
                double T, int m) = 0;

};

#endif // PRICER_SV_H
```

EuropeanCall_SV_MC.h

```
#ifndef EUROPEANCALL_SV_MC_H
#define EUROPEANCALL_SV_MC_H

#include "Pricer_SV.h"

class EuropeanCall_SV_MC : public Pricer_SV
{
      public:
            EuropeanCall_SV_MC(int mc);
            virtual ~EuropeanCall_SV_MC(){};
            virtual double * getprice(double r, double alpha, double beta,
                double gamma, double rho, double v0, double S0, double K,
                double T, int m);

      private:
             int n;
};

#endif // EUROPEANCALL_SV_MC_H
```

EuropeanCall_SV_MC.cpp

```
#include <algorithm> // max
#include <cmath> // exp, sqrt, log
#include <new> // new
#include "EuropeanCall_SV_MC.h"
#include "Sampler.h" // normal sampler

using namespace std;

EuropeanCall_SV_MC::EuropeanCall_SV_MC(int mc)
{
       n = mc;
}

double * EuropeanCall_SV_MC::getprice(double r, double alpha, double beta,
    double gamma, double rho, double v0, double S0, double K, double T,
        int m)
{
       double eps = 1e-6;
       double h = T/m;

       NormalSampler NS;
       double Zv;
       double Zs;

       double *v = new double[m+1];
       double *S = new double[m+1];
```

```
        double *f = new double[n];

        v[0] = v0;
        S[0] = log(S0);
        double F = 0;
        for(int i = 0; i < n; i++)
        {
                for(int j = 1; j <=m; j++)
                {
                        Zv = NS.getnumber();
                        Zs = rho*Zv + sqrt(1-pow(rho,2))*NS.getnumber();

                        v[j] = v[j-1] + alpha * ( beta - max(eps,v[j-1]) )
                            * h +
                             gamma * sqrt( max(eps,v[j-1]) ) * Zv * sqrt(h);
                        S[j] = S[j-1] + ( r - 0.5 * max(eps,v[j-1]) ) * h +
                             sqrt( max(eps,v[j-1]) ) * Zs * sqrt(h);
                }
                f[i] = exp(-r*T)*max(exp(S[m])-K,0.0);
                F += f[i];
        }
        F = F/n;

        double sn = 0;
        for(int i = 0; i < n; i++)
        {
                sn += (f[i]-F)*(f[i]-F);
        }
        sn = sn/n;

        double a = 1.96;
        double CIl = F-a*sn/sqrt(n);
        double CIu = F+a*sn/sqrt(n);

        static double result[2];
        result[0] = CIl;
        result[1] = F;
        result[2] = CIu;

        delete [] v;
        delete [] S;
        delete [] f;

        return result;
}
```

AsianCall_SV_MC.h

```
#ifndef ASIANCALL_SV_MC_H
#define ASIANCALL_SV_MC_H

#include "Pricer_SV.h"

class AsianCall_SV_MC : public Pricer_SV
{
      public:
              AsianCall_SV_MC(int mc);
              virtual ~AsianCall_SV_MC(){};
```

```
        virtual double * getprice(double r, double alpha, double beta,
            double gamma, double rho, double v0, double S0, double K,
            double T, int m);

    private:
        int n;
};

#endif // ASIANCALL_SV_MC_H
```

AsianCall_SV_MC.cpp

```
#include <algorithm> // max
#include <cmath> // exp, sqrt, log
#include <new> // new

#include "AsianCall_SV_MC.h"
#include "Sampler.h" // normal sampler

using namespace std;

AsianCall_SV_MC::AsianCall_SV_MC(int mc)
{
    n = mc;
}

double * AsianCall_SV_MC::getprice(double r, double alpha, double beta,
    double gamma, double rho, double v0, double S0, double K, double T,
    int m)
{
    double eps = 1e-6;
    double h = T/m;

    NormalSampler NS;
    double Zv;
    double Zs;

    double *v = new double[m+1];
    double *S = new double[m+1];
    double *f = new double[n];

    v[0] = v0;
    S[0] = log(S0);
    double F = 0;

    // ---------
    double sumS;
    // ---------

    for(int i = 0; i < n; i++)
    {

        // ------
        sumS = 0;
        // ------

        for(int j = 1; j <=m; j++)
        {
```

```
                        Zv = NS.getnumber();
                        Zs = rho*Zv + sqrt(1-pow(rho,2))*NS.getnumber();

                        v[j] = v[j-1] + alpha * ( beta - max(eps,v[j-1]) )
                            * h +
                             gamma * sqrt( max(eps,v[j-1]) ) * Zv * sqrt(h);
                        S[j] = S[j-1] + ( r - 0.5 * max(eps,v[j-1]) ) * h +
                             sqrt( max(eps,v[j-1]) ) * Zs * sqrt(h);

                        // ----------
                        sumS += exp(S[j]);
                        // ----------
                }
                f[i] = exp(-r*T) * max( sumS/m - K, 0.0);
                F += f[i];
        }
        F = F/n;

        double sn = 0;
        for(int i = 0; i < n; i++)
        {
                sn += (f[i]-F)*(f[i]-F);
        }
        sn = sn/n;

        double a = 1.96;
        double CIl = F-a*sn/sqrt(n);
        double CIu = F+a*sn/sqrt(n);

        static double result[2];
        result[0] = CIl;
        result[1] = F;
        result[2] = CIu;

        delete [] v;
        delete [] S;
        delete [] f;

        return result;
}
```

BarrierPricer_SV.h

```
#ifndef BARRIERPRICER_SV_H
#define BARRIERPRICER_SV_H

class BarrierPricer_SV{

      public:
             virtual double * getprice(double barr, double r, double alpha,
                 double beta, double gamma, double rho, double v0, double
                    S0,
                 double K, double T, int m) = 0;
};

#endif // BARRIERPRICER_SV_H
```

EuropeanCallDownAndOut_SV_MC.h

```
#ifndef EUROPEANCALLDOWNANDOUT_SV_MC_H
#define EUROPEANCALLDOWNANDOUT_SV_MC_H

#include "BarrierPricer_SV.h"

class EuropeanCallDownAndOut_SV_MC : public BarrierPricer_SV
{
	public:
		EuropeanCallDownAndOut_SV_MC(int mc);
		virtual ~EuropeanCallDownAndOut_SV_MC(){};
		virtual double * getprice(double barr, double r, double alpha,
		    double beta, double gamma, double rho, double v0, double
		       S0,
		    double K, double T, int m);

	private:
		 int n;
};

#endif // EUROPEANCALLDOWNANDOUT_SV_MC_H
```

EuropeanCallDownAndOut_SV_MC.cpp

```
#include <algorithm> // max
#include <cmath> // exp, sqrt, log
#include <new> // new

#include "EuropeanCallDownAndOut_SV_MC.h"
#include "Sampler.h" // normal sampler

using namespace std;

EuropeanCallDownAndOut_SV_MC::EuropeanCallDownAndOut_SV_MC(int mc)
{
	n = mc;
}

double * EuropeanCallDownAndOut_SV_MC::getprice(double barr, double r,
    double alpha, double beta, double gamma, double rho, double v0, double
       S0,
    double K, double T, int m)
{
	double eps = 1e-6;
	double h = T/m;

	NormalSampler NS;
	double Zv;
	double Zs;

	double *v = new double[m+1];
	double *S = new double[m+1];
	double *f = new double[n];

	v[0] = v0;
	S[0] = log(S0);
```

```
        double F = 0;
        // ----------------
        double minS;
        barr = log(barr);
        // ----------------
        for(int i = 0; i < n; i++)
        {
                // ---------
                minS = S[0];
                // ---------

                for(int j = 1; j <=m; j++)
                {
                        Zv = NS.getnumber();
                        Zs = rho*Zv + sqrt(1-pow(rho,2))*NS.getnumber();

                        v[j] = v[j-1] + alpha * ( beta - max(eps,v[j-1]) )
                            * h +
                             gamma * sqrt( max(eps,v[j-1]) ) * Zv * sqrt(h);
                        S[j] = S[j-1] + ( r - 0.5 * max(eps,v[j-1]) ) * h +
                             sqrt( max(eps,v[j-1]) ) * Zs * sqrt(h);

                        // --------------------
                        minS = min(minS,S[j]);
                        // --------------------
                }
                f[i] = exp(-r*T) * max(exp(S[m])-K,0.0) *
                    ( minS > barr ? 1 : 0 );
                F += f[i];
        }
        F = F/n;

        double sn = 0;
        for(int i = 0; i < n; i++)
        {
                sn += (f[i]-F)*(f[i]-F);
        }
        sn = sn/n;

        double a = 1.96;
        double CIl = F-a*sn/sqrt(n);
        double CIu = F+a*sn/sqrt(n);

        static double result[2];
        result[0] = CIl;
        result[1] = F;
        result[2] = CIu;

        delete [] v;
        delete [] S;
        delete [] f;

        return result;
}
```

VolRiskClass.h

```
#ifndef VOLRISKCLASS_H
```

```
#define VOLRISKCLASS_H

class VolRiskClass{

      public:
            virtual double Value(double lambda) = 0;

};

#endif // VOLRISKCLASS_H
```

VolRiskPrice.h

```
#ifndef VOLRISKPRICE_H
#define VOLRISKPRICE_H

#include "VolRiskClass.h"

class VolRiskPrice : public VolRiskClass
{
      public:
            VolRiskPrice(double _r, double _alpha, double _beta, double
                _gamma,
                double _rho,double _v0,double _S0,double _K,double _T);
            virtual ~VolRiskPrice(){};
            virtual double Value(double lambda);

      private:
            double S0;
            double K;
            double T;
            double r;

            double v0;
            double alpha;
            double beta;
            double gamma;
            double rho;
};

#endif // VOLRISKPRICE_H
```

VolRiskPrice.cpp

```
#include "VolRiskPrice.h"

// declare function returning closed form price from Heston model
double getpriceCF(double r, double alpha, double beta, double gamma,
    double rho, double lambda, double v0, double S0, double K, double T);

using namespace std;

// constructor
VolRiskPrice::VolRiskPrice(double _r, double _alpha, double _beta, double
    _gamma,
    double _rho,double _v0,double _S0,double _K,double _T){
```

```
    S0 = _S0;
    K  = _K;
    T  = _T;
    r  = _r;

    v0    = _v0;
    alpha = _alpha;
    beta  = _beta;
    gamma = _gamma;
    rho   = _rho;

}

// define member function
double VolRiskPrice::Value(double lambda)
{
    return getpriceCF(r,alpha,beta,gamma,rho,lambda,v0,S0,K,T);
}
```