Computational Methods: Final Report

Matthew Mercuri

Introduction and Purpose

The goal of this report is to demonstrate a plethora of computational methods that correspond to a variety of practical applications. For the most part, the methods are used within the applied mathematics field, specifically mathematical finance. Though the methods extend to an assortment of use-cases. The majority of this report centers around implementing the methodology behind the pricing of a financial call option (European). The model that is used for the report is called the local volatility function (LVF) model. What follows is the calibration of the model.

The setup for the model assumes:

$$dS = r \cdot Sdt + \sigma(S,t)SdW_t,$$

where $\sigma(S,t) = \max(0.0, x_1 + x_2 S + x_3 S^2)$. The European call option value is then the solution of the PDE

$$V_\tau - \frac{\sigma^2(S,\tau)}{2} S^2 V_{SS} - rSV_S + rV = 0, \qquad (1)$$

where $\tau = T - t$ is the time to maturity.

1. Initial and Boundary Conditions

As this is a European call option, we have:

$Payoff = max(S_T - K,\ 0)$ where $S_T$is the stock price at expiration & $K$is the strike price

Of course the payoff is the value of the option at **expiration**, so it follows that:

$$V(S,t)\text{: at } t = 0 \Rightarrow V(S_0, 0) = max(S_0 - K, 0)$$

This is our **initial condition** for the partial differential equation (PDE) from above. Now moving on to the boundary conditions:

$$V(S,t)\text{: at } S = 0 \Rightarrow V(0,t) = max(0 - K, 0) = 0$$

$$V(S,t)\text{: at } S = S_{max} \Rightarrow V(S_{max}, t) = max(S_{max} - K, 0)$$

These are our boundary conditions: $V(0,t) = 0$and$V(S_{max}, t) = max(S_{max} - K, 0)$. It is important to note here that although the value of the call is always zero when the underlying stock price is 0, the maximum value is ***potentially unbounded***. In fact, it technically is, as a European call option has unlimited potential for gain given the stock price can tend towards infinity. For convenience and practical purposes, we can assume the maximum value of the option is its value when the underlying stock price is a value that has an extremely low probability of being realized (S_{max}). This will likely be multiples of the starting underlying price and of course greater than the strike price (upside only).

2. Monte Carlo Pricing

We can use the Monte Carlo method in order to price the option. To do this, we basically use random generation to simulate paths the stock price can take through time. Of course, these paths will be generated in such a way that conform to the LVF model. From there we can compute the value of the option.

Pseudo-Code

```
1. Pass initial information to the function
    a. $S_0$- initial stock price
    b. $K$- option's strike price
    c. $T$- time to maturity
    d. $r$- risk-free rate
    e. $x$- parameters to calibrate volatility
    f. $M$- desired number of paths to simulate
    g. $N$- number of desired time steps, $\delta t = T/N$
2. Create a vector (1 by M) that stores the final value of stock's prices
3. Create a for loop that runs from 1 to M (only runs for as many simulations as defined)
    a. Create a vector (1 by N) to store results of stock price
    b. Create a for loop that runs from 1 to N (runs for each time step)
        i. Calculate stock price at current time step using model formula
        ii. Save stock price in vector
        iii. End for loop after N time steps
    c. End for loop after M simulations
4. Use the vector of final stock prices to calculate the value of the call option for each path
5. Find the average value of the call option
6. Discount the average value with $e^{-rT}$ to obtain the value of the option
```

Code

```
% Matthew Mercuri
% MTH 600 Final
% April 24, 2021

function V = Eur_Call_LVF_MC(S0, K, T, r, x, M, N)
    %
    % Price the European call option of the LVF model by the Monte Carlo method
    %
    % Input
    % S0 - initial stock price
    % K - strike price
    % T - maturity
    % r - risk free interest rate
    % x - vector parameters for the LVF σ, [x1, x2, x3]
    % M - number of simulated paths
    % N - number of time steps, i.e., δt = T /N.
    %
    % Output
    % V - European call option price at t = 0 and S0.

    % creating vector to store stock price at expiration
    S_finals = zeros(1, M);
    dt = T/N;  % constant value for delta t to use later

    % for loop to generate M simulations (paths)
    for i = 1:M
        S = zeros(1, N); % vector to store path's prices
        S(1, 1) = S0;  % assigning initial price as first value
        % for loop to generate stock price at each time step
        for j = 2:N
            % calculating volatitily by LVF model
            sigma = max([0.0, x(1)+(x(2)*S(j-1))+(x(3)*(S(j-1)^2))]);
            % calculating time step's price by LVF
            S(1, j) = S(1, j-1) + (r*S(1, j-1)*dt) + (sigma*S(1, j-1)*(normrnd(0, 1)*sqrt(dt)));
        end
        S_finals(1, i) = S(1, N);  % storing result of final price
    end
    % finding value for call (each path)
    V_finals = S_finals - K;
    V_finals = max(V_finals, 0);

    % finding average value
    V_avg = mean(V_finals);

    % discounting for final price
    V = exp(-r*T)*V_avg;
end
```

3. Explicit Pricing - Finite Difference Method

In this section, we provide a means to implement the finding of a solution (thus option price) to the LVF model PDE. This is very much different from the previous method as we are trying to *explicitly* solve the equation through finding the input parameters (essentially its derivatives). Of course, it is difficult to analytically solve for the derivatives, so we can use a numerical method instead. Specifically we are presenting the Crank-Nicolson method with central differencing and upstream weighting.

Pseudo-Code

```
1. Pass the initial parameters to the function
    a. $S_0$- the stock's initial price
    b. $K$- strike price of the option
    c. $T$- time to maturity
    d. $r$- risk free interest rate
    e. $x$- parameters for LVF's sigma
    f. $S_{max}$- presumed maximum attainable stock price
    g. $M$- number of differences in stock price for FD method
    h. $N$- number of time steps for FD method
2. Compute necessary constants for later use (stock price interval and time interval)
3. Create vector from $S_0 = 0$to $S_{max}$
4. Create matrix that will store solutions
5. Calculate initial conditions for each price
6. Create a for loop for time steps
    a. Create a for loop for stock prices
    b. Compute value for sigma using LVF model
    c. Compute derivatives using FD method
    d. Enact upstream waiting
    e. Calculate option value by LVF model and store result, move to next price
    f. End time for stock prices
7. End for loop for time steps
8. Select for current price and return option value
```

Code

```
% Matthew Mercuri
% MTH 600 Final
% April 24, 2021

function V0 = Eur_Call_LVF_FD(S0, K, T, r, x, Smax, M, N)
    %
    % Price the European call option of the LVF model by the explicit finite difference method.
    %
    % Input
    % S0 - initial stock price
    % K - strike price
    % T - maturity
    % r - risk free interest rate
    % x - vector parameters for the LVF σ, [x1, x2, x3]
    % Smax - upper bound of the stock price
    % M - number of stock price difference, i.e., δS = Smax/M
    % N - number of time steps, i.e., δt = T /N.
    %
    % Output
    % V 0 - European call option price at t = 0 and S0.

    % computing necessary constants
    dS = Smax/M;  % increments for stock price
    dtau = T/N; % incremens for time steps

    % creating vector of all possible stock prices
    S = 0:dS:Smax;

    % creating matrix to represent solutions to PDE (grid)
    V = zeros(N,M+1);

    % computing initial conditions for each price and saving the result to
    % our grid
    for i = 1:M+1
        V(1,i) = max((S(i) - K), 0);
    end

    for n = 1:N-1
        % computing boundary conditions for each price of 0 and Smax
        V(n+1, M+1) = V(n, M+1);  % this is just payoff of Smax
        V(n+1, 1) = V(n, 1)*((1-r)*dtau); % this is just 0

        % using finite difference to computer values within boundaries
        for j = 2:M
            % compute new value for sigma
            sigma = max([0.0, x(1)+(x(2)*S(j-1))+(x(3)*(S(j-1)^2))]);

            % Calculating our derivates approximated using the FD method.
            % The first two are central differences
            alpha_central = ((sigma^2)*(S(j)^2))/(((S(j)-S(j-1))*(S(j+1)-S(j-1))))-(r*S(j))/(S(j+1)-S(j-1));
            beta_central = ((sigma^2)*(S(j)^2))/(((S(j+1)-S(j))*(S(j+1)-S(j-1))))+(r*S(j))/(S(j+1)-S(j-1));
            % forward differences
            alpha_forward = ((sigma^2)*(S(j)^2))/(((S(j)-S(j-1))*(S(j+1)-S(j-1))));
            beta_forward = ((sigma^2)*(S(j)^2))/(((S(j+1)-S(j))*(S(j+1)-S(j-1))))+(r*S(j))/(S(j+1)-S(j-1));

            % Upstream weighting requires that both our values for beta and
            % alpha be postive or zero. As such, if our central difference
            % yields a negative result, we use the forward difference which
            % by definition will be positive
            if (alpha_central >= 0 ) && (beta_central >= 0)
                alpha = alpha_central;
                beta = beta_central;
            else
                alpha = alpha_forward;
                beta = beta_forward;
            end

            % computing our value for the corresponding node (n+1, j) by
            % LVF model
            V(n+1, j) = V(n, j)*(1-(alpha+beta+r)*dtau)+alpha*dtau*V(n, j-1)+beta*dtau*V(n, j+1);

        end
    end

    % selecting the option value at starting price at t=0
    V0 = V(N, S==S0);

end
```

4. Comparing the Two Methods

Now that we have put the techniques into code, we can use them to actually find the price of a European call option. The parameters That will be used to test the models are summarized in the following table:

Monte Carlo	Explicit
$S_0 = 1$ $K = 0$ $T = 0.25$ $r = 0.03$ $x = [0.2,\ 0.001,\ 0.003]$ $M = 10,000$ $N = 100$	$S_0 = 1$ $K = 0$ $T = 0.25$ $r = 0.03$ $x = [0.2,\ 0.001,\ 0.003]$ $S_{max} = 3$ $M = 30$ $N = 100$

Using the code presented above, we can find the value of the call option:

Monte Carlo	Explicit
European Call Value = $0.0428	European Call Value = $0.038

As shown in the table above, the two methods yield similar results. Both price the option to be approximately **$0.04**. Of course the difference can be attributed to the individual pitfalls of the techniques.

For the Monte Carlo method, we can converge to a more accurate solution by either increasing the amount of simulations or using a variance reduction technique (like the antithetic variates technique). In the explicit method, we can increase the size of our solution grid. What this essentially means is we can compute solutions using smaller time intervals and/or smaller price intervals. Of course, the tradeoff that comes with making these improvements is computational time and complexity. This would entail both methods being more expensive to compute than they currently are with the above parameters.

Code

```
% Matthew Mercuri
% MTH 600 Final
% April 24, 2021

% This file calls two functions to compute the value of the same European
% call option. The first uses a Monte Carlo method and the second use an
% explicit method (FD PDE solution). Please note the input parameters are
% slightly different depending on the method

% Monte Carlo Method
V_mc = Eur_Call_LVF_MC(1, 1, 0.25, 0.03, [0.2 0.001 0.003], 10000, 100);

% Explicit Method
V_e = Eur_Call_LVF_FD(1, 1, 0.25, 0.03, [0.2 0.001 0.003], 3, 30, 100);

fprintf('The value for the call using the MC method is: $%d \n', V_mc)
fprintf('The value for the call using the explicit method is: $%d \n', V_e)
```

5. Model Calibration

The purpose of this section is to demonstrate how we can calibrate the LVF model. If you notice, there is some freedom of choice with the initial value of x. So far, we have chosen them seemingly arbitrarily. Through, in practice we can choose x such that it best approximates the pricing we see in the market. This is what we call calibration. The calibration problem is essentially that of solving a non-linear least squares problem that looks like:

$$\min_{x \in \Re^n} \frac{1}{2} \sum_{j=1}^{m} \Big(V_0(K_j, T_j; x) - V_0^{\text{mkt}}(K_j, T_j) \Big)^2 . \tag{2}$$

Let $F(x)$ denote the vector of model option value errors from the market prices:

$$F(x) = \begin{bmatrix} V_0(K_1, T_1; x) - V_0^{\text{mkt}}(K_1, T_1) \\ \vdots \\ V_0(K_m, T_m; x) - V_0^{\text{mkt}}(K_m, T_m) \end{bmatrix}$$

Vector $F(x) : \Re^n \rightarrow \Re^m$ is a nonlinear function of the model parameter $x \in \Re^n$ and the calibration problem (2) is equivalently formulated as

$$\min_{x \in \Re^n} \left(f(x) \overset{\text{def}}{=} \frac{1}{2} \|F(x)\|_2^2 \right) \tag{3}$$

In equation 2, we are essentially trying to find x such that the value we compute for options of different strike prices closely matches what we see in the market. The most efficient way to solve this problem is through a numerical method like the Gauss-Newton method. It requires the computation of the Jacobian matrix, which we can use the finite difference method to approximate. The Jacobian takes the form:

$$J(x) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \frac{\partial F_m}{\partial x_2} & \cdots & \frac{\partial F_m}{\partial x_n} \end{pmatrix}$$

We will be using the Monte Carlo method from above to calibrate the LVF model against this data (parameters not listed can be assumed to be the same from above):

K	0.80	0.85	0.90	0.95	1.00	1.05	1.10
V_0	0.3570	0.2792	0.2146	0.1747	0.1425	0.1206	0.0676

Code - Computing Jacobian and Objective Function (Part A)

The code below simply computes the Jacobian matrix and the $F(x)$from above. It is documented to explain what it is trying to achieve at each major step.

```
% Matthew Mercuri
% MTH 600 Final
% April 24, 2021

function [F, J] = myfun(x)
   % ========== computing F ==========
   % saving observed value to vector
   V_observed = [0.3570 0.2792 0.2146 0.1747 0.1425 0.1206 0.0676];
   % corresponding strikes for options
   Ks = [0.80 0.85 0.90 0.95 1.00 1.05 1.10];

   V_computed = zeros(1, length(Ks));  % vector to store computed values
   % computing values for option given x (using Monte Carlo method)
   % NOTE: this takes quite a bit of time
   for i = 1:length(Ks)
       V_computed(i) = Eur_Call_LVF_MC(1, Ks(i), 0.25, 0.03, x, 10000, 100);
   end

   % Objective function values at x
   F = V_computed - V_observed;
   % =================================

   if nargout > 1 % Two output arguments
       % ========== computing J ==========
       J = zeros(length(Ks), length(x));  % create a zeros matrix to store Jacobian as we compute it
       delta_x = 0.0001;  % small value for our FD approximation
       delta_x_vector = (zeros(1, length(x)) + delta_x); % increment for FD method
       for m = 1:length(x)
           xp = x + delta_x_vector(m);
           for n = 1:length(Ks)
               F_p = Eur_Call_LVF_MC(1, Ks(n), 0.25, 0.03, xp, 10000, 100);  % compute value for F
               J(n, m) = (F_p - F(n)) / delta_x;  % apply FD approx.
           end
       end
       % =================================
   end
end
```

Code - Calibration (Part B)

Next, we calibrate the model using the above code. We use two sets of points as our initial value for x:

$$x_0 = [0.2\ 0\ 0]$$

$$x_0 = [0.2\ 0.1\ 0.01]$$

The following code calibrates the LVF model:

```
% Matthew Mercuri
% MTH 600 Final
% April 24, 2021

% observed values
Ks = [0.80 0.85 0.90 0.95 1.00 1.05 1.10];
V_observed = [0.3570 0.2792 0.2146 0.1747 0.1425 0.1206 0.0676];

% initial values of x to start calibration
x01 = [0.2; 0.0; 0.0];
x02 = [0.2; 0.1; 0.01];

% configuring LM method to find the best values for x
options = optimoptions('lsqnonlin', 'SpecifyObjectiveGradient', true);
options.Algorithm = 'levenberg-marquardt';
options.Display = 'iter';

% using non-linear least squares and the above built-in function to
% achieve the best values for x
[x1, res1] = lsqnonlin(@myfun, x01, [], [], options);  % for x01
[x2, res2] = lsqnonlin(@myfun, x02, [], [], options);  % for x02

% displaying results in console
disp(x1)
disp(x2)

% using the optimal x values that were computed above to price our options
% creating vectors to store calibrated values
V_x1 = zeros(1, length(Ks));
V_x2 = zeros(1, length(Ks));

for b = 1:length(Ks)
   V_x1(1, b) = Eur_Call_LVF_MC(1, Ks(b), 0.25, 0.03, x1, 10000, 100);
   V_x2(1, b) = Eur_Call_LVF_MC(1, Ks(b), 0.25, 0.03, x2, 10000, 100);
end

% Plotting results
figure(1)
plot(Ks, V_x1, 'LineWidth', 1);
hold on;

plot(Ks, V_x2, 'LineWidth', 1);
plot(Ks, V_observed,'LineWidth', 1);
hold off;

```

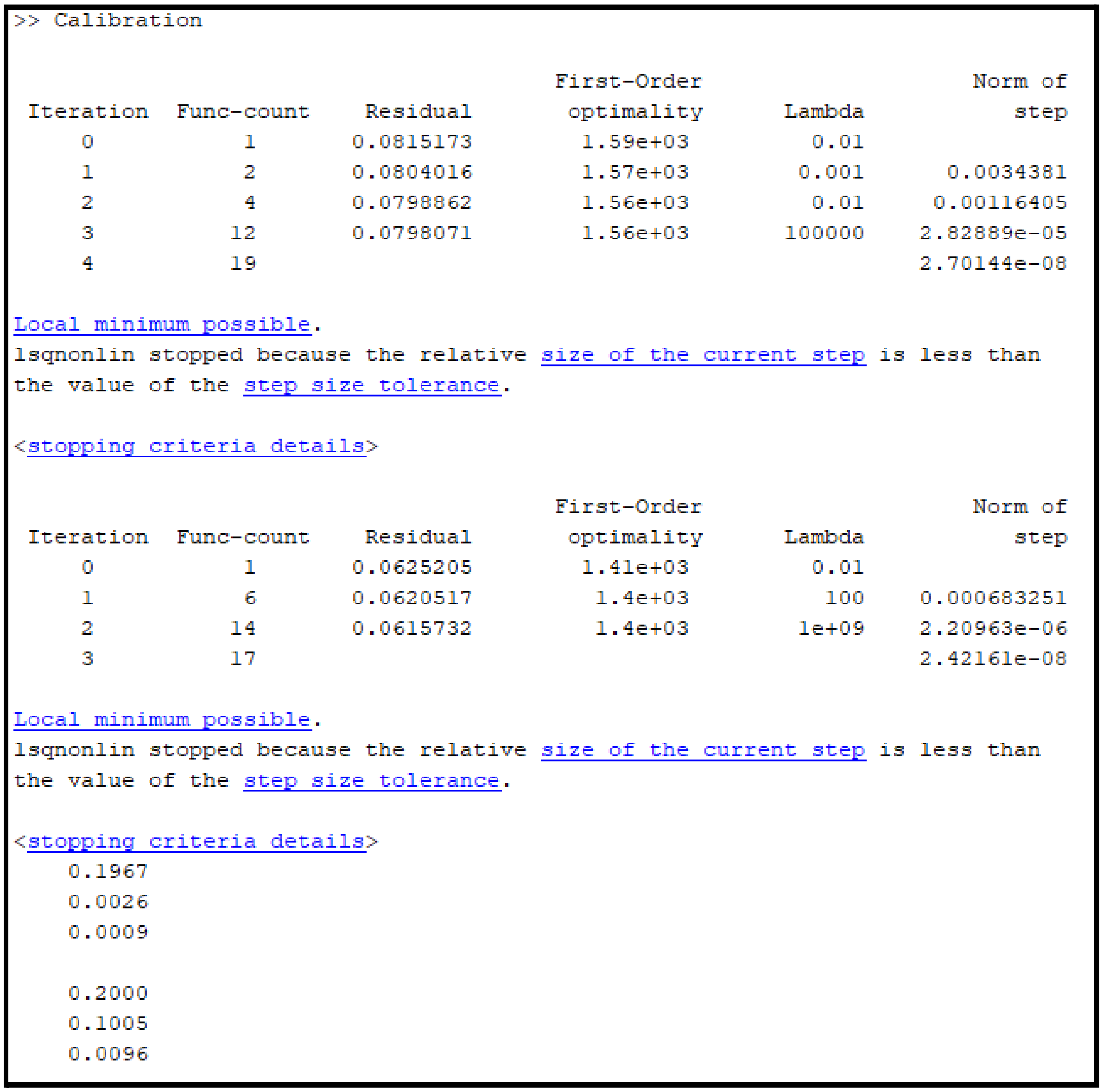
```
title('Calibrating x with MC Method for Pricing Options (LVF Model)');
xlabel('Strike Price (K)');
ylabel('Initial Value V(K,0)');
axis('tight');
legend('MC 1', 'MC 2', 'Market');

% computing implied volatilities
i_mkt = blsimpv(1, Ks, 0.03, 0.25, V_observed);
i_mc1 = blsimpv(1, Ks, 0.03, 0.25, V_x1);
i_mc2 = blsimpv(1, Ks, 0.03, 0.25, V_x2);


% Plotting implied volatility
figure(2)
plot(Ks, i_mkt, 'ko')
hold on

plot(Ks, i_mc1, 'LineWidth', 1.5)
plot(Ks, i_mc2, 'LineWidth', 1.5)

axis('tight')
legend('Observed', 'Fit 1', 'Fit 2')
title('Implied Volatility')
xlabel('Strike Price (K)')
ylabel('Sigma')
hold off

```

Results and Discussion

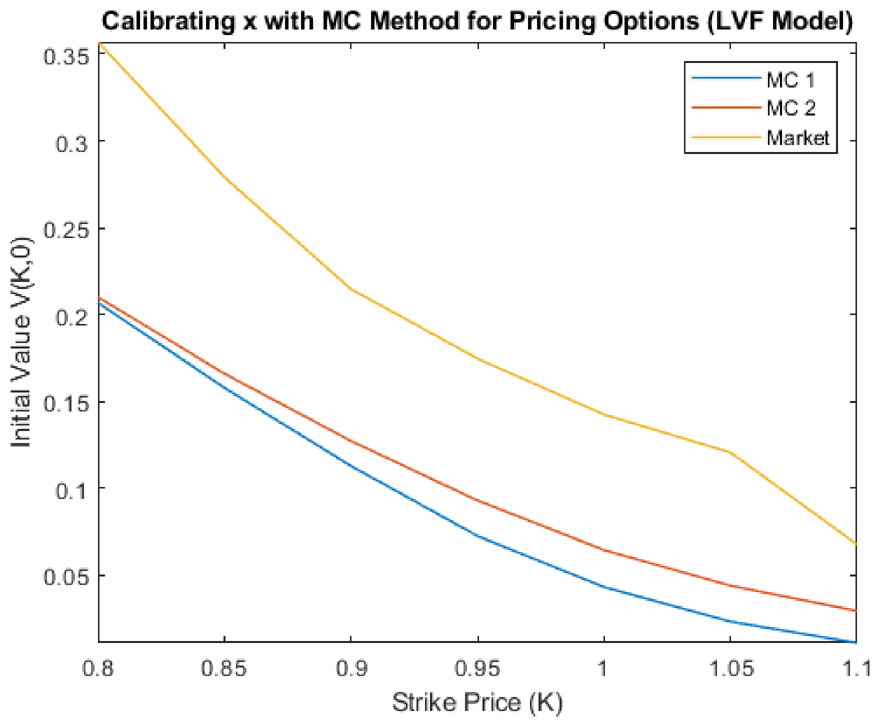
Computed Values for x (table 4)

$x_0 = [0.2\ 0\ 0]$	$x^* = [0.1967\ 0.0026\ 0.0009]$	F = [-0.1512 -0.1212 -0.1033 -0.1014 -0.0992 -0.0979 -0.0563]
$x_0 = [0.2\ 0.1\ 0.01]$	$x^* = [0.200\ 0.1005\ 0.0096]$	F = [-0.1463 -0.1138 -0.0868 -0.0823 -0.0765 -0.0757 -0.0389]

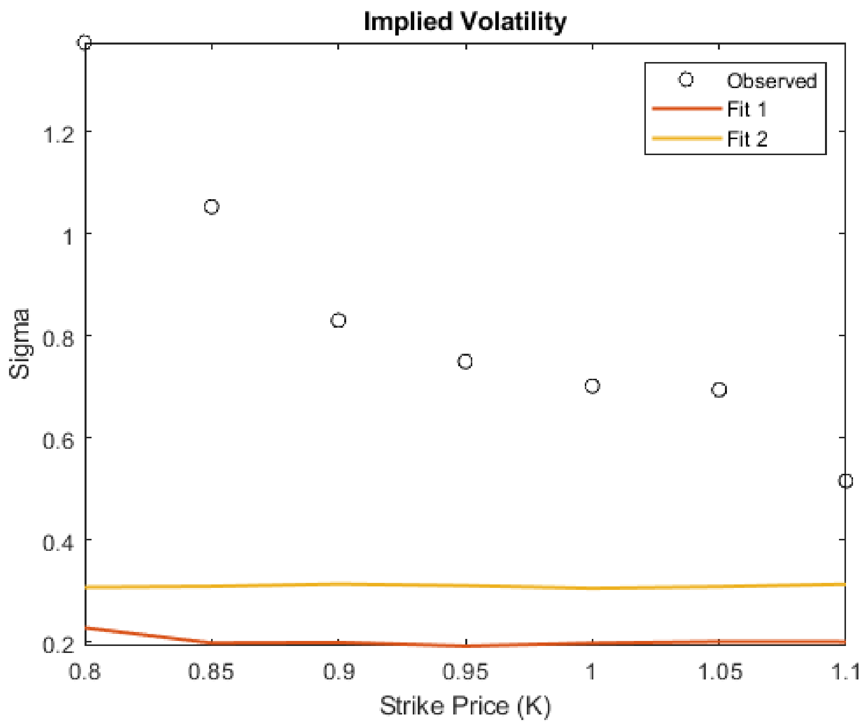
As you can see, the fitted values for x deviate slightly from the initial. This essentially means that the initial values are crude approximations that may still provide insight for the call option value. However there is still a lot of deviation from the observed market prices. Below is the result of the solver:



```
>> Calibration

                                         First-Order                    Norm of
 Iteration  Func-count     Residual       optimality      Lambda           step
     0           1        0.0815173       1.59e+03         0.01
     1           2        0.0804016       1.57e+03        0.001       0.0034381
     2           4        0.0798862       1.56e+03         0.01      0.00116405
     3          12        0.0798071       1.56e+03       100000     2.82889e-05
     4          19                                                  2.70144e-08

Local minimum possible.
lsqnonlin stopped because the relative size of the current step is less than
the value of the step size tolerance.

<stopping criteria details>

                                         First-Order                    Norm of
 Iteration  Func-count     Residual       optimality      Lambda           step
     0           1        0.0625205       1.41e+03         0.01
     1           6        0.0620517        1.4e+03          100     0.000683251
     2          14        0.0615732        1.4e+03        1e+09     2.20963e-06
     3          17                                                  2.42161e-08

Local minimum possible.
lsqnonlin stopped because the relative size of the current step is less than
the value of the step size tolerance.

<stopping criteria details>
    0.1967
    0.0026
    0.0009

    0.2000
    0.1005
    0.0096
```



Graph of the initial option prices at various strike prices (given different fits)



Implied volatility of the two fits compared to BS parameter

Based on the above charts and solver results, it is clear that the fit achieved by the code is not as strong as it perhaps could have been. Suffice to say, the solver did not achieve an appropriate fit to allow the LVF model to price options accurately. This is especially evident if you look at the values for F in the (table 4). The calibration error for both fits are:

$x_0 = [0.2\ 0\ 0]$	$x^* = [0.1967\ 0.0026\ 0.000$	Calibration Error $\|\|F(x^*)\|\|_2^2 = 0.0811$
$x_0 = [0.2\ 0.1\ 0.0$	$x^* = [0.200\ 0.1005\ 0.0096]$	Calibration Error $\|\|F(x^*)\|\|_2^2 = 0.0638$