Computational Methods: Assignment 1

Matthew Mercuri

Introduction and Purpose

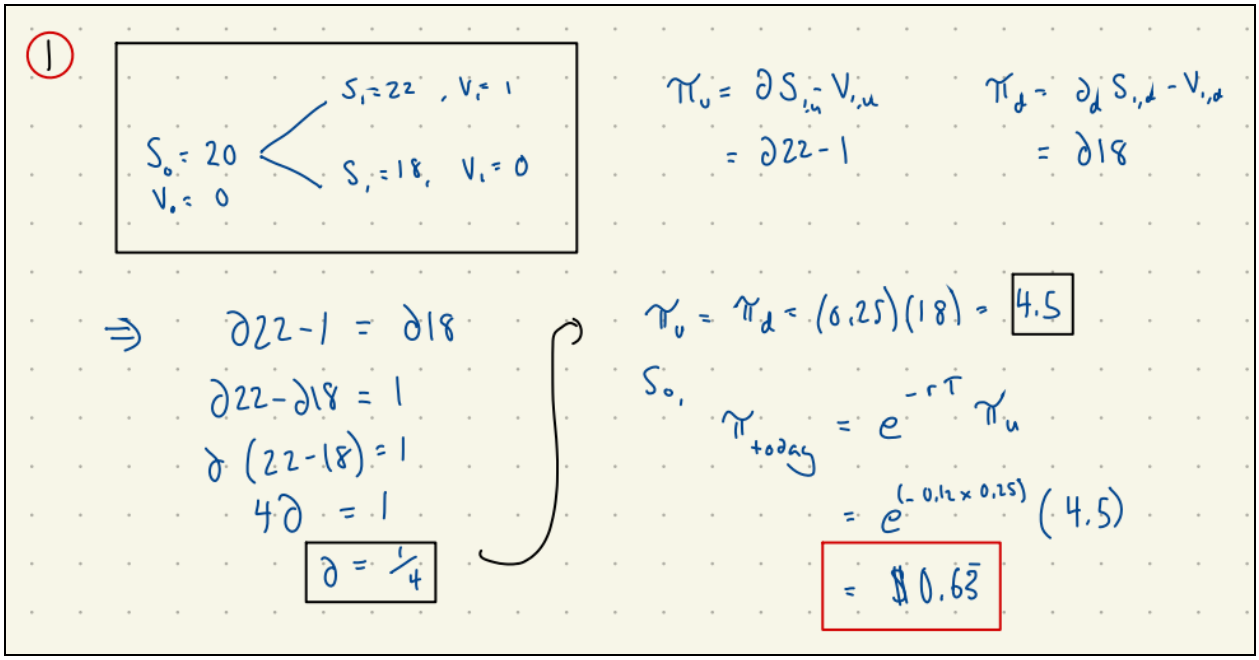
In this report, many techniques of computational methods are demonstrated. The report begins by using a non-arbitrage model to price a call option. Then, we work out proofs to analytical solutions for a pair of stochastic differential equations (SDEs). Afterwards, we demonstrate some of the properties of Brownian motion by generating paths from a standard normal distribution. Following that, we use an inverse cumulative distribution function (CDF) to generate random numbers from a Cauchy distribution using the uniform distribution. Next, we compute the value of a definite double integral using Monte Carlo integration. Lastly, we demonstrate the use of stochastic volatility in another option pricing example.

This report spans a great number of computational methods that can be used to solve a multitude of problems. Specifically, the applications of the methods suit mathematical finance. Practicing and demonstrating these skills is the motivation for this report.

1. Non-Arbitrage Option Pricing

Question:

Given the underlying stock price S = 20 today, its price has a 90% possibility to be S = 18 and a 10% possibility to be S = 22 in three months. Thus, based on the the non-arbitrage pricing principal, the fair price of a call option expiring in three month at strike price K = 21 is $0.633 with the annual risk free interest rate r = 12%. If the current market price of the call falls to $0.50, does there exist an arbitrage opportunity? If yes, construct a strategy to do the arbitrage and explain it.



See above, the derivation of the option's fair price given a non-arbitrage pricing model. The way we arrive to such a solution is as follows:

1. We try to construct a portfolio that cannot lose money, regardless of the stock's price in three months. We can accomplish this by being long or shirt with both shares and the option. The equations below show how we can obtain the value of such a portfolio. Essentially, the value of the portfolio today is made up of the stock's price today multiplied by delta shares, subtracted by the value of the call option today. This assumes we will be long the stock and short the option. If we can figure out what the portfolio's value will be in 3 months, we can use a discount term (*e* in the second equation) to find the present value of our portfolio. The goal here is to calculate how many shares of stock we would need to be long to take no price risk.

$$\Pi_{T=0} = \delta S_{T=0} - V_{T=0} \quad (1)$$

$$\Pi_{T=0} = e^{-rT} \Pi_{T=3} \quad (2)$$

2. Once we calculate how many shares of the stock we would need to purchase and assume no risk in price change, we can calculate the portfolio's value in three months. Which of course should be the same regardless of the price going up or down. Again, discounting the portfolio's value in 3 months gives us the present value.
3. Once we obtain the present value, we can use equation 1 to calculate the fair value of the call option, which we get to be **$0.63**.

Assuming the price changes to $0.50:

We can construct a strategy to take advantage of the situation and generate a risk-free profit. The procedure for which is outlined below:

1. We know the fair value of the portfolio today should be **$4.367**. Instead, the value is:

$$\Pi_{T=0} = \delta S_{T=0} - V_{T=0}$$

$$\Pi_{T=0} = (0.25)(20) - 0.50$$

$$\Pi_{T=0} = \$4.50$$

2. Given the value of the portfolio is more than the fair value, we short the portfolio by buying the option and shorting the stock (**+$4.50**).
3. We then take the proceeds of the short sale and invest it at the risk-free rate of 12% for three months (value at end of three months: **$4.637**):

$$\Pi_{T=3} = e^{rT}(4.50)$$

$$\Pi_{T=3} = e^{0.12 \cdot 0.25}(4.50)$$

$$\Pi_{T=3} = \$4.637$$

4. After the three months, we buy back the portfolio for **$4.50** (which we know would be the price based off of previous calculations.

This yields a **riskless profit of ~$0.14** ($4.637-$4.50).

2. Analytical Solutions of Special SDEs

(2) a) $F_t = e^{Z_t - \frac{1}{2}t}$ solves $dF_t = F_t\, dZ_t$

let $f(F_t) = \log(F_t)$

$$\Rightarrow df = f'(F_t)\,dF_t + \frac{1}{2} f''(F_t)\,(dF_t)^2$$

$$= \frac{1}{F_t}(F_t\, dZ_t) - \frac{1}{2}\frac{1}{F_t^2}\left(F_t^2\, dZ_t^2\right)$$

$$df = dZ_t - \frac{1}{2}\, dZ_t^2$$

$$f = \int dZ_t - \frac{1}{2}\int dt$$

$$= Z_t + \frac{1}{2}t$$

$$\Rightarrow e^f = F_t$$

$$\boxed{F_t = e^{Z_t - \frac{1}{2}t}}$$

b) $F_t = e^{2z_t - t}$ solves $dF_t = F_t\,dt + 2F_t\,dz_t$

Let $G(F_t) = \log(F_t)$

$$\Rightarrow dG = G'(F_t)\,dF_t + \tfrac{1}{2}G''(F_t)(dF_t)^2$$

$$= \frac{1}{F_t}(F_t\,dt + 2F_t\,dz_t) - \frac{1}{2}\frac{1}{F_t^2}(F_t\,dt + 2F_t\,dz_t)^2$$

$$= dt + 2\,dz_t - \tfrac{1}{2}\frac{1}{F_t^2}(F_t^2\,dt^2 + 4F_t^2\,dt\,dz_t + 4F_t^2\,dz_t^2)$$

$$\int dG = \int\left(dt + 2\,dz_t - \left[\tfrac{1}{2}\cancel{dt^2}^{\,0} + 2\,\cancel{dt\,dz_t}^{\,0} + 2\,dz_t^2\right]\right)$$

$$G = t + 2z_t - 2t$$

$$= 2z_t - t$$

$$\Rightarrow e^G = F_t$$

$$\boxed{F_t = e^{2z_t - t}}$$

3. Properties of Brownian Motion

The following script was used to evaluate the system described by:

Let

$$\begin{aligned} \Delta Z_i &\equiv Z(t_{i+1}) - Z(t_i) = \phi(t_i)\sqrt{\Delta t} \\ \phi(t_i) &\sim N(0,1) \\ Z(0) &= 0 \end{aligned}$$

and

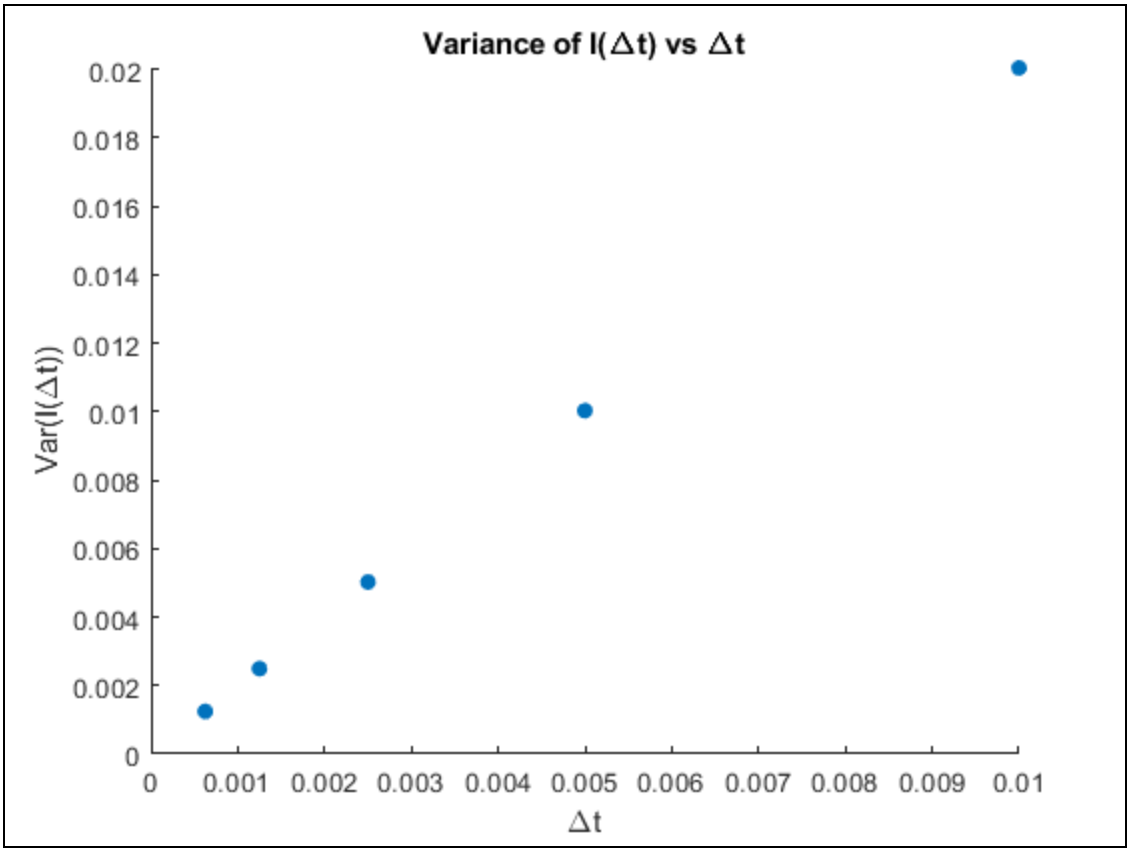
$$I(\Delta t) = \sum_{i=0}^{i=K-1} (\Delta Z_i)^2$$

$$\Delta t = \frac{T}{K}$$

Throughout the code, notice the comments that specify what the goal of each block is.

```
% Matthew Mercuri
% MTH 600 - Assignment 1
% Question 3

paths = 100000; % defining the number of paths to use

% generating empty results vectors
I_results1 = zeros(1, paths);
I_results2 = zeros(1, paths);
I_results3 = zeros(1, paths);
I_results4 = zeros(1, paths);
I_results5 = zeros(1, paths);

% creating vectors for timesteps (different delta t from k)
ts1 = linspace(0, 1, 100);
ts2 = linspace(0, 1, 200);
ts3 = linspace(0, 1, 400);
ts4 = linspace(0, 1, 800);
ts5 = linspace(0, 1, 1600);


% calculating delta t for each timestep
delta_t1 = 1/100;
delta_t2 = 1/200;
delta_t3 = 1/400;
delta_t4 = 1/800;
delta_t5 = 1/1600;

% simulating paths
for p = 1:paths

    % for 100 timesteps
    rv1=normrnd(0, 1, [1,100]); % generating the required number of std norm RVs
    del_z1 = rv1*sqrt(delta_t1); % applying the formula for delta z of i
    del_z1_sq = del_z1.^2; % squaring every entry in sequence

    I1 = sum(del_z1_sq); % summing for solution of I

    I_results1(p) = I1; % saving value for I in results vector

    % for 200 timesteps
    rv2=normrnd(0, 1, [1,200]);
    del_z2 = rv2*sqrt(delta_t2);
    del_z2_sq = del_z2.^2;

    I2 = sum(del_z2_sq);

    I_results2(p) = I2;

    % for 400 timesteps
    rv3=normrnd(0, 1, [1,400]);
    del_z3 = rv3*sqrt(delta_t3);
```

```
    del_z3_sq = del_z3.^2;

    I3 = sum(del_z3_sq);

    I_results3(p) = I3;

    % for 800 timesteps
    rv4=normrnd(0, 1, [1,800]);
    del_z4 = rv4*sqrt(delta_t4);
    del_z4_sq = del_z4.^2;

    I4 = sum(del_z4_sq);

    I_results4(p) = I4;

    % for 1600 timesteps
    rv5=normrnd(0, 1, [1,1600]);
    del_z5 = rv5*sqrt(delta_t5);
    del_z5_sq = del_z5.^2;

    I5 = sum(del_z5_sq);

    I_results5(p) = I5;

end

% calculating mean and variance of resulting paths
I_results1_mean = mean(I_results1);
I_results1_var = var(I_results1);

I_results2_mean = mean(I_results2);
I_results2_var = var(I_results2);

I_results3_mean = mean(I_results3);
I_results3_var = var(I_results3);

I_results4_mean = mean(I_results4);
I_results4_var = var(I_results4);

I_results5_mean = mean(I_results5);
I_results5_var = var(I_results5);


% saving final results to vectors and plotting them
x = [delta_t1 delta_t2 delta_t3 delta_t4 delta_t5];
y = [I_results1_var I_results2_var I_results3_var I_results4_var
   I_results5_var];

   scatter(x,y,'filled')
   title('Variance of I(\Deltat) vs \Deltat')
   xlabel('\Deltat')
   ylabel('Var(I(\Deltat))')
```



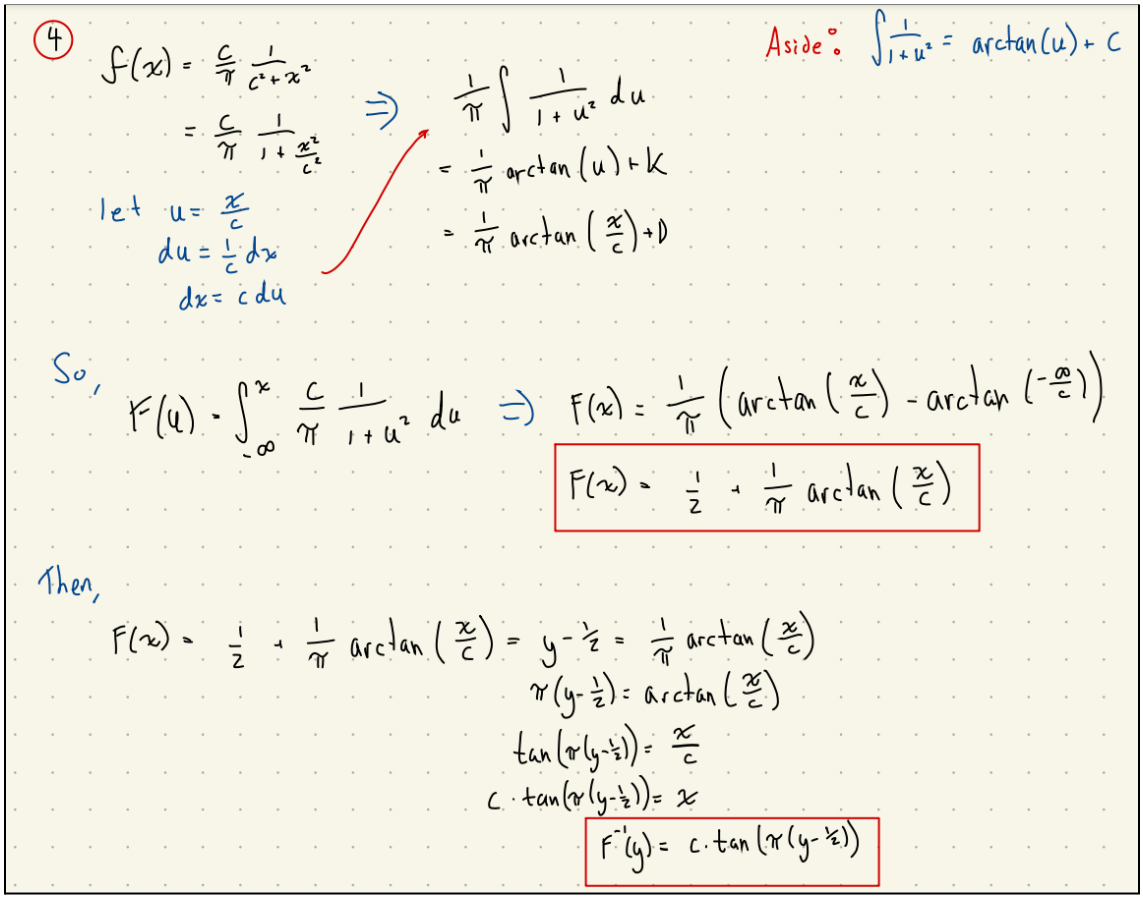
Through the above graph, we notice that as delta *t* approaches 0 the variance of *I* approaches zero as well. Essentially, this empirically means that as the difference in time gets infinitesimally small, the function, *I*, is deterministic. This of course is directly related to the derivation of Ito's Lemma. The result arises from the quadratic variation of a stochastic process, in this case, Brownian motion. This verifies that terms $dB_t dt$ and dt^2 can be set to zero to arrive at Ito's Lemma. The **aforementioned terms tend to zero faster than dB_t^2 which can be set to dt.**

4. Generating Random Numbers for a Cauchy Distribution

Computers are able to generate random numbers from any probability distribution by modifying the distribution's CDF. In this case, we use the CDF for the Cauchy distribution to generate Cauchy distributed numbers. To achieve this, we first find the inversion of the CDF. Then we generate numbers from the standard uniform distribution. Essentially, these act as percentiles that will be used as the parameter to input into the inverse CDF. After inputting the uniform values, the inverse CDF will generate a number from the Cauchy distribution.

First, we find the CDF and inverse CDF of the Cauchy distribution, given its PDF:

$$f(x) = \frac{c}{\pi}\frac{1}{c^2+x^2}$$



The following MATLAB code generates Cauchy distributed numbers from a standard uniform distribution. Again, it is well documented to explain what each particular block of code achieves:

```
% Matthew Mercuri
% MTH 600 - Assignment 1
% Question 4

p = cauchy_rv(10, 1);
disp(p)

% The function below uses a uniform distribution to return value from a
% Cauchy distribution. It inputs the number of cauchy values you wish to
% have and also the value for c (1 for standard Cauchy distribution
function cauchy_rvs = cauchy_rv(n, c)
    % creating a zeros vector for as many RVs requested through param n
    cauchy_rvs = zeros(1, n);

    % creating a vector of uniform RVs to use for Cauchy RVs
    u_rvs = unifrnd(0, 1, 1, n);

    % calculating cauchy RVs
    for i = 1:n
        u = u_rvs(1, i); % picking uniform RV
        c_rv = c*tan(pi*(u-0.5)); % calculating cauchy RV from inverse CDF using
    u
        cauchy_rvs(1, i) = c_rv; % saving result to vector for return
    end
end
```

5. Monte Carlo Integration

Given an integral:

$$I = \int_a^b \int_c^d f(x, y) dy dx$$

The procedure for higher dimension integration using the Monte Carlo computational method is as follows:

1. Define the amount of simulations you would like to perform (in this case we use N = 100, 500, 2000, 3000)

2. For each simulation, generate a standard uniform random variable
3. Scale each random variable using (replace parameters for y):

$$x_i = a + \xi(b - a) \text{ where } \xi \text{is the standard uniform RV}$$

4. Compute *f(x, y)* using the generated values
5. Store the computed result
6. Determine the *domain* of the integral though:

$$m = (b - a) \cdot (d - c)$$

7. Find the expectation of *f* by summing all the results for *f* and dividing it by the number of results (taking the empirical mean)
8. To get the final value of *I*, multiply the expectation of *f* by the *domain m*

The above procedure is for classical Monte Carlo integration. In practice, we can apply the *variance reduction* technique of using *antithetic variates*. The difference being, upon computing the uniform random variables for *x* and *y*, subtract them from 1 to obtain a new random variable with much less computational cost. We then compute the integral using them as well. Essentially, using this technique for each simulation we obtain new RVs:

$$x_i^{'} = 1 - x_i$$

$$y_i^{'} = 1 - y_i$$

In this section, we use Monte Carlo integration to compute:

$$\int_0^2 \int_0^{1/2} xy^2 dydx$$

First, it is worth mentioning that computing this analytically, we obtain the result $0.0\overline{83}$. We will compare this result to the results of the Monte Carlo integrations. What follows is the code that

is used to perform the Monte Carlo integration. Although it is documented, it essentially follows the procedures outlined above.

Using classical technique:

```
% Matthew Mercuri
% MTH 600 - Assignment 1
% Question 5a

simulations = [100 500 2000 3000]; % creating a vector of all simulations to select later
results = zeros(1, length(simulations)); % creating a zero vector to store results of MC int.

for i = 1:length(simulations) % performing for the 4 different amounts of simulations
    sims = simulations(1, i); % selects the amount of simulations to be done
    sim_results = zeros(1, sims); % creates a zero vector to store results

    for s = 1:sims % executes once for each simulation
        y_i = rand*0.5; % select and scales a standard uniform RV for y
        x_i = rand*2; % same as above for x

        f_i = x_i*(y_i^2); % computes value for f

        sim_results(1, s) = f_i; % stores value for f before proceeding
    end

    m = 2*0.5; % calculating integral domain
    E_f = (1/sims)*sum(sim_results); % finding expected value of f
    I = m*E_f; % computing value of integral

    results(1, i) = I; % storing final result
end

disp(results) % displaying final result
```

Using antithetic variates:

```
% Matthew Mercuri
% MTH 600 - Assignment 1
% Question 5b

% NOTE: the code that follows is essentially the same as what is done in
% question 5a. The differences:
% - for each simulation we store two results since we are using antithetic
% variates
% - we store rand to its own variable before scaling so we can use it as
% needed in the antithetic variates technique
% - we compute two values for f during each simulation
% - after flattening our results array, we average over double the amount of
% simulations

simulations = [100 500 2000 3000];
results = zeros(1, length(simulations));

for i = 1:length(simulations)
    sims = simulations(1, i);
    sim_results = zeros(2, sims);

```

```
    for s = 1:sims
        y_rand = rand;
        x_rand = rand;

        y_i = y_rand*0.5;
        x_i = x_rand*2;

        f_i = x_i*(y_i^2);

        y_i_2 = (1-y_rand)*0.5;
        x_i_2 = (1-x_rand)*2;

        f_i_2 = x_i_2*(y_i_2^2);

        sim_results(1, s) = f_i;
        sim_results(2, s) = f_i_2;
    end

    sim_results = [sim_results(1, :) sim_results(2, :)];

    m = 2*0.5;
    E_f = (1/(2*sims))*sum(sim_results);
    I = m*E_f;

    results(1, i) = I;
end

disp(results)
```

Results:

After running each method once, the yielded results are:

Classical			
0.0925	0.0823	0.0798	0.0837

Antithetic Variates			
0.0920	0.0831	0.0823	0.0825

Looking at the tables above, we can make some general statements regarding the advantages of using a variance reduction technique. For one, it does seem like the lower the number of simulations, *N*, the better the antithetic variates technique is compared to the classical technique. So at small values of *N*, using this variance reduction technique is a good idea. Really, this means that the technique is most valuable when computing costs are an important consideration.

6. Stochastic Volatility Model

The value of a European call option at expiry is given by the formula:

$$C_T = max(S_T - K,\ 0)$$

Where:

S_T is the stock's price at expiry

K is the strike price of the option

We can find the initial value of the option by finding the value of the option at expiry, and then discounting the result back to time zero, given the risk-free rate. Of course, there is no way to know for certain what a stock's price will be at expiry. To price options using Monte Carlo, we must use a model to generate possible paths for how the stock will move into the future. After simulating many paths, we can use them to find the expected value of the stock's price at expiration. In this section, we will use the Heston stochastic volatility model (Heston, 1993) for such a purpose. The model is defined below:

$$\frac{dS}{S} = rdt + \sqrt{v}dZ^{(1)}$$
$$dv = -\lambda(v - \bar{v})dt + \eta\sqrt{v}dZ^{(2)}$$
$$\mathbf{E}(dZ^{(1)}dZ^{(2)}) = \rho dt$$

In the above formulas, *S* is the stock's price, *r* is the risk-free rate, *v* is the variance, $\bar{v}$ is the long-term mean variance, and λ is the speed of reversion. We can extend the model to help us generate paths using the Monte Carlo method:

$$x_{i+1} = x_i + r\Delta t - \frac{v_i}{2}\Delta t + \sqrt{v_i \Delta t}\phi_i^{(1)}$$

$$v_{i+1} = \left(\sqrt{v_i} + \frac{\eta}{2}\sqrt{\Delta t}\phi_i^{(2)}\right)^2 - \lambda(v_i - \bar{v})\delta t - \frac{\eta^2}{4}\Delta t$$

$$E(\phi_i^{(1)}\phi_i^{(2)}) = \rho$$

Looking above, notice that there are two standard normal variables that are correlated by ρ. We can use the following procedure to ensure that we generate correlated standard random normal variables in our MATLAB script:

1. Generate two standard normal variables, x_1 and x_2
2. Use:

$$y_i = \rho \cdot x_1 + \sqrt{1 - \rho^2 \cdot x_2}$$

3. Both x_1 and y_1 are correlated by rho and are standard normal random variables

In this section we will be using the Heston stochastic volatility model to price an option with the given parameters:

r	.02
Initial variance v	0.0174
$\bar{v}$	0.0354
η	0.3877
ρ	- 0.7165
λ	1.3253
Initial asset price S^0	$100
Strike K	$100
Expiry T	1

The MATLAB code that achieves solving the option pricing follows:

```
% Matthew Mercuri
% MTH 600 - Assignment 1
% Question 6

% defining constants
simulations = 10000;
del_t = 1/100;
rho = -0.7165;
r = 0.02;
v_i = 0.0174;
x_i = log(100);
n = 0.3877;
v_bar = 0.0354;
lambda = 1.3253;
K = 100;

% ceating empty zero vector to store results
S_T = zeros(1, simulations);

% simulating paths for the stock, given Heston's model
for s = 1:simulations
    % creating empty vectors to store the stock price and volatility at
    % each timestep
    xs = zeros(1, (1/del_t));
    vs = zeros(1, (1/del_t));

    % setting the initial values for x (log of S) and v (volatility), both
    % are given
    x = x_i;
    v = v_i;

    % storing the initial values in the results vector
    xs(1, 1) = x;
    vs(1, 1) = v;

    % looping to generate values for each timestep
    for t = 2:(1/del_t)
        % generating two, correlated standard normal RVs
        z_1 = normrnd(0,1);
        z_1_prime = normrnd(0,1);
        z_2 = (rho*z_1)+(sqrt((1-(rho^2)))*z_1_prime);

        % calculating the value of x and v for the timestep, given Heston's
        % formulas
        x = x+(r*del_t)-((v/2)*del_t)+(sqrt(v*del_t)*z_1);
        v =
    ((sqrt(v)+((n/2)*sqrt(del_t)*z_2))^2)-(lambda*(v-v_bar)*del_t)-(((n/2)^2)*del_t
    );

        % should the volatility be less than zero (cannot happen), we set
        % it equal to the absolute value of itself
        if v < 0
            v = abs(v);
        end

        % storing the result for x and v
        xs(1, t) = x;
        vs(1, t) = v;
    end
    % adding the final price at T=1 to the final results vector
    S_T(1,s) = exp(xs(end));
```

```
end

% calculating expected value of S at T=1 (expected value given the
% simulated paths) and then displaying result
E_S_1 = (1/simulations)*sum(S_T);
disp(E_S_1)

% finding the value of the call option at T=1
value_1 = E_S_1-K;
disp(value_1)

% discounting result to T=0
value_0 = exp(-r)*value_1;
disp(value_0)
```

After running the code, the value of the European call option at T=1 is $2.02. Discounting this value to find the initial value of the option, we obtain the result that the option is worth **$1.98**.

Result

$$S_T = \$102.02$$

$$C_T = \$102.02 - \$100 = \$2.02$$

$$C_0 = e^{-rT} \cdot \$2.02 = e^{-0.02} \cdot \$2.02 = \$1.98$$

References

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