

MATLAB Project Developer Documentation

With Instructions

Tanishq Chauhan

UNIVERSITY OF YORK 205012860

CODE DOCUMENTATION AND END USER'S INSTRUCTIONS

The following document will attempt to demonstrate the rationale behind the project code. This includes the mathematical methodology and logic for the program. Additionally, the end user's document will describe the output to be expected post program execution, and the conditions on the stability of the program.


This guide has been written by Tanishq Chauhan (205012860) for the purpose of supplementing the MATLAB Practical Project code (included in the attached .zip file), and for evaluation by Dr. Eric Dykeman. The written code in shade boxes such as these  is supported by mathematical theory behind it. However, the discretization schemes, the matrix and the subsequent procedures has not been presented here as they are described sufficiently in the lecture notes (which have been referred to while coding).

Table of Contents

DEVELOPER DOCUMENTATION

IMPLEMENTING MONTECARLO SIMULATION **3**

SECTION 1.01 – DISCRETIZATION & ANTITHETIC SAMPLING 3

SECTION 1.02 – CODE EXPLANATION 3

OPTION PRICING USING FINITE DIFFERENCE METHODS **8**

SECTION 1.03 – EXPLICIT FINITE DIFFERENCE METHOD 8

SECTION 1.04 – IMPLICIT FINITE DIFFERENCE METHOD 11

SECTION 1.05 – CRANK NICHOLSON SCHEME 12

END USER'S INSTRUCTIONS **15**

TEST RUN (RESULTS, ANOMALIES & CONVERGENCE) 16

Implementing Monte Carlo Simulation using Euler scheme to discretize a stochastic stock price process

Section 1.01 The Discretization and application of Antithetic sampling

The stock price process follows stochastic process defined by the equation

$$1 \quad dS_t = (r - q) * S_t dt + \sigma(S_t, t) * S_t * dW_t$$

Option price $V(S_t, t)$ on the above stock satisfies the PDE below

$$2 \quad \frac{\partial V}{\partial t} + \frac{1}{2} \frac{\partial^2 V}{\partial S^2} \sigma(S, t)^2 S^2 + (r - q) S \frac{\partial V}{\partial S} - rV = 0$$

The stock price process is discretized using Euler method as follows:

$$3 \quad S_{t_{k+1}} = S_{t_k} + (r - q)S_{t_k}(t_{k+1} - t_k) + 0.25 * \exp(-t_k) * (100^\alpha) * S_{t_k}^{1-\alpha} * \sqrt{(t_{k+1} - t_k)} * (Z_{k+1})$$

$$4 \quad S_{t_{k+1}} = S_{t_k} - (r - q)S_{t_k}(t_{k+1} - t_k) - 0.25 * \exp(-t_k) * (100^\alpha) * S_{t_k}^{1-\alpha} * \sqrt{(t_{k+1} - t_k)} * (Z_{k+1})$$

As observed in the 3rd and 4th equations, the stochastic stock price process has been discretized using Euler's method. Simultaneously an antithetic sample has also been created (equation 4).

Section 1.02 Code explanation

We start by adding the required parameters for calculating the option price

Code Snippet 1

```
S0 = 100; K = 100; B = 130; q = 0.05; r = 0.03;  
T = 0.5; alpha = 0.35; N = 1000000; M = 500;
```

The next step is to specify the time frame and the number of partitions to be made for the time

Code Snippet 2

```
dt = T/M;  
tm = 0:dt:T;  
m = length(tm)-1;  
dtm = diff(tm);
```

Creating vectors to store original stock price sample, and the antithetic sample:

Code Snippet 3

```
S = zeros(m+1,1);  
Sa = zeros(m+1,1);
```

This payoff vector stores the values for the original and antithetic samples simultaneously

Code Snippet 4

```
X = zeros(N,2);
```

After defining the base parameters and vectors, we now face the problem of creating the sample and its antithetic variate using the Euler Discretization scheme. This task is accomplished by creating a for loop which incorporates both type of samples, as seen below. Each sample generation code is discretized as required.

Code Snippet 5

```
for k=1:m  
    z = randn;  
    S(k+1) = S(k) + (r-q)*S(k)*dtm(k) +  
    0.25*exp( tm(k))*(100^alpha)*(S(k)^(1-  
    alpha))*...  
    sqrt(dtm(k))*z;  
    Sa(k+1) = Sa(k) + (r-q)*Sa(k)*dtm(k) -  
    0.25*exp(-tm(k))*(100^alpha)*(Sa(k)^(1-  
    alpha))*...  
    sqrt(dtm(k))*z;  
end
```

Since we are pricing Barrier Options, it is first important to understand the payoff pattern of Barrier option. Mathematically this is similar to a vanilla European Call option, with the exception that once the stock price crosses the Barrier agreed by both the parties, the option becomes void.

$$h_{up-and-out\ call}(S_T) = \begin{cases} \max(S_T - K, 0) & \text{if } \max_{0 \leq t \leq T} S_t < B \\ 0 & \text{otherwise} \end{cases}$$

5

The above equation shows the Barrier option payoff process. Keeping this in mind, we use the If-else condition to enforce the barrier on the payoffs stored from both discretized samples. The vector defined above as `X = zeros(N, 2)` stores payoff values, thus the condition is applied to it.

Code Snippet 6

```
if any(S >= 130)
X(i,1) = 0;
else
X(i,1) = exp(-T*r)*max(S(m+1)-K,0);
end
if any(Sa >= 130)
X(i,2) = 0;
else
X(i,2) = exp(-T*r)*max(Sa(m+1)-K,0);
end
```

We use a discounted payoff function in our payoff vector. The initial and final prices of the stock price are a known, hence can be computed without the scheme. To counterweight this, we have to discount the payoff throughout the process from S_2, \dots, S_{N-1} . This is because for MATLAB the solution already exists at the terminal points. So for it, S_2 is the first stock price, and S_{N-1} is the last stock price.

We now calculate the mean of the payoff vector. However, due to two samples being stored in a single vector we need to take their mean too.

The standard error is computed using the formula:

$$\sqrt{\frac{1}{n(n-1)} \left(\sum_{i=1}^n H_i^2 - n\hat{\mu}^2 \right)}$$

6

Code Snippet 7

```
B_mc = mean(mean(X));
% Compute the standard error of the sample
SE_C = sqrt((sum(mean(X,2).^2) -
N*B_mc^2)/(N*(N-1)));
```

```
% Compute the confidence interval.
CI_low = B_mc - 1.96*SE_C;
CI_up = B_mc + 1.96*SE_C;
```

The resultant option price is around the figures of **3.8...**, the values starting from the 3rd significant figure keeps changing every time the code is executed. This is due to the randomness of the sample generated in Monte Carlo simulation. Nonetheless, this gives us a stand-alone figure to which we can now compare option price obtained by Finite Difference Methods.

To give an idea about the random path generation, two figures obtained by plotting time partition against the stock price and its antithetic variate have been presented.

Code Snippet 8

```
figure(1)
plot(tm,S)
xlabel('Time')
ylabel('Stock Price')
title('Stock Price path in Monte Carlo using Euler
Discretization')
legend('Original Sample Stock Price')
```

Code Snippet 9

```
figure(2)
plot(tm,Sa)
xlabel('Time')
ylabel('Stock Price')
title('Stock Price path in Monte Carlo using Euler
Discretization')
legend('Antithetic Sample Stock Price')
```

We run a standard MATLAB code to plot the price path with respect to the time partitions. However, the price path changes every time the code is executed. This is due to the random path generation which is denoted by (Z_{k+1}) in the discretized equation (3 – 4).

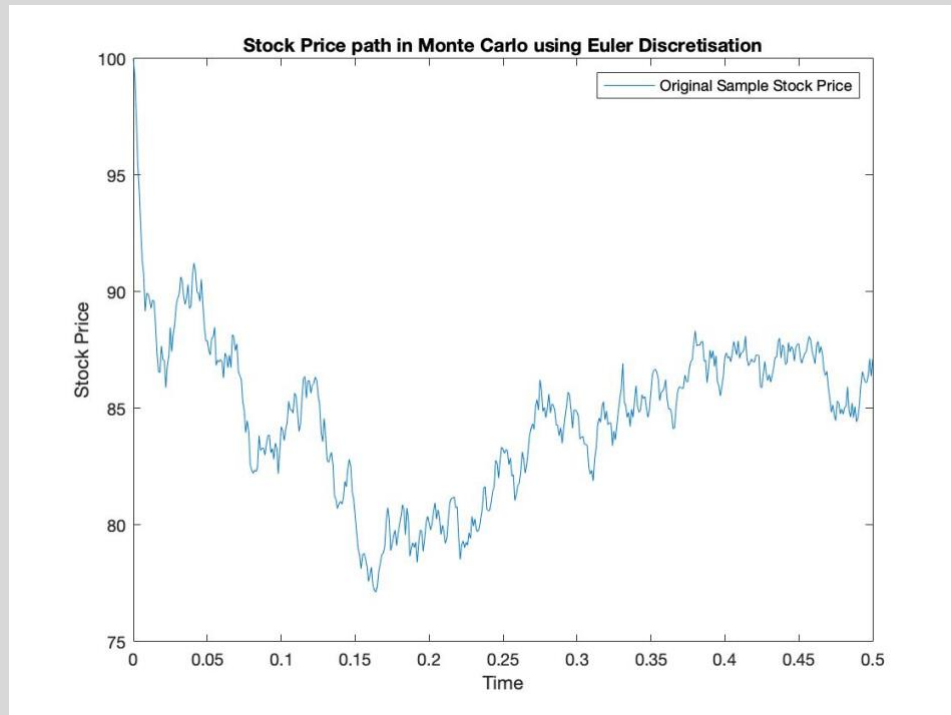


Figure 1 - Monte Carlo Simulation for the original process (Refer Test Run)

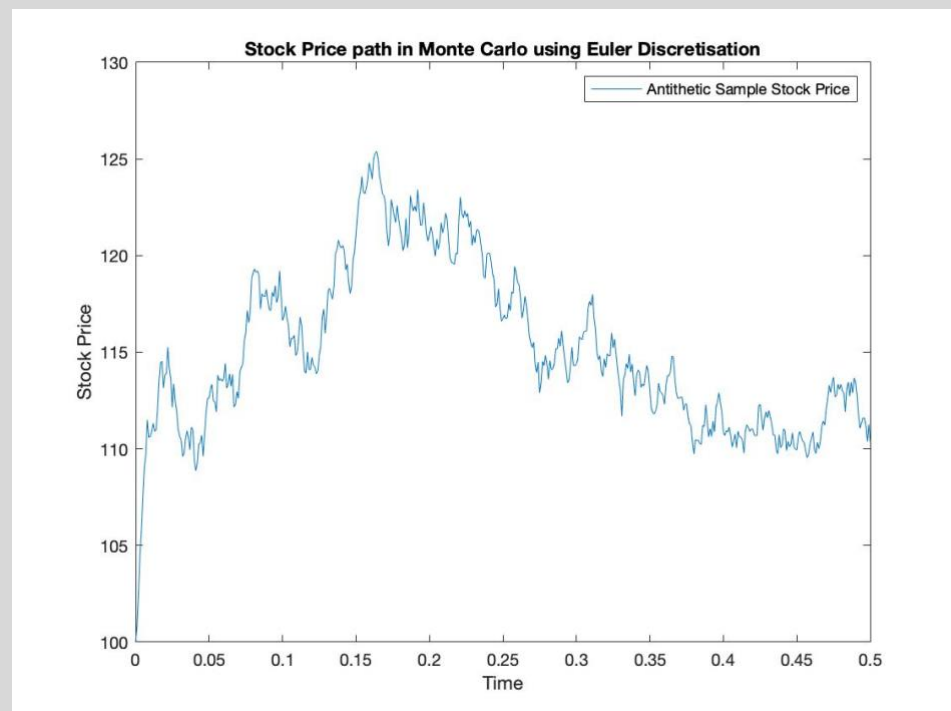


Figure 2 - Monte Carlo Simulation for the Antithetic process (Refer Test Run)

Figure 2 is visually antisymmetric to Figure 1 due to its antithetic nature

Above is the path followed by the original stock price sample in Figure 1, at no point the process goes above the barrier. This is due to the barrier condition we added in the later part of the code. Once the program starts running, the sample values are stored in their respective vectors. This is irrespective of whether the values at each point cross the barrier. The stored values are then filtered through the barrier limit (if-else condition), which then places the value of 0 in the vector grid in case the barrier is crossed at any point.

The code for this assignment has been taken from Lecture Week 10 spring term, and Practical Week 9 solutions provide by Dr. Eric Dykeman.

Using Finite Difference Methods to price Exotic Options

We discuss Explicit, Implicit, and Crank-Nicholson schemes to price exotics such as barrier options.

Section 1.03 Explicit Finite Difference Method (E-FDM)

A finite difference scheme is explicit when it can be computed forward in time using quantities from previous time steps. The code for E-FDM has been adapted from the Lecture Week 2 Spring code, as taught by Dr. Eric Dykeman. Appropriate boundary conditions have been added; sigma has been changed to match the variability of Local Volatility Surface.

We start by adding parameters essential for our computation, these will be same as the ones we described above in the Monte Carlo scheme.

Code Snippet 10

```
K = 100; S0 = 100; N = 1000; T = 0.5; M = 50000;
r = 0.03; q = 0.05; a = 0.35; % a is Volatility Alpha
```

This code here marks the prime difference between the Monte Carlo scheme we simulated and the Finite Difference scheme we are about to implement. As you can see, we are creating a N-dimensional vector of stock prices, compared to the Monte Carlo scheme where the original stochastic process was simulated by Euler's discretization method. The maximum number has been kept as 130 for the barrier condition.

Define the sequence of stock prices as an (N-1) vector, similarly for time.

Code Snippet 11

```
S1 =
linspace(Smin, Smax, N+1)';
dS = S1(2) - S1(1);
```

```
S = S1(2:N);
```

Code Snippet 12

```
tau = linspace(0,T,M+1);
dtau = tau(2) - tau(1);
```

For the given sigma: $\sigma(S_t, t) = \frac{1}{4} * e^{-t} * \frac{(100)^\alpha}{(S_t)^\alpha}$

7

we have to discretize the equation to add variability in the volatility.

Code Snippet 13

```
sigma = zeros(N-1,M);
for k=1:M
    for j=1:N-1
        sigma(j,k)=0.25*exp(-tau(k))*((100/S(j))^a);
    end
end
```

We define α and β which will change as per the grid point, for each grid point the α and the β will be constant. These are the two components of the discrete equation we define below.

Code Snippet 14

```
alpha = 0.5*(sigma.^2).*(S.^2)*dtau/(dS^2);
beta = (r - q)*S*dtau/(2*dS);
```

After resubstituting α and β into the discretized equation and re-arranging it, we get another equation of the form

$$\begin{aligned} &(-\alpha_{j,k+1} + \beta_{(j,k+1)})V_{j-1,k+1} + (1 + r_{k+1}\Delta\tau + 2\alpha_{j,k+1})V_{j,k+1} \\ &+ (-\alpha_{j,k+1} - \beta_{j,k+1})V_{j+1,k+1} = V_{j,k} \end{aligned}$$

8

Where $\hat{I}_{j,k+1}$, $\hat{d}_{j,k+1}$, $\hat{u}_{j,k+1}$ represent their respective constituents and are defined accordingly.

Code Snippet 15

```
l = alpha - beta;
d = 1 - r*dtau - 2*alpha;
u = alpha + beta;
```

Defining the payoff condition for the Barrier Option, which is similar to a vanilla European Call option.

Code Snippet 16

```
Vold = max(S - K, 0);
Vnew = Vold;
```

Finally, we implement the Explicit discretization scheme. This is similar to the Implicit code for pricing European Call with constant volatility, except that we have a barrier and a variable volatility.

We know that the following are the Dirichlet boundary conditions for the European Call option

$$V(S_{min}, \tau) = 0 ; V(S_{max}, \tau) = e^{-\delta\tau} S_{max} - e^{-r\tau} K$$

9

We set $V(S_{max}, \tau)$ as zero. To understand this first we need to get a rough idea of what happens when we initialize the **Vnew** and **Vold** vectors. The explanation is adapted from Finite Difference Method Notes for Spring Term (pg. 16) by Dr. Eric Dykeman.

$$Vnew = Vold[\max(S_2 - K, 0), \max(S_3 - K, 0), \dots, \max(S_N - K, 0)]$$

10

MATLAB starts by processing values from S_{min} and terminates at S_{max} , however since the option price for these values can be calculated without executing the scheme (as they are terminal points), the logic process skips them and starts from the second price point, as shown in equation 10.

The first process starts with $k = 1$, applying equation 10 and given the terminal conditions, MATLAB considers 2nd and $(N-1)^{th}$ price points as the 1st and the N^{th} point. Bringing the discretized equation 8 into scenario, it is visible how variable l is used to calculate S_2 ; l, d for S_3 ; l, u, d for S_4 , and so forth. Once the process reaches S_{max} , it is by conditioning not allowed to consider S_{max} because of the barrier. Hence the boundary value is allocated as zero, and the process (equation 10) moves forward to $k = 2$. The prices calculated at $k = 1$ form the base for calculating the values for $k = 2$, and the exact process re-ensues.

This same process will be the reason why boundary condition will be zero for all the FDM codes.

Earlier we discretized our sigma so that it produces a $(N-1)*(M)$ -dimensional matrix, thus the sigma produces a constant value for every grid point. This step makes α an $(N-1)*(M)$ -dimensional matrix, so we discretize l, u, d i.e. the lower, main, and upper diagonal of the tridiagonal matrix.

```

for k=1:M
    % Boundary condition for call option
    boundary = [1(1)*0;zeros(N-3,1);u(N-1)*0];
    % Explicit iteration scheme
    for j=1:N-1
        if(j==1)
            Vnew(j) = d(j,k)*Vold(j) + u(j,k)*Vold(j+1);
        elseif(j<N-1)
            Vnew(j) = l(j,k)*Vold(j-1) + d(j,k)*Vold(j) +
u(j,k)*Vold(j+1);
        else
            Vnew(j) = l(j,k)*Vold(j-1) + d(j,k)*Vold(j);
        end
    end
    % Update the vectors from time k to time k+1
    Vold = Vnew + boundary;
end

```

The final option price will be computed using the inbuilt interpolation function in MATLAB.

```

% Interpolation to find the call price when S0 = 100
call_fdm = interp1(S,Vold,S0);

```

Section 1.04 Implicit Finite Difference Method (I-FDM)

The program for I-FDM will be similar to E-FDM. The code snippets from 10 – 16 will be the same except for the boundary condition, which we change as per the implicit discretization scheme. We set $V(S_{max}, \tau)$ as zero for the exact reason explained in the previous scheme.

Tridiagonal system is similar to the code discussed in week 4 spring lecture except that the lower, upper and main diagonal of the tridiagonal matrix has been discretized due to the discretization of sigma (discussed in Section 1.03). With the tridiagonal solver included in files, we use it to calculate the option prices and store in the vector as described below.

Code Snippet 19

```
for k=1:M
    % Dirichlet Boundary condition for European put option
    boundary = [-1(1)*0;zeros(N-3,1);-u(N-1,k)*0];
    % Solving the tridiagonal system for the implicit scheme
    %Vnew = tridiag(d,u,l,Vold+boundary);
    Vnew=tridiag(d(:,k),u(:,k),l(:,k),Vold+boundary);
    % Update the option prices from k to k+1
    Vold = Vnew;
end
```

Section 1.05 The Crank-Nicholson Method (CR-M)

The initial coding of this program i.e. parameterization and designation of appropriate variables for discretization process, is same as the previous two schemes. Exact code has been taken from Dr. Erick Dykeman's lecture code on pricing European Call Option using the CR-M, with the exception of discretization of lower, upper, and main diagonals of the implicit and explicit scheme used in the code.

Code Snippet 20

```
% lower, main and upper diagonal of the tridiagonal matrix for
the explicit finite difference scheme
lE = alpha - beta;
dE = 1 - r*dtau - 2*alpha;
uE = alpha + beta;
```

Code Snippet 21

```
% Vector to store the option prices at time k+1
Vnew = Vold;
temp = Vold;
```

The snippet below is self-explanatory with the comments in it.

Code Snippet 22

```
for k=1:M
    % Boundary Condition for Barrier Call option at time k+1
    boundary_I = [-1(1)*0;zeros(N-3,1);-u(N-1)*0];
    % Boundary Condition for Barrier Call option at time k
    boundary = [1(1)*0;zeros(N-3,1);uE(N-1)*0];
    % To make sure 2nd order convergence in time, a couple of
    implicit
```

```

    % scheme is implemented before switching to crank nicolson
scheme
    if(k<3)
        Vnew = tridiag(d(:,k),u(:,k),l(:,k),Vold+boundary_I);
    else
        % Crank Nicolson iteration scheme
        % The correct scheme is that A+I, hence instead of dE, we
need
        % to use dE+1
        % Explicit iteration scheme
        for j=1:N-1
            if(j==1)
                temp(j) = (dE(j,k)+1)*Vold(j) + uE(j,k)*Vold(j+1);
            elseif(j<N-1)
                temp(j) = lE(j,k)*Vold(j-1) + (dE(j,k)+1)*Vold(j) +
uE(j,k)*Vold(j+1);
            else
                temp(j) = lE(j,k)*Vold(j-1) + (dE(j,k)+1)*Vold(j);
            end
        end
        %temp = tridiag_prod(dE+1,uE,lE,Vold);
        Vnew = tridiag(d(:,k)+1,u(:,k),l(:,k),temp+boundary);
    end
    % Update the vectors from time k to time k+1
    Vold = Vnew;
endß

```

END USER'S INSTRUCTIONS

Section 1.06 Code Initialization and Test Run

Monte Carlo method

Name ▲	Value
alpha	0.3500
B	130
Barrier_mc	3.8870
CI_low	3.8801
CI_up	3.8939
dt	1.0000e-03
dtm	1x500 double
i	1000000
k	500
K	100
m	500
M	500
N	1000000
payoff	1000000x2 double
q	0.0500
r	0.0300
S0	100
S_antithetic	501x1 double
Sample	501x1 double
SE_C	0.0035
T	0.5000
tm	1x501 double
z	1.7140

Figure 3

On running the Monte Carlo code, the result we get is shown in the figure. The option price denoted by B_mc is an approximate of 3.89. Approximate because this is a simulation by random number generation, This value changes by the 2nd and 3rd significant figures every time the program is re-run. So, one must be careful while testing this code. Also, the upper and lower confidence interval defines the range in which our subsequent price results from other schemes must lie.

Now that we have a value for the option, the prices computed from the other three Finite Difference Schemes can be compared to M.C. price.

Explicit FDM

Name ▲	Value
a	0.3500
alpha	999x50000 dou
beta	999x1 double
boundary	999x1 double
call_fdm	3.8551
d	999x50000 dou
dS	0.1300
dtau	1.0000e-05
j	999
k	50000
K	100
l	999x50000 dou
M	50000
N	1000
q	0.0500
r	0.0300
S	999x1 double
S0	100
S1	1001x1 double
sigma	999x50000 dou
Smax	130
Smin	0
T	0.5000
tau	1x50001 double
u	999x50000 double
Vnew	999x1 double
Vold	999x1 double

Figure 4

The option price from the explicit scheme is 3.85 approximately. The price does not lie in the Monte Carlo confidence interval. As we increase the stock points the confidence interval shrinks. So, for lower stock points say $N = 1000$, the confidence interval will be larger and thus FDM prices will lie within the range. For the current N of one million the C.I. is too small and hence the the FDM prices will not lie within the range.

The implicit scheme is only stable for larger values of time steps. It is **very important** to not let the time step be a small value, such as 500. For this, MATLAB will return the values for **Vold** and **call_fdm** as **NaN**, an abbreviation for Not a Real Number. This is due to the infinitesimal values calculated for the payoff vector at time step $(k+1)$, a direct consequence of performing test run on smaller time steps. Hence, it is recommended to use reasonable number for time steps (as demonstrated in the code file BarrierExplicit.m).

Implicit FDM

Name ▲	Value
a	0.3500
alpha	999x50000 doub
beta	999x1 double
boundary	999x1 double
call_fdm	3.8551
d	999x50000 doub
dS	0.1300
dtau	1.0000e-05
j	999
k	50000
K	100
l	999x50000 doub
M	50000
N	1000
q	0.0500
r	0.0300
S	999x1 double
S0	100
S1	1001x1 double
sigma	999x50000 doub
Smax	130
Smin	0
T	0.5000
tau	1x50001 double
u	999x50000 doub
Vnew	999x1 double
Vold	999x1 double

Figure 5

CR-FDM

a	0.3500
alpha	999x5000 doub
beta	999x1 double
boundary	999x1 double
boundary_l	999x1 double
call_fdm	3.8551
d	999x5000 doub
dE	999x5000 doub
dS	0.1300
dtau	1.0000e-04
j	999
k	5000
K	100
l	999x5000 doub
lE	999x5000 doub
M	5000
N	1000
q	0.0500
r	0.0300
S	999x1 double
S0	100
S1	1001x1 double
sigma	999x5000 doub
Smax	130
Smin	0
T	0.5000
tau	1x5001 double
temp	999x1 double
u	999x5000 doub
uE	999x5000 doub
Vnew	999x1 double
Vold	999x1 double

Figure 6

As seen above in Figures 5 – 6 respectively, the resultant option price denoted by **call_fdm** is the same as the price computed from the explicit scheme. The price computed by all the Finite Difference Schemes are the exact same.

Anomalies & Convergence

The program will take longer time as the number of stock price samples in the vector, and the number of times steps will be increased.

As the (N) and (M) are increased, the option price computed using FDM converges to that of Monte Carlo. However, it never reaches the exact Monte Carlo price. This is the convergence property we intended to find out as this is the conclusion for the given project. We wanted to find whether the simulated stock price points using FDM ever converge to the Monte Carlo price points (simulated using a genuine stochastic process).

The difference from the Monte Carlo price will remain as the random factor is always present. Though our FDM prices fall within the confidence interval range as computed using Monte Carlo.

Hence we can say the project is a success and conclude our documentation.