Assignment 3

Monte Carlo methods vs. Finite difference methods

UPPSALA UNIVERSITET

Computational Finance: Pricing and Valuation

Authors:

Edward Glöckner
David Hammar
Karl Westermark

Uppsala
October 5, 2022

Contents

1 Introduction

Black-Scholes is a model within mathematical finance used for modelling financial markets containing investment instruments. From the model one can derive the Black-Scholes equation which gives a theoretical value of the price of an European-style option.

The CEV model (constant elasticity of variance) is a stochastic volatility model used in the financial industry. The main applications of the model is to price options, commodities and equities.

The task of this paper is to price an European call option using the CEV model. The underlying asset is assumed to follow the following stochastic differential equation:

$$dS_t = rS_t dt + \sigma S_t^{\gamma} dW_t \tag{1}$$

In this equation S is the spot price of the underlying asset to the option, t is the time, r is the drift σ is the volatility and W is brownian motion. γ is the main feature of the model, and controls the relationship between the volatility and the price. If $\gamma < 1$ the volatility and the price are inversely related.

There are various types of numerical methods that can be used to approximate the CEV model. In this paper the Monte-Carlo method, the explicit finite difference method and the implicit finite difference method will be implemented in MATLAB. Lattice method will explained and discussed but will not be implemented in MATLAB.

2 Numerical methods - overview

2.1 Finite difference methods

By defining a grid in space S and time t, the Black-Scholes PDE can be solved, numerically finding V(S,t), using Finite Differences. Using Taylor expansions for discretizing the differentials of the Black-Scholes PDE yields two ways of solving the the PDE, an Explicit and an Implicit. These are defined in equation (2) and (3).

$$Explicit : V_{k-1} = A_{explicit} V_k + b \tag{2}$$

$$Explicit : V_{k-1} = A_{implicit}^{-1} V_k - b \tag{3}$$

These are both iterated from time $k- > 0$, finding $V_0 = V(S, 0)$. A is a matrix, and b are the boundary conditions.

2.2 Monte-Carlo methods

The Monte-Carlo methods for solving Black Scholes SDE are stochastic simulations of the options price movement from an arbitrary S_0 to S_T. Using Brownian-motion to simulate the movement in time of the option, a final price of V_T can be estimated. This singular estimate becomes more accurate if the simulated movement has smaller steps in time-space, dt.

Since this estimate is stochastic, its accuracy can be improved upon by simply repeating the process N number of times. The Law of Large Numbers states that the estimate becomes closer to the true value by using the mean estimate of V_T as N goes to infinity. Using the estimated V_T an estimation of V_0 can be obtained by discounting by a factor e^{-rT}.

2.3 Lattice methods

The Lattice method or the binomial method is an options pricing model that is popular in finance given its' flexibility and simplicity. The binomial method uses, much like other numerical methods, an iterative procedure to produce points in time between valuation date and the options expiration date. The method relies on the following assumptions:

- From time step t_i to time step t_{i+1} the price S_i can take only one of two paths, $S_i * u$ or $S_i * d$ where u and d are upward- and downward movement respectively.
- The probability of and upward movement is p.
- Expectation and variance of the price S refer to the continuous counterparts evaluated for the discount rate r.

Moving forward, pricing an option using this method consists of three key steps. These are:

1. Create a binomial price tree.
2. Finding the option value at each of the final nodes.
3. Finding the option value at the earlier nodes.

2.4 "Greeks" - Delta

"Greeks" are a way of measuring the sensitivity of the price of an underlying option, and are derivatives of the pay-off function V(S,t) with respect to dt and ds. In this assignment we calculate one of the most common "Greeks", Delta. Delta is defined in equation (4)

$$\Delta = \frac{\partial V(S,t)}{\partial s} \tag{4}$$

For both Monte-Carlo and Finite Differences the numerical Delta are calculated with equation (5), where index i and s represents time and s respectively

$$\frac{\partial V_{i,s}}{\partial s} = \frac{V_{i,s+ds} - V_{i,s-ds}}{2ds} \tag{5}$$

2.5 Analytically Derived Soultions

There are analytically derived solutions to Black-Scholes equation and Black-Scholes Delta, that are valid for the standard non-CEV version (e.g. $\gamma = 1$). These have been used in order to enable comparisons between Monte-Carlo and Finite Difference methods of solving the PDE. These analytical solutions are stated in equations (6) through (10), where erf(x) is the Error Function:

$$V(S,t) = SF(d_1) - KF(d_2)e^{-r(T-t)} \tag{6}$$

$$d_1 = \frac{log(S/K) + (r + \sigma^2/2)(T - t)}{\sigma(\sqrt{T - t})} \tag{7}$$

$$d_2 = d_1 \sigma(T - t) \tag{8}$$

$$\Delta = F(d_1) \tag{9}$$

$$F(x) = \frac{1}{2}(1 + erf(\frac{x}{\sqrt{2}})) \tag{10}$$

3 Numerical results

By implementing the Black-Scholes equation in MATLAB we can obtain a theoretical estimate of the option value. We use our function (A.2) in the appendix with the following parameters:

- σ (volatility) = 0.25
- r (drift) = 0.1
- E (strike price) = 15,
- T (time to maturity) = 0.5
- s (spot price) = 14

```matlab
theoretical = bsexact(0.25, 0.1, 15, 0.5, 14);
disp("Theoretical option value: " + theoretical);
```

which gives us:

Theoretical option value: 0.86701

By a discretisation of the time points and the spot price we can create a two dimensional plane. On each point on the plane we can calculate the theoretical estimate of the option value using the Black-Scholes equation. We use the function (A.2) in the appendix to calculate the option values, and plot the results

```matlab
S0 = 14;        % Start value
K = 15;         % Strike price
r = 0.1;        % Risk free interest rate
sigma = 0.25;   % Volatility
T = 0.5;        % Maturity

dt = 0.002;
dx = 0.7;

[V_analyti, spat, time] = black_scholes(S0, K, r, sigma, dt,
    dx, T);
time_to_expire = T-time;

figure(1)
surf(time_to_expire, spat, V_analyti)
xlabel('Time to expire (T-t)'), ylabel('Stock price (S)'),
    zlabel('Option Price (V)')
title("Black Scholes solver for European Call option");
```

which gives us:

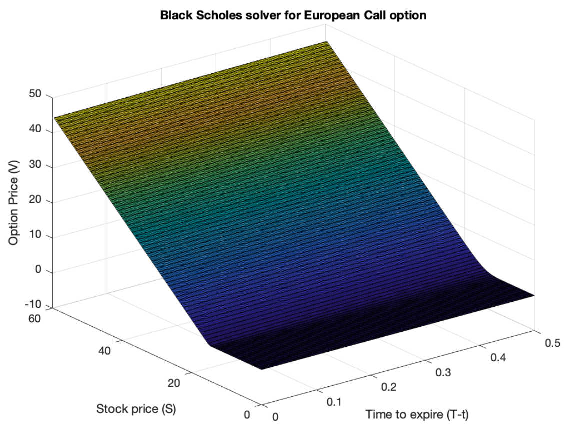


Figure 1: Option value as a function of time to maturity and spot price

As we can see the option expires useless when the spot price of the underlying asset is lower than the strike price. Since this is an European call option this results is what we expect.

3.1 Monte-Carlo methods

By a discretisation of the time we can run a Monte-Carlo simulation to estimate the option value. We use function (A.3) in the appendix to calculate the option value

```
S0 = 14;
K = 15;
r = 0.1;
sigma = 0.25;
T = 0.5;
gamma = 1;

N_samples = 100000;
n_timeponts = 100;
V_vec = STD_solverv1(N_samples, n_timeponts, T, S0, sigma,
    gamma, K, r);
V_est = exp(-r * T) * mean(V_vec);

disp("Estimated price of the option: " + num2str(V_est));
```

which gives us:

Estimated price of the option: 0.86624

3.1.1 Accuracy

The error of the numerical estimation using the Monte-Carlo method is made up of two parts: the sample error which is due to the fact that you can't use an ∞ number of sample paths, and the discretization error/bias which is due to the fact that you can't use an ∞ number of time-points.

To obtain the sample error as a function of the number of sample paths we estimate the option price using different amount of sample paths. Then we can calculate the error using the theoretical estimate obtained by the Black Scholes equation. Since we keep the timestep low, we can assume the discretisation error to be negligible.

```
S0 = 14;
K = 15;
r = 0.1;
sigma = 0.25;
T = 0.5;
gamma = 1;

V_analytical = bsexact(sigma, r, K, T, S0); % Theoretical
   estimate for the given parameters.

N_samples_vec = [1000, 10000, 100000, 1000000, 5000000];
V_est_vec1 = zeros(1,length(N_samples_vec));

n_timeponts = 256;

for k = 1:length(N_samples_vec)

    N_samples = N_samples_vec(k);
    V_vec1 = STD_solverv1(N_samples, n_timeponts,T, S0, sigma
   , gamma, K, r);
    V_est_vec1(k) = exp(-r * T) * mean(V_vec1);
end

err_sample1 = abs(V_analytical -  V_est_vec1);
figure()
loglog(N_samples_vec, err_sample1)
xlabel("Number of samples")
ylabel("Absolute error")

p = polyfit(log(N_samples_vec), log(err_sample1),1);
disp("Estimated rate of convergence: " + num2str(-p(1)))
```

which gives us:

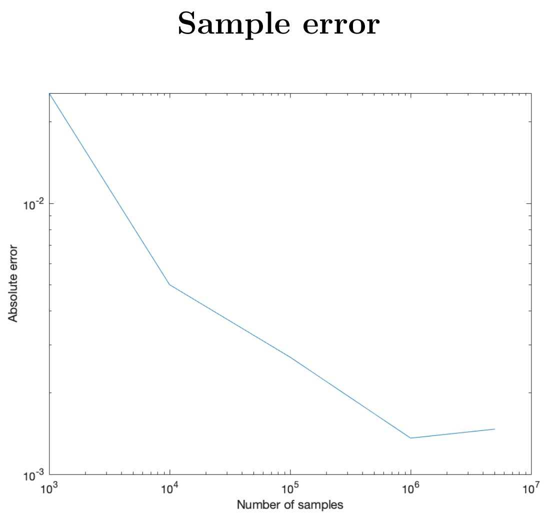


Figure 2: Sample error as a function of number of samples

and

Estimated rate of convergence: 0.32994

The sample error should be going down with an increase in the number of samples, which our results confirm.

Similarly we can calculate the discretization error as a function of the time step. We calculate different estimates of the option price and compare with the result from the Black Scholes equation, and calculate the error. When calculating the discretization error we keep the number of samples high so we can neglect the sample error.

```
S0 = 14;
K = 15;
r = 0.1;
sigma = 0.25;
T = 0.5;
gamma = 1;

V_analytical = bsexact(sigma, r, K, T, S0); % Theoretical
    estimate for the given parameters.

N_samples = 5000000;
n_points_vec = 2 .^ [0, 1, 2, 3, 4, 5, 6, 7, 8,9];
V_est_vec1 = zeros(1,length(n_points_vec));

for k = 1:length(n_points_vec)

```

```
    n_timeponts = n_points_vec(k);
    V_vec1 = STD_solverv1(N_samples, n_timeponts, T, S0,
   sigma, gamma, K, r);
    V_est_vec1(k) = exp(-r * T) * mean(V_vec1);
end

err_time1 = abs(V_analytical -  V_est_vec1);

figure()
loglog(n_points_vec, err_time1)
xlabel("Number of timepoints")
ylabel("Absolute error")

p = polyfit(log(n_points_vec), log(err_time1),1);
disp("Estimated rate of convergence: " + num2str(-p(1)))
```

which gives us:

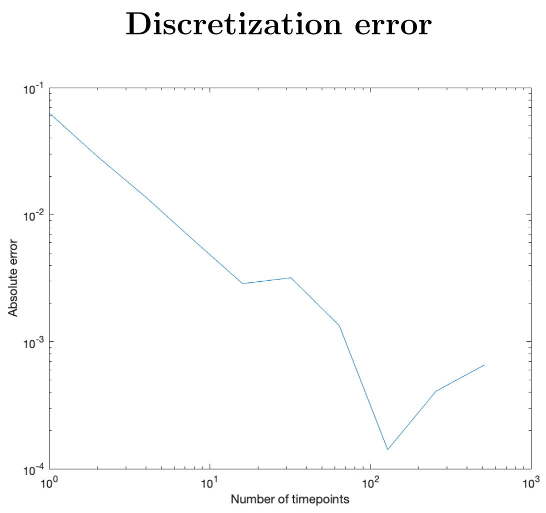


Figure 3: Discretization error as a function of the time step

and

"Estimated rate of convergence: 0.85934"

The discretization error should be going down when the number of time points increases, which our results confirms.

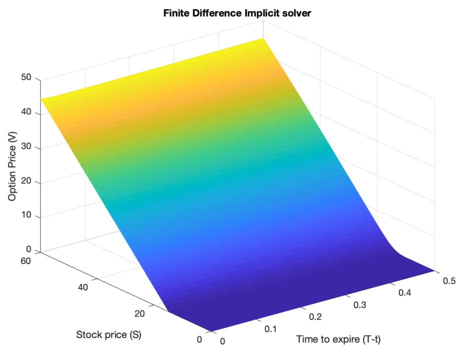
3.2 Finite difference methods

For the finite difference methods we discretize the time values and the spot price values. We use function (A.4) and (A.5) to estimate the option values using the explicit and the implicit method.

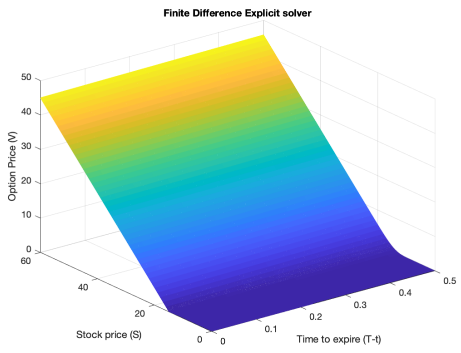
```
S0 = 14;        % Start value
K = 15;         % Strike price
r = 0.1;        % Risk free interest rate
sigma = 0.25; % Volatility
T = 0.5;        % Maturity

dt = 0.001;
dx = 1;
gamma = 1;

[V_finite_ex, spatial_points_ex, time_points_ex] =
    CEV_Solver_Explicit(S0,...
     K, T, dt, dx, sigma, r, gamma);
time_to_expire_ex = T-time_points_ex;
[V_finite_im, spatial_points_im, time_points_im] =
    CEV_Solver_Implicit(S0,...
     K, T, dt, dx, sigma, r, gamma);
time_to_expire_im = T-time_points_im;

figure(1)
mesh(time_to_expire_ex, spatial_points_ex, V_finite_ex);
xlabel('Time to expire (T-t)'), ylabel('Stock price (S)'),
    zlabel('Option Price (V)')
title("Finite Difference Explicit solver");

figure(2)
mesh(time_to_expire_im, spatial_points_im, V_finite_im);
xlabel('Time to expire (T-t)'), ylabel('Stock price (S)'),
    zlabel('Option Price (V)')
title("Finite Difference Implicit solver");
```

which gives us:

Estimate of option value using finite difference methods



(a) Implicit Finite Difference



(b) Explicit Finite Difference

3.2.1 Accuracy

The error of the numerical estimation using the Finite Difference schemes is made up of discretization error for both the spatial step and time step. These occur since we cannot use an ∞ number of time- and spatial-points.

Similaraly as for the Monte-Carlo method we can calculate the discretization error as a function of the time-step. We keep the spatial step small so we can neglect the discretization error in the spatial step, and calculate different estimates of the option value using different time-steps, and compare to the theoretical estimate.

```
%%Input paramaters
clear all
close all
clc
%  inputs
S_Max = 60;     %Max S, 4*K.
S_Min =0;
S_0 = S_Max/2;
K = 15;
r = 0.1;
sigma = 0.25;
T = 0.5;
gamma = 1;
t0=7;
error = zeros(2,t0)
for t= 1:1:t0

    N=10000;                   %Timesteps
    M=100*t;                   %Steps S-space.

    dt=T/N;              %Resulting time-step.
    ds=(S_Max-S_Min)/M; %Resulting S-space-step (not
   laplacian, but...option-space)

    m = 0:M-1;

    p1 = 0.5*dt.*(r.*m - sigma^2.*m.*m.^gamma);
    p2 = 1 + dt*(sigma^2.*m.*m.^gamma + r);
    p3 = -0.5*dt.*(r.*m + sigma^2.*m.*m.^gamma);


    V(1:M,1:N) = nan;
    V(1:M,1)=max((S_Min+(m+1)*ds-K),zeros(size(1:M)))';
    V(1,:) = 0;

    V(M,2:N)=(S_Max+S_Min)-K*exp(-r*(1:1:N-1)*dt);
    z=V;

    A = diag(p1(3:M-1),-1) + diag(p2(2:M-1)) + diag(p3(2:M-2)
   ,1);
    [L,U] = lu(A);


    boundary = zeros(size(A,2),1);
    for  k =  1:1:(N-1)%
        boundary(1) = p1(2)*V(1,k);
        boundary(end) = p3(M-1)*V(M,k);



        V(2:(M-1),k+1) = U\(L\(V(2:(M-1),k) - boundary));
    end

    V = fliplr(V);
    figure(1)
    mesh(S_Min+ds*(0:M-1),dt*(0:N-1),V(1:M,1:N)');
    xlabel('S')
    ylabel('t')

    V;

    E = K; s = 0;

    %error = abs(A((M),1) - V_0_exact)

    for j=1:M
        s = s+ds;
        V_0_exact = bsexact(sigma, r, E, T, s);
        errorJ(j) = abs(V((j), 1) - V_0_exact);
    end

    error(1,t) = mean(errorJ);
    error(2,t) = ds;
end


figure(3)
loglog(error(2,:),error(1,:),'',linspace(error(2,t0),error
    (2,1),10),(linspace(error(2,t0),error(2,1),10)))
xlabel('ds where delta-t is small.')
ylabel('Absolute error of V_0')
title('Absolute error of V_0, with respect to ds.')
legend('Abs(V_0-V exact)','ds^2')
```

which gives us

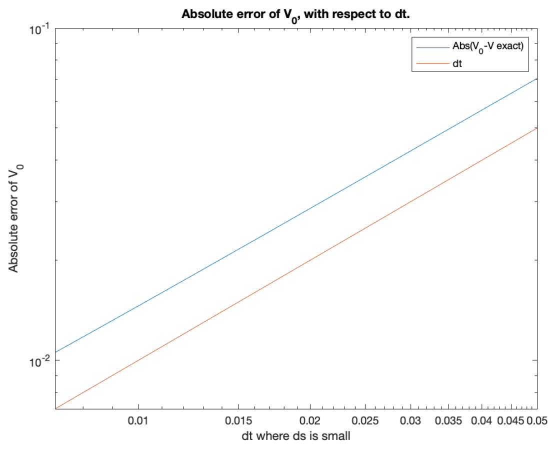


Figure 5: Error with respect to the time step

As the time-step gets larger and larger the number of time-points get lower, which should lead to an increase in the error, which is what our result confirms.

Similaraly we can calculate the discretization error with respect to the spatial step. We keep the time-step low so we can neglect the discretization error with respect to the time-step.

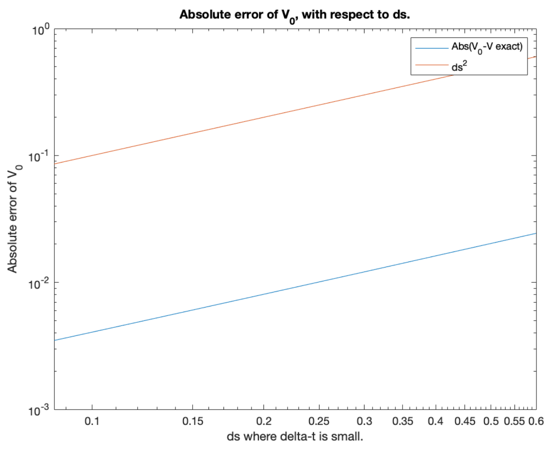


Figure 6: Error with respect to the spatial step

3.3 Comparison between Monte-Carlo methods and Finite difference methods

3.3.1 Accuracy and time complexity

One important property of numerical algorithms is the accuracy and the time complexity. We want to compare the Monte-Carlo method and the finite difference method with respect to these properties.

3.3.1.1 Monte-Carlo method

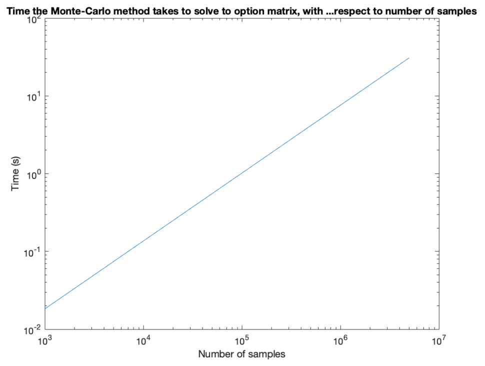
We start by analyzing the Monte-Carlo method. We calculate the time the Monte-Carlo-algorithm takes to calculate the estimated option value, and calculate the error using section (3.1.1). We start by analyzing the error and the time complexity with respect to the number of samples paths.

```
S0 = 14;
K = 15;
r = 0.1;
sigma = 0.25;
T = 0.5;
gamma = 1;

N_samples_vec = [1000, 10000, 100000, 1000000, 5000000];
V_est_vec1 = zeros(1,length(N_samples_vec));

time = zeros(length(N_samples_vec),1);

n_timeponts = 256;
```

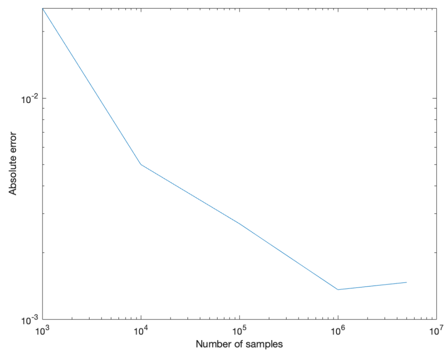
```

for k = 1:length(N_samples_vec)
    tic
    N_samples = N_samples_vec(k);
    V_vec1 = STD_solverv1(N_samples, n_timeponts,T, S0, sigma, gamma, K, r);
    V_est_vec1(k) = exp(-r * T) * mean(V_vec1);
    time(k) = toc;
end


p = polyfit(log(N_samples_vec), log(time), 1);
pol_val = polyval(p, log(N_samples_vec));

figure()
loglog(N_samples_vec, exp(pol_val));
xlabel("Number of samples")
ylabel("Time (s)")
title("Time the Monte-Carlo method takes to solve to option matrix, with ..." + ...
    "respect to number of samples")
```

which gives us:

Accuracy and time complexity of the Monte-Carlo method



(a) Time as a function of the number of samples



(b) Error as a function of number of samples

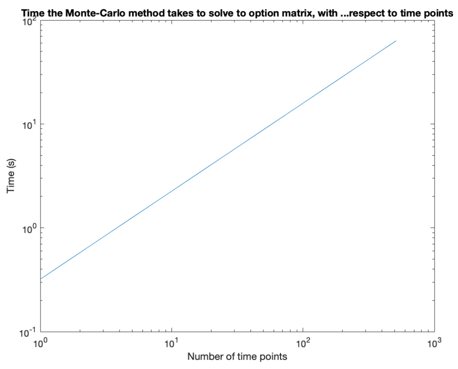
Now we analyze the error and the time complexity with respect to the time-step.

```
S0 = 14;
K = 15;
r = 0.1;
sigma = 0.25;
```

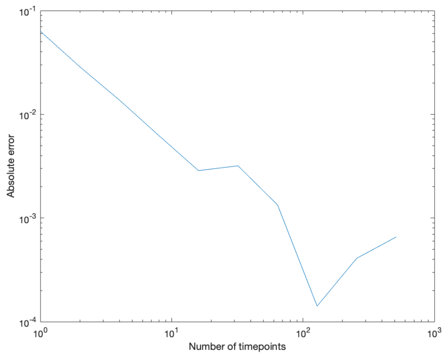
```
T = 0.5;
gamma = 1;

N_samples = 5000000;
n_points_vec = 2 .^ [0, 1, 2, 3, 4, 5, 6, 7, 8,9];
V_est_vec1 = zeros(1,length(n_points_vec));

time = zeros(length(n_points_vec),1);

for k = 1:length(n_points_vec)
    tic
    n_timeponts = n_points_vec(k);
    V_vec1 = STD_solverv1(N_samples, n_timeponts, T, S0, sigma, gamma, K, r);
    V_est_vec1(k) = exp(-r * T) * mean(V_vec1);
    time(k) = toc;
end


p = polyfit(log(n_points_vec), log(time), 1);
pol_val = polyval(p, log(n_points_vec));

figure()
loglog(n_points_vec, exp(pol_val));
xlabel("Number of time points")
ylabel("Time (s)")
title("Time the Monte-Carlo method takes to solve to option matrix, with ..." + ...
    "respect to time points")
```

which gives us:

Accuracy and time complexity of the Monte-Carlo method



(a) Time as a function of the number of time points



(b) Error as a function of number of time points

3.3.1.2 Finite difference methods

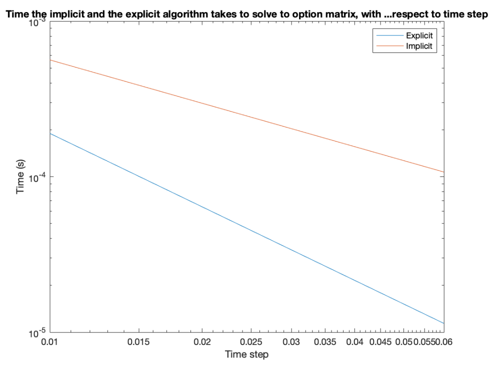
Now we analyze the finite difference method. We calculate the time the explicit and the implicit method takes to calculate the estimated option value, and calculate the error using section (3.2.1). We start by analyzing the error and the time complexity with respect to the time-step:

```
S0 = 14;       % Start value
K = 15;        % Strike price
r = 0.1;       % Risk free interest rate
sigma = 0.25;  % Volatility
T = 0.5;       % Maturity

dt_vector_complexity = 0.01:0.00005:0.06;
time_ex = zeros(length(dt_vector_complexity), 1);
time_im = zeros(length(dt_vector_complexity), 1);

dx = 1;
gamma = 1;

for i = 1:length(dt_vector_complexity)
    tic
    [V_finite_ex_tic, spatial_points_ex_tic,
   time_points_ex_tic] = ...
        CEV_Solver_Explicit(S0, K, T, dt_vector_complexity(i)
   , dx, sigma, r, gamma);

    time_ex(i) = toc;
end

for k = 1:length(dt_vector_complexity)
    tic
    [V_finite_ex_tic, spatial_points_ex_tic,
   time_points_ex_tic] =...
        CEV_Solver_Implicit(S0, K, T, dt_vector_complexity(k)
   , dx, sigma, r, gamma);
    time_im(k) = toc;
end



p_ex = polyfit(log(dt_vector_complexity), log(time_ex), 1);
pol_val_ex = polyval(p_ex, log(dt_vector_complexity));

p_im = polyfit(log(dt_vector_complexity), log(time_im), 1);
pol_val_im = polyval(p_im, log(dt_vector_complexity));


figure(9)
```

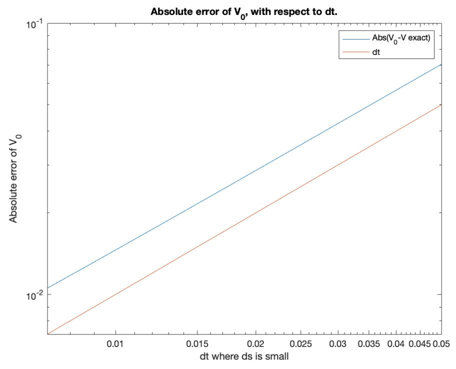
```
loglog(dt_vector_complexity, exp(pol_val_ex), dt_vector_complexity, exp(pol_val_im));
xlabel("Time step")
ylabel("Time (s)")
title("Time the implicit and the explicit algorithm takes to solve to option matrix, with ..." + ...
    "respect to time step")
legend("Explicit", "Implicit")
```

which gives us:

Accuracy and time complexity of the finite difference methods



(a) Time as a function of the time-step



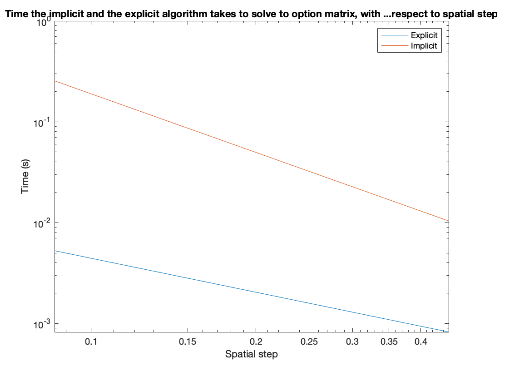
(b) Error as a function of time-step

Now we analyze the error and the time complexity with respect to the spatial-step:

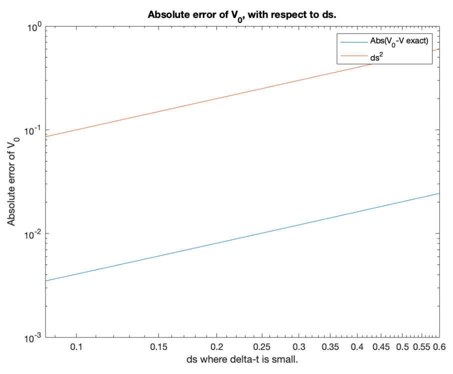
```
S0 = 14;       % Start value
K = 15;        % Strike price
r = 0.1;       % Risk free interest rate
sigma = 0.25;  % Volatility
T = 0.5;       % Maturity

dx_vector_complexity = 0.0857:0.001:0.45;
time_ex_x = zeros(length(dx_vector_complexity), 1);
time_im_x = zeros(length(dx_vector_complexity), 1);

dt = 0.001;
gamma = 1;
for i = 1:length(dx_vector_complexity)
    tic
    [V_finite_ex_tic_2, spatial_points_ex_tic_2, time_points_ex_tic_2] =...
        CEV_Solver_Explicit(S0, K, T, dt, dx_vector_complexity(i), sigma, r, gamma);

    time_ex_x(i) = toc;

end

for k = 1:length(dx_vector_complexity)
    tic
    [V_finite_ex_tic_2, spatial_points_ex_tic_2,
   time_points_ex_tic_2] =...
        CEV_Solver_Implicit(S0, K, T, dt,
   dx_vector_complexity(k), sigma, r, gamma);
    time_im_x(k) = toc;
end


p_ex = polyfit(log(dx_vector_complexity), log(time_ex_x), 1);
pol_val_ex = polyval(p_ex, log(dx_vector_complexity));

p_im = polyfit(log(dx_vector_complexity), log(time_im_x), 1);
pol_val_im = polyval(p_im, log(dx_vector_complexity));

figure()
loglog(dx_vector_complexity, exp(pol_val_ex),
   dx_vector_complexity, exp(pol_val_im))
xlabel("Spatial step")
ylabel("Time (s)")
title("Time the implicit and the explicit algorithm takes to
   solve to option matrix, with ..." + ...
    "respect to spatial step")
legend("Explicit", "Implicit")
```

which gives us:

Accuracy and time complexity of the finite difference methods



(a) Time as a function of the spatial-step



(b) Error as a function of spatial-step

3.3.2 Delta

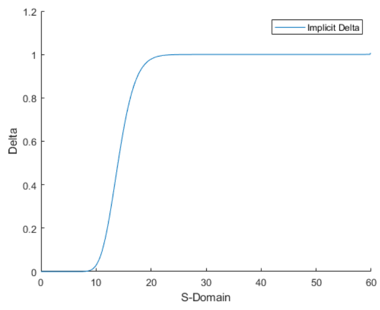
Delta was computed for V(S,t) with arbitrarily chosen S, and t = 0. A comparison of the numeric calculations of delta for Monte Carlo and Finite differences was made by measuring the time both methods took for an average error of similar size, when compared to the analytical solution. The results of this is shown in table (1)

	MC	FD
Time	78,5 s	2,32 s
Mean Error	$13,8 * 10^{-4}$	$10,9 * 10^{-4}$

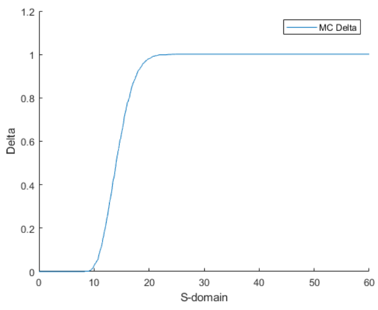
Table 1: Table of mean error and time computing Delta using Monte-Carlo and Finite Differences

Figure (6) show estimated Delta using Finite Differences (a) and Monte-Carlo (b). Figure (7) is a plot of the theoretical Delta.

Figure 6: Estimate of Delta



(a) Estimated Delta of Implicit Finite Difference



(b) Estimated Delta of Monte-Carlo

Theoretical Delta

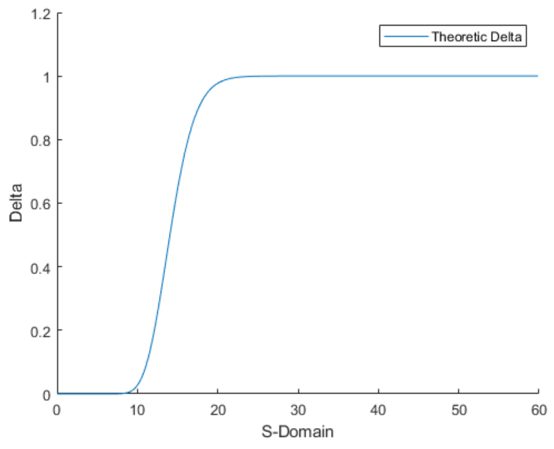


Figure 12: Theoretical Delta

4 Discussion of results

4.1 Several underlying assets

The two types of options that are dependent on several underlying assets that will be considered in this part of the discussion will be limited to rainbow options and baskets.

When moving from one underlying asset to two or more underlying assets variations of numerical methods are implemented to adjust for the change. One of the changes that are possible include adding weights. Because rainbow options compare the value of individual assets the payoff functions consists of two or more asset prices. An example of implementing weights is $(c_iS_2 - c_2S_1)^+$ where c_i represents the weighting variable.

Moving on, PDE methods require relevant PDEs and boundary conditions. This is fairly straight forward when assessing an option with a single underlying assets. However as the number of underlying assets, n, increases the computational demand and memory tend to make these methods struggle, this is known as the curse of dimensions. As for finite difference-discretization, as the number of n increases, an additional dimension is added to the matrix to accommodate for the additional underlying assets.

On a side note, the curse of dimensions also applies to tree methods such as the binomial method as they are also relatively computationally expensive.

Furthermore the Monte-Carlo method does not suffer from the curse of dimensions which makes it an attractive alternative to other numerical methods as n increases. Moreover Monte-Carlo does not require boundary conditions which further alleviates complexity and computational demand as the number of underlying assets increase. As a rule of thumb, the MC method is generally considered competitive when looking at two to three underlying assets and preferable when $n > 3$. Also, the MC error is barely affected by an increase in underlying assets.

Finally, as touched on before, implementation of a higher dimensional solver using the finite difference method quickly becomes disadvantageous due to the heavy computational load. Different BCs may also increase implementation time. It is also worth mentioning that running the implicit finite difference method will more computationally expensive than the explicit one due to the use of memory. Implementing Monte Carlo will be a better alternative since it is both easier and computationally less demanding.

4.2 Implementation

We all concluded that Monte-Carlo was easier to implement than Finite Differences, due to how straight-forward the algorithm is. Finite Differences required some consideration in order to find and implement suitable initial and boundary conditions. We found the Explicit solution to the PDE easier to implement than the Implicit.

4.3 Accuracy and time complexity

From section (3.3.1.1) and (3.3.1.2) we can observe some obvious differences between the Monte-Carlo method and the finite difference methods with respect to accuracy and time complexity. The table below illustrates and approximate the time it takes for the methods to reach a certain error tolerance:

Accuracy and time complexity Monte-Carlo method		
Error type	**log(Absolute Error)**	**log(Time (s))**
Sample paths	10^{-2}	≈ 0.08
Discretization	10^{-2}	≈ 1.05

Table 2: Table of time complexity and error of Monte-Carlo method

Accuracy and time complexity Finite difference method		
Error type	**log(Absolute Error)**	**log(Time (s))**
Time-step	$1.1 \cdot 10^{-2}$	$\approx 1.1 \cdot 10^{-4}$
Spatial-step	10^{-2}	$\approx 1.2 \cdot 10^{-2}$

Table 3: Table of time complexity and error of implicit finite difference method

Note the finite difference method used for (3) is implicit. We only use the implicit method since it does not have the same problems with stability as the explicit method does.

This method is not an ideal experiment to compare the Monte-Carlo method with the finite difference methods. This is due to the fact that the Monte-Carlo method and the finite difference method is quite different, and have different types of errors which make it difficult to compare in a fair manner.

The results illustrated in the table should therefor be taken with a grain of salt. We can however make some general conclusions about the two different methods. We can see that overall, the time it takes for the finite difference method to estimate the option value is much faster compared to the Monte-Carlo method when the error is similar. We can therefor conclude that the error for the finite difference methods converge substantially quicker compared to the Monte-Carlo method.

4.4 Computation of Delta

When computing Delta the Finite Difference methods are more efficient than Monte Carlo. In order to calculate Delta for any V(S,0) the numerical calculation require 2 bordering points V(S+ds,0) and V(S+ds,0). Finite Differences already have those calculated, and it's simple to implement the calculation required to find Delta for V(S,0). Monte-Carlo, however, does not have those bordering points necessary already estimated. Thus V(S+ds,0) and V(S+ds,0) need to be estimated before Delta of V(S,0) can be calculated. If both V(S,0) and Delta of V(S,0) are of interest, then Monte Carlo will be considerably less efficient than Finite Differences.

The graphs in section 3.3.2 Delta show that both numerical methods work well for obtaining Delta. Both are very similar to the theoretical Delta.

Appendices

A MATLAB Functions

A.1 Black Scholes Theoretical Estimate

```matlab
function sol = bsexact(sigma, r, E, T, s)
    % sigma: Volatility
    % r:     Drift
    % E:     Strike price
    % T:     Maturity
    % s:     Spot price of the underlying asset

    d1 = ( log(s/E) + (r + 0.5*sigma^2)*T )/(sigma*sqrt(T));
    d2 = d1 - sigma*sqrt(T);
    F = 0.5*s*(1+erf(d1/sqrt(2))) - exp(-r*T)*E*0.5*(1+erf(d2
   /sqrt(2)))';
    sol = F;

end
```

A.2 Black Scholes Two Dimensional Grid

```matlab
function [exact, spatial_points, time_points] = black_scholes
   (S0, K, r, sigma, dt, dx, T)

    SMAX=4*K;
    time_points = 0:dt:T; %time
    spatial_points = 0:dx:SMAX; %price of the underlying

    exact = zeros(length(spatial_points), length(time_points)
   );

    % Calculate the black scholes value on the entire grid
    for n = 1:length(time_points) % All the time points
        for i = 1:length(spatial_points) % All the spot
   prices
            exact(i, n) = bsexact(sigma, r, K, T-time_points(
   n), spatial_points(i));
        end
    end
end
```

A.3 Monte Carlo Simulation

```matlab
function V_vec = STD_solverv1(N_samples, n_timepoints, T, S0,
    sigma, gamma, K, r)

    V_vec = zeros(1,N_samples);
    dt = T / n_timepoints;

    for i = 1:N_samples
        S_prev = S0;
        for j = 1:n_timepoints
            Z = randn;
            % Euler forward scheme
            S = S_prev + r * S_prev * dt + sigma * S_prev ^
   gamma * Z * sqrt(dt);
            S_prev = S;
        end
        V_vec(i) = max([S - K, 0]);

    end
end
```

A.4 Explicit Finite Difference Method

```
function [V, spatial_points, time_points] =
   CEV_Solver_Explicit(S0, K, T, dt, dx, sigma, r, gamma)
    % S0    - Start price
    % K     - Strike price
    % T     - Maturity
    % SMAX  - Maximal S value
    % dt    - Time step
    % dx    - Spatial step
    % sigma - Volatility
    % r     - Risk free interest rate
    % gamma - Controls the relationship between volatility
   and price

    SMIN = 0;
    SMAX = 4*K;

    % Discretisize the spot price, and the time
    time_points = 0:dt:T;
    spatial_points = SMIN:dx:SMAX;

    V = zeros(length(spatial_points), length(time_points)); %
    Solution matrix

    % We start by settings some boundary conditions

    % Final time point
    for i = 1:length(spatial_points)
        % Pay off function
        V(i, length(time_points)) = max(spatial_points(i)-K
   ,0);
    end
    % Start time point
    for k = 1:length(spatial_points)
        V(i, 1) = 0;
    end


    for n = length(time_points):-1:2 % Time levels
        % Set boundary conditions
        V(1, n-1) = 0;
        V(length(spatial_points), n-1) = SMAX -K*exp(-r*(T-time_points(n-1)));
        for j = 2:length(spatial_points)-1 % Spatial
            % Finite difference scheme
            V(j, n-1) = V(j, n) + (sigma^(2) * 0.5 * dt * spatial_points(j)^(2*gamma))/(dx^(2)) * ...
                (V(j+1, n) - 2*V(j, n) + V(j-1, n)) + ...
                (dt * r * spatial_points(j) * (V(j+1, n) - V(j-1, n)) * 0.5)/(dx) - ...
                r * dt * V(j,n);
        end
    end
end
```

A.5 Implicit Finite Difference Method

```
function [V, spatial_points, time_points] = CEV_Solver_Implicit(S0, K, T, dt, dx, sigma, r, gamma)
    % S0    - Start price
    % K     - Strike price
    % T     - Maturity
    % SMAX  - Maximal S value
    % dt    - Time step
    % dx    - Spatial step
    % sigma - Volatility
    % r     - Risk free interest rate
    % gamma - Controls the relationship between volatility and price

    SMIN = 0;
    SMAX = 4*K;

    % Discretisize the spot price, and the time
    time_points = 0:dt:T;
    spatial_points = SMIN:dx:SMAX;

    V(length(spatial_points), length(time_points)) = nan; % Solution matrix

    % We start by settings some boundary conditions
```

```matlab

    % Final time point
    for i = 1:length(spatial_points)
        % Pay off function
        V(i, length(time_points)) = max(spatial_points(i)-K,0);
    end
    % Start time point
    for k = 1:length(spatial_points)
        V(i, 1) = 0;
    end

    % Final and start spot price
    for n = 1:length(time_points)
        V(1, n) = 0;
        V(length(spatial_points), n) = SMAX - K*exp(-r*(T-time_points(n)));
    end

    % Create matrices
    ax = 0.5 * (r * dt * spatial_points - sigma^(2) * dt * (spatial_points.^(2*gamma)));
    bx = 1 + sigma^(2) * dt * spatial_points.^(2*gamma) + r * dt;
    cx = -0.5 * (r * dt * spatial_points + sigma^(2) * dt * (spatial_points.^(2*gamma)));

    B = diag(ax(3:length(spatial_points)), -1) + diag(bx(2:length(spatial_points))) + ...
    diag(cx(2:length(spatial_points)-1),1);

    [L, U] = lu(B); % lu factorization

    lost = zeros(size(B,2) ,1);

    for i = length(time_points)-1:-1:1 % Propagate backwards in time
        lost(1) = -ax(2) * V(1, i);
        lost(end) = -cx(end) * V(end, i);
        if length(lost) == 1
            lost = -ax(2) * V(1 , i) - cx(end) * V(end , i);
        end
        V(2:length(spatial_points),i) = U \ (L \ (V(2:length(spatial_points),i+1) + lost));
    end
end
```