Final Project Report

Zainab Afzali Kusha

COMP521

December 14, 2023

Abstract:

Numerically estimating high-order derivatives poses challenges, especially when considering finite difference schemes. One study [1] explores an alternative approach employing the inverse of a Vandermonde matrix to compute polynomial derivatives. Building upon the work of the study, we implement a scheme for calculating the closed form of the Vandermonde matrix inverse. Testing this scheme on two functions, $f(x) = e^{x}$ and $f\ (x)\ =\ e^{-5x}\ sin(75x)$, with varying orders of accuracy (1:5), we estimate first and second derivatives over specified step sizes. Additionally, we introduce noise to simulate real-world scenarios and evaluate the method's performance on noisy data. Comparative analysis with a standard finite difference scheme provides insights into accuracy and computational efficiency. The study sheds light on the scheme's efficacy and computational speed, comparing favorably with MATLAB's derivative calculation techniques, especially as the dataset size increases.

Problem introduction:

Numerical differentiation is a fundamental aspect of scientific modeling and numerical analysis, particularly in scenarios where the original generating function of data samples or discrete values is unknown. This necessity arises, for example, when numerically solving ordinary or partial differential equations. The Taylor series-based finite difference approximation is a valuable numerical method, but its complexity increases with higher degrees and orders, limiting its practicality. In the study [1] efforts have been made to tabulate pre-calculated coefficients and find explicit algebraic forms to improve computational efficiency.

The paper introduces a novel approach to numerical approximation of derivatives that overcomes the challenges associated with previous methods. The emphasis is on providing a dependable and efficient method with broader applicability and improved computational characteristics. For instance, the algorithm does not use the Vandermonde matrix (a matrix with the terms of a geometric progression in each row) or its determinant in finding weighting coefficients. Also, the explicit formula imposes less calculation burden, requiring less computing time and storage compared to existing methods. Their method comes with an easy and user-friendly computer code, making it accessible for other researchers to use directly in their calculations. Moreover, the code is recommended for researchers concerned with derivative approximation, especially in solving Ordinary Differential Equations (ODEs) and Partial Differential Equations (PDEs). It is presented as a programming function that, when called with input parameters such as sampling data, degree, and order, returns corresponding derivatives at all grid points. More importantly, the proposed method ensures a unified order of accuracy for derivatives at all grid nodes, addressing concerns related to different accuracies at various points.

Used Functions for Inverse of the Vandermonde:

The MATLAB program in the paper is designed to facilitate the numerical calculation of arbitrary-degree derivatives with arbitrary orders of accuracy. The code includes functions for calculating matrices, determining weighting coefficients, and generating derivatives over the specified domain. It addresses cases where the approximation order needs adjustment for a unified order of accuracy. The functions are explained below:

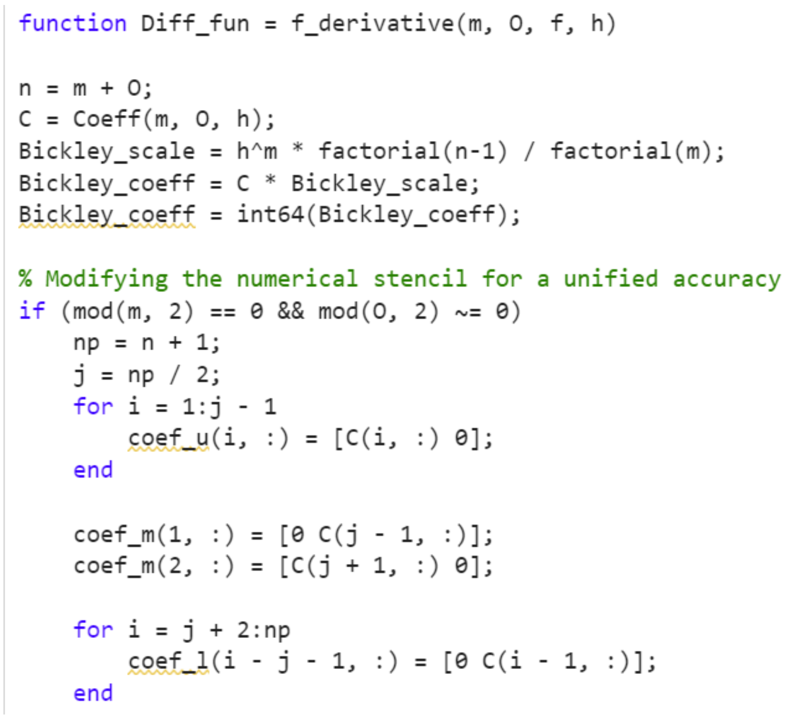
1. f_derivative function:

The function "f_derivative" computes the numerical derivative of a given function "f" using a finite difference scheme with variable accuracy orders ("O") and differentiating orders ("m"). It employs a Bickley-type stencil for numerical differentiation.

The function first calculates the coefficients ("C") based on the desired accuracy order and differentiating order. It then modifies the stencil for even orders to ensure unified accuracy. The size of the stencil ("np") is adjusted accordingly.

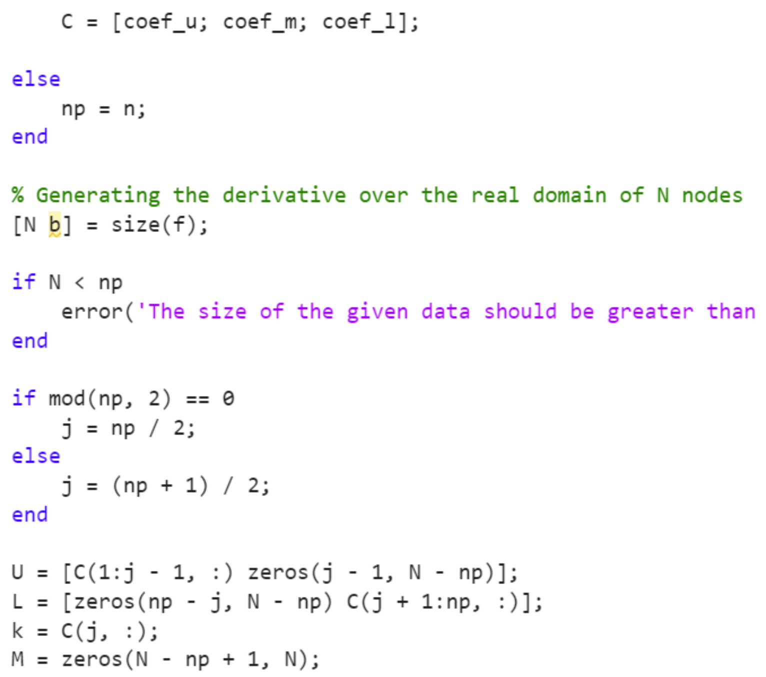
The numerical derivative is generated over the real domain of "N" nodes. The stencil coefficients are combined into matrices ("U", "M", "L") based on their positions in the stencil. The resulting coefficient matrix ("coef") is multiplied with the function values ("f") to obtain the numerical derivative ("Diff_fun").

The function includes error handling to check if the size of the input data is sufficient for the chosen stencil size. The result is the numerical derivative of the input function using the specified accuracy and differentiating orders.

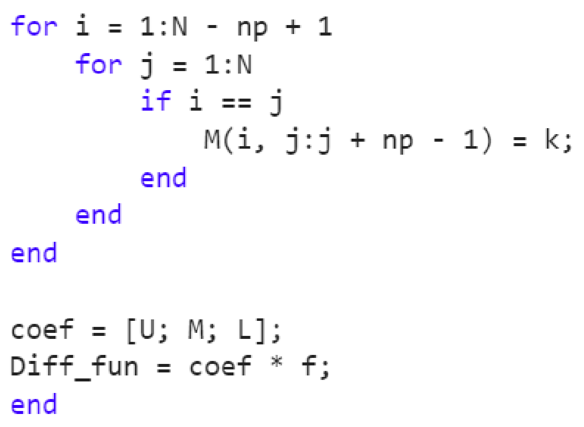


```
function Diff_fun = f_derivative(m, O, f, h)

n = m + O;
C = Coeff(m, O, h);
Bickley_scale = h^m * factorial(n-1) / factorial(m);
Bickley_coeff = C * Bickley_scale;
Bickley_coeff = int64(Bickley_coeff);

% Modifying the numerical stencil for a unified accuracy
if (mod(m, 2) == 0 && mod(O, 2) ~= 0)
    np = n + 1;
    j = np / 2;
    for i = 1:j - 1
        coef_u(i, :) = [C(i, :) 0];
    end

    coef_m(1, :) = [0 C(j - 1, :)];
    coef_m(2, :) = [C(j + 1, :) 0];

    for i = j + 2:np
        coef_l(i - j - 1, :) = [0 C(i - 1, :)];
    end
```

Code Snippet 1: "f_derivative function"



```
    C = [coef_u; coef_m; coef_l];

else
    np = n;
end

% Generating the derivative over the real domain of N nodes
[N b] = size(f);

if N < np
    error('The size of the given data should be greater than
end

if mod(np, 2) == 0
    j = np / 2;
else
    j = (np + 1) / 2;
end

U = [C(1:j - 1, :) zeros(j - 1, N - np)];
L = [zeros(np - j, N - np) C(j + 1:np, :)];
k = C(j, :);
M = zeros(N - np + 1, N);
```

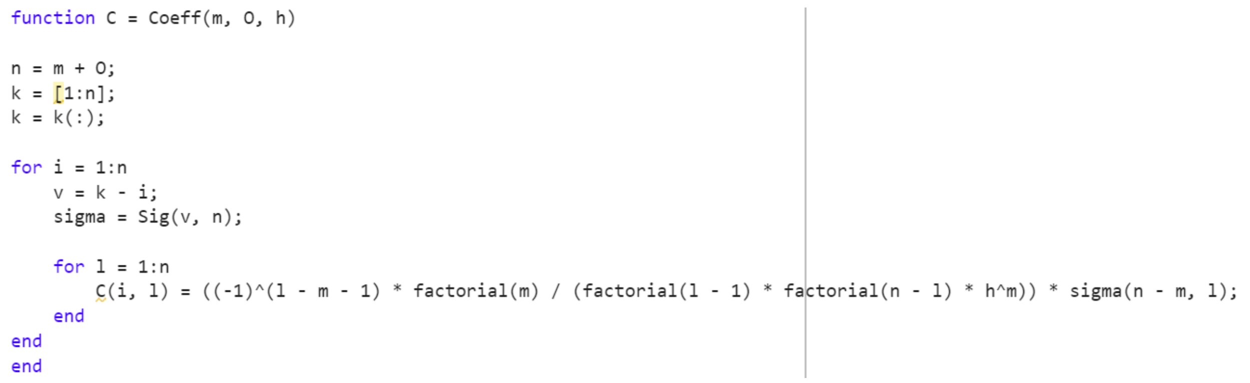


```
for i = 1:N - np + 1
    for j = 1:N
        if i == j
            M(i, j:j + np - 1) = k;
        end
    end
end

coef = [U; M; L];
Diff_fun = coef * f;
end
```

Code Snippet 2: "f_derivative function"

2. Coeff function:

The function "Coeff" calculates the coefficients ("C") for the Bickley-type stencil used in numerical differentiation. The coefficients are determined based on the desired accuracy order ("O") and differentiating order ("m"). The function utilizes a loop to compute each entry of the coefficient matrix.

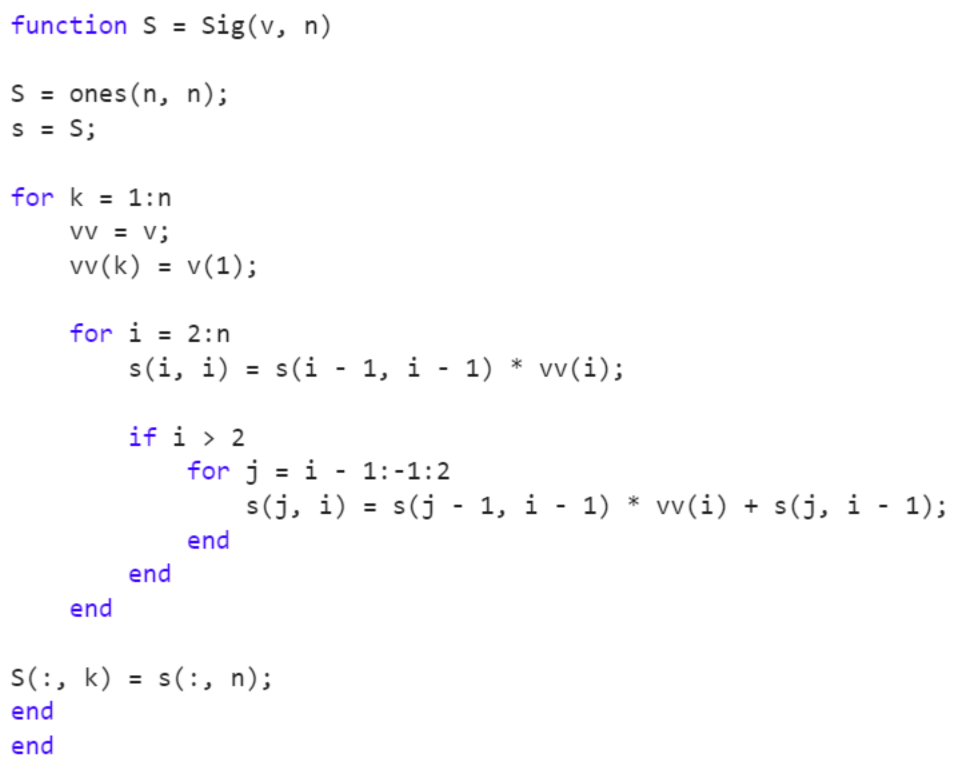


```
function C = Coeff(m, O, h)

n = m + O;
k = [1:n];
k = k(:);

for i = 1:n
    v = k - i;
    sigma = Sig(v, n);

    for l = 1:n
        C(i, l) = ((-1)^(l - m - 1) * factorial(m) / (factorial(l - 1) * factorial(n - l) * h^m)) * sigma(n - m, l);
    end
end
end
```

Code Snippet 3: "Coeff function"

3. Sig function:

The "Sig" function constructs a square matrix "S" using an iterative algorithm. It iterates through each column of "S", modifying its elements based on a derived vector "vv" from the input vector "v". The updates are performed in a specific pattern, and the final matrix "S" reflects this pattern based on the input vector and its length.



```
function S = Sig(v, n)

S = ones(n, n);
s = S;

for k = 1:n
    vv = v;
    vv(k) = v(1);

    for i = 2:n
        s(i, i) = s(i - 1, i - 1) * vv(i);

        if i > 2
            for j = i - 1:-1:2
                s(j, i) = s(j - 1, i - 1) * vv(i) + s(j, i - 1);
            end
        end
    end

S(:, k) = s(:, n);
end
end
```

Code Snippet 4: "Sig function"

Problem 1:

Equation 1:

Using the function

$f(x) = e^x$ (EQ1)

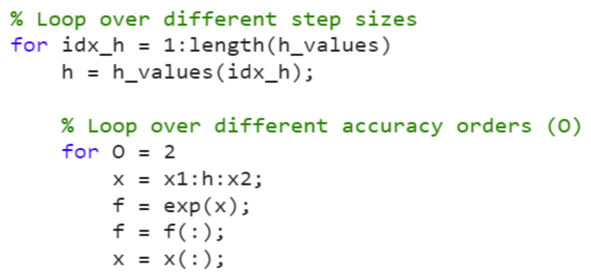
estimate the derivatives using the inverse Vandermonde matrix for step sizes $h \in [0.1, 0.050.0250.0125]$ on the interval $x \in [0, 1]$. Do this for the first and second derivatives. Show and discuss the accuracy of the scheme, and the time needed to solve. Compare this to a standard finite difference scheme of your choice.

Method 1: Inverse of the Vandermonde (Hassan's method)

In this problem code, I conducted a numerical study to analyze the performance of a numerical derivative method. Also, I understand how the error and computation time behave with varying step sizes and accuracy orders. The fitted lines on the log-log plots help analyze the convergence rates of Hassan's numerical method.

1. **Setup:**

I define the function **f(x) = exp(x)** and set up the interval **[0, 1]** with different step sizes (**h_values**).



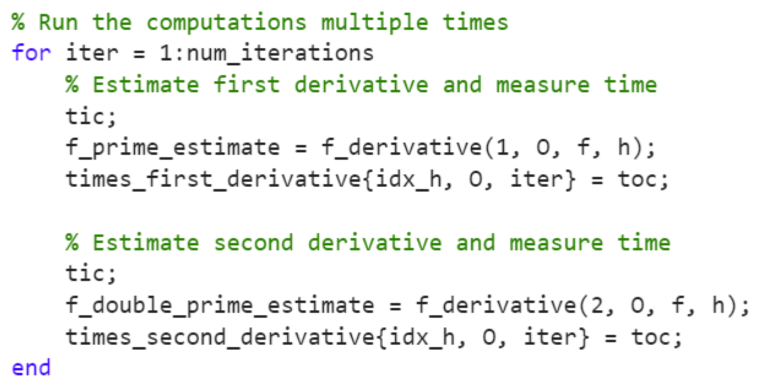
```
% Loop over different step sizes
for idx_h = 1:length(h_values)
    h = h_values(idx_h);

    % Loop over different accuracy orders (O)
    for O = 2
        x = x1:h:x2;
        f = exp(x);
        f = f(:);
        x = x(:);
```

Code Snippet 5: "Setup"

2. **Iterative Computation:**

I iterate over different step sizes and accuracy orders (O) to calculate the numerical first and second derivatives of the function **f(x)** using the **f_derivative** function.

I run the computation multiple times (controlled by **num_iterations**) and measure the time taken for each iteration, and then visualize the mean time that is needed for the method.



```
% Run the computations multiple times
for iter = 1:num_iterations
    % Estimate first derivative and measure time
    tic;
    f_prime_estimate = f_derivative(1, O, f, h);
    times_first_derivative{idx_h, O, iter} = toc;

    % Estimate second derivative and measure time
    tic;
    f_double_prime_estimate = f_derivative(2, O, f, h);
    times_second_derivative{idx_h, O, iter} = toc;
end
```

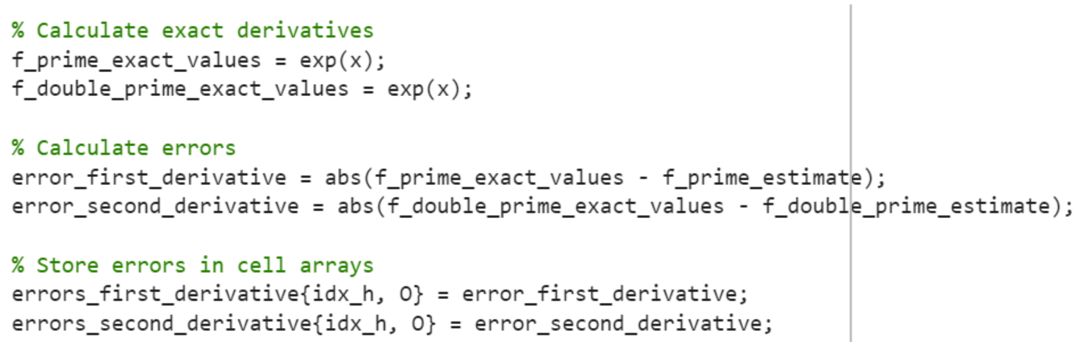
Code Snippet 6: "First and Second derivative approximation"

3. **Comparison with Exact Derivatives:**

I calculate the exact derivatives (**f_prime_exact_values** and **f_double_prime_exact_values**) for comparison, and then compute the errors between the numerical and exact derivatives for each iteration and store them in cell arrays (**errors_first_derivative** and **errors_second_derivative**).

4. **Error Analysis:**

I calculate the max and mean errors and fit lines to the log-log plots to estimate the order of accuracy. The slopes of these lines represent the order of accuracy for each case.

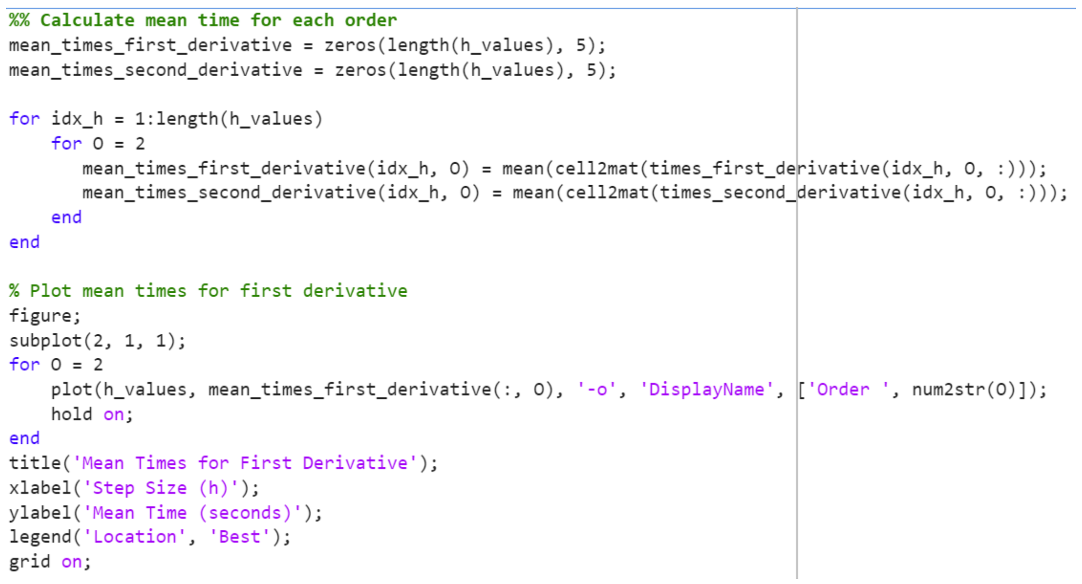


```
% Calculate exact derivatives
f_prime_exact_values = exp(x);
f_double_prime_exact_values = exp(x);

% Calculate errors
error_first_derivative = abs(f_prime_exact_values - f_prime_estimate);
error_second_derivative = abs(f_double_prime_exact_values - f_double_prime_estimate);

% Store errors in cell arrays
errors_first_derivative{idx_h, O} = error_first_derivative;
errors_second_derivative{idx_h, O} = error_second_derivative;
```

Code Snippet 7: "Error Analysis"

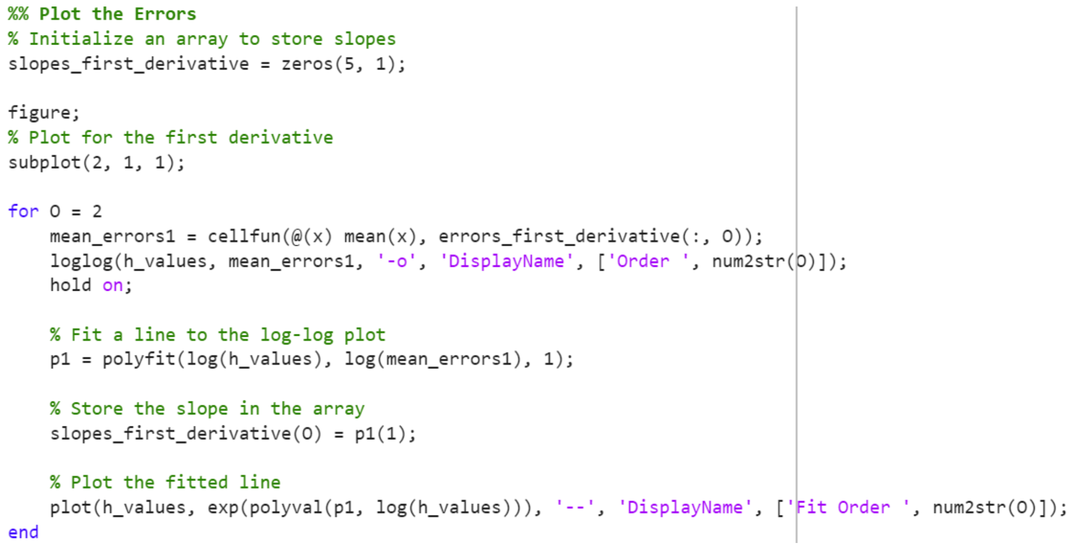
5. **Visualization:**

I visualize the results by plotting mean times and error analyses for both the first and second derivatives.



```
%% Calculate mean time for each order
mean_times_first_derivative = zeros(length(h_values), 5);
mean_times_second_derivative = zeros(length(h_values), 5);

for idx_h = 1:length(h_values)
    for O = 2
       mean_times_first_derivative(idx_h, O) = mean(cell2mat(times_first_derivative(idx_h, O, :)));
       mean_times_second_derivative(idx_h, O) = mean(cell2mat(times_second_derivative(idx_h, O, :)));
    end
end

% Plot mean times for first derivative
figure;
subplot(2, 1, 1);
for O = 2
    plot(h_values, mean_times_first_derivative(:, O), '-o', 'DisplayName', ['Order ', num2str(O)]);
    hold on;
end
title('Mean Times for First Derivative');
xlabel('Step Size (h)');
ylabel('Mean Time (seconds)');
legend('Location', 'Best');
grid on;
```

Code Snippet 8: “Mean time calculation and visualization”

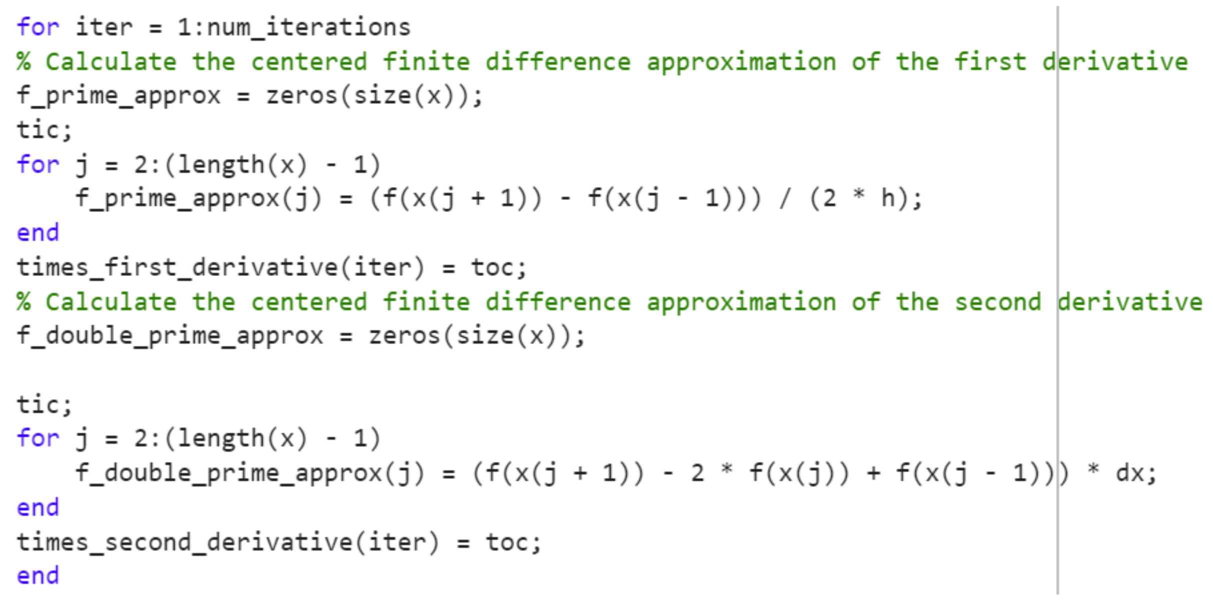


```
%% Plot the Errors
% Initialize an array to store slopes
slopes_first_derivative = zeros(5, 1);

figure;
% Plot for the first derivative
subplot(2, 1, 1);

for O = 2
    mean_errors1 = cellfun(@(x) mean(x), errors_first_derivative(:, O));
    loglog(h_values, mean_errors1, '-o', 'DisplayName', ['Order ', num2str(O)]);
    hold on;

    % Fit a line to the log-log plot
    p1 = polyfit(log(h_values), log(mean_errors1), 1);

    % Store the slope in the array
    slopes_first_derivative(O) = p1(1);

    % Plot the fitted line
    plot(h_values, exp(polyval(p1, log(h_values))), '--', 'DisplayName', ['Fit Order ', num2str(O)]);
end
```

Code Snippet 9: “Calculation order of accuracy and visualization”

Method 2: Centered Finite Difference

This method performs a numerical analysis of the first and second derivatives of the function using centered finite difference approximations. Most of the parts of this code are similar to the last method, only the derivatives calculation is different.



```
for iter = 1:num_iterations
% Calculate the centered finite difference approximation of the first derivative
f_prime_approx = zeros(size(x));
tic;
for j = 2:(length(x) - 1)
    f_prime_approx(j) = (f(x(j + 1)) - f(x(j - 1))) / (2 * h);
end
times_first_derivative(iter) = toc;
% Calculate the centered finite difference approximation of the second derivative
f_double_prime_approx = zeros(size(x));

tic;
for j = 2:(length(x) - 1)
    f_double_prime_approx(j) = (f(x(j + 1)) - 2 * f(x(j)) + f(x(j - 1))) * dx;
end
times_second_derivative(iter) = toc;
end
```

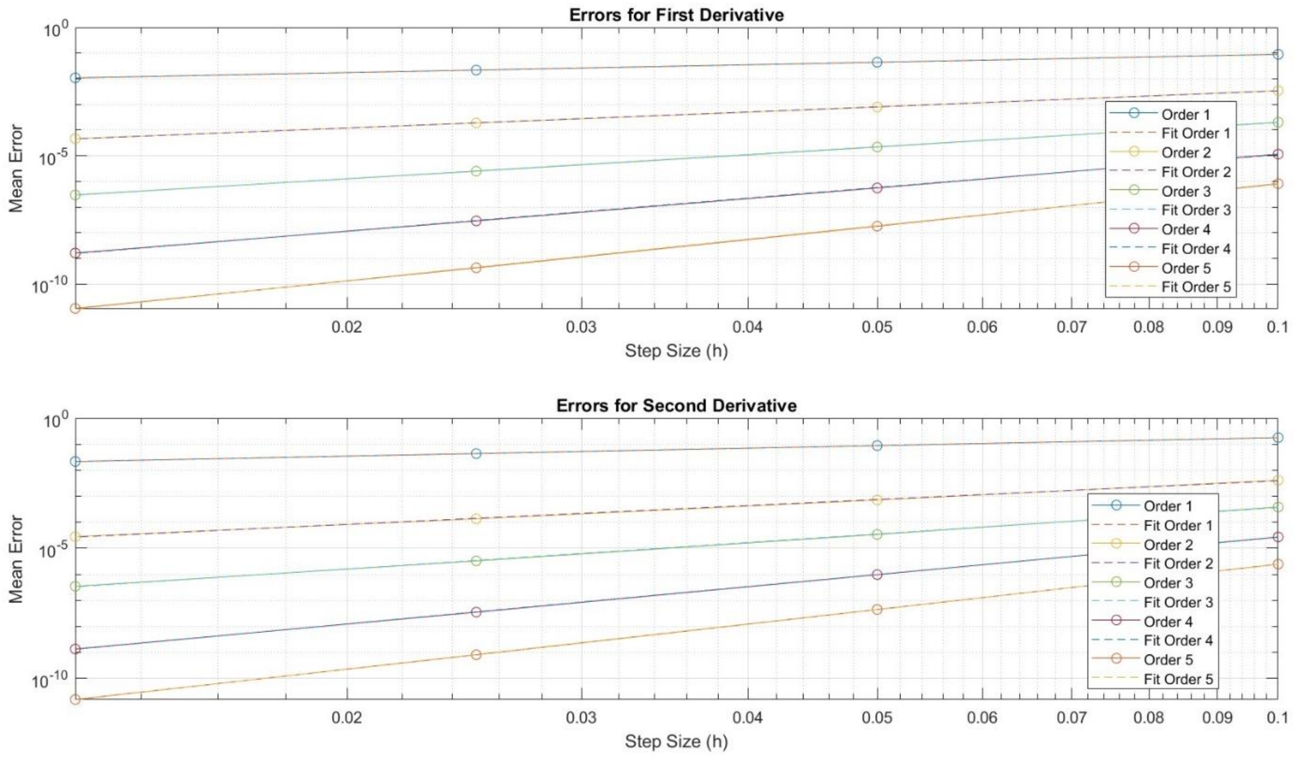
Code Snippet 10: "Derivatives calculation"

Results:

Figures 1 and 2 are for Hassan's method (If the quality of the images is not very high, all of them exist in the plot folder for your reference). I plotted the mean errors for each order of accuracy vs step sizes. As we can see, when the step size decreases, the mean error decreases too which is as we expected. On the other hand, as the error vs step size decreases, we have a higher order of accuracy.

To be more specific, the order of convergence in numerical methods refers to the rate at which the numerical solution approaches the true solution as the step size decreases. It provides information about how quickly the numerical method converges to the exact solution as the computational effort increases.

In the context of numerical differentiation or integration methods, the order of convergence is typically assessed by studying how the error in the numerical result changes with a reduction in step size. The order of convergence is often denoted by the symbol p, and it is determined by the power to which the step size (h) is raised in the error term: $E(h) \approx Ch^p$



"Figure 1. Mean errors for O = 1:5, Hassan's method"

The below table is the calculated order of accuracies from errors vs step size to evaluate Hassan's method: The calculated orders are what we expected them to be, and it shows that the method is working properly.

"Table 1. Comparison between actual and calculated O, First derivative"

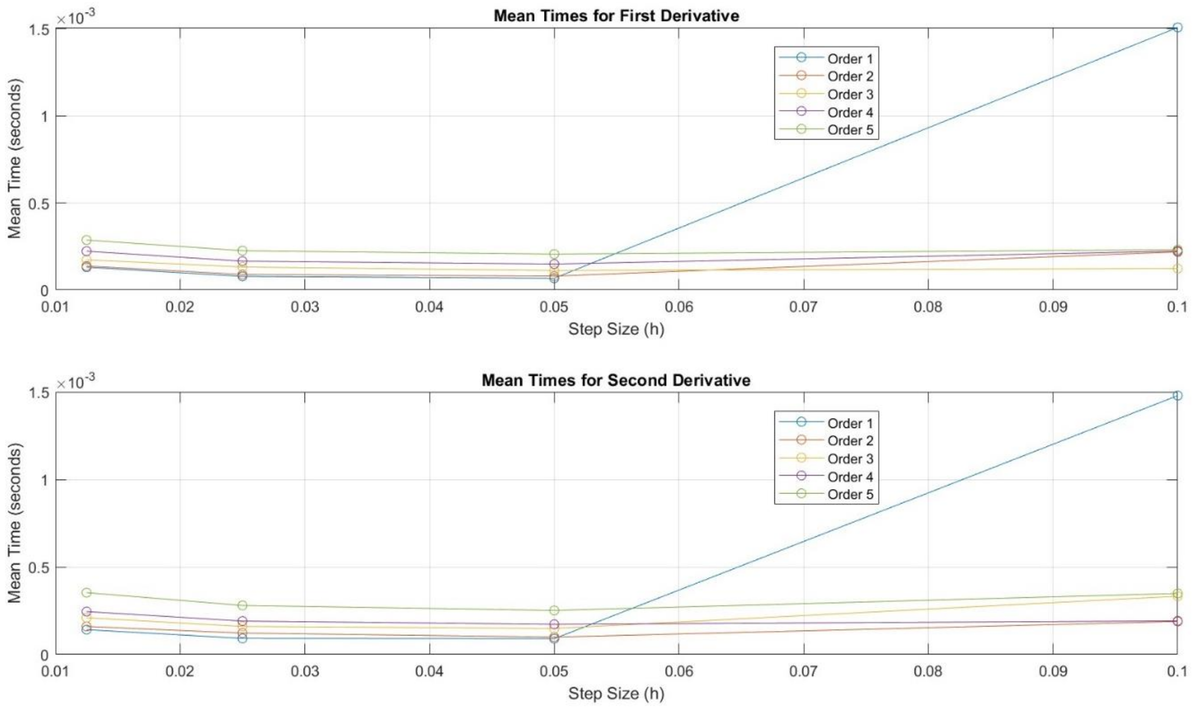
Actual order of accuracy	**1**	**2**	**3**	**4**	**5**
Calculated order of accuracy	**0.9863**	**1.9692**	**2.9507**	**3.9316**	**4.9114**

“Table 2. Comparison between actual and calculated O, Second derivative”

The actual order of accuracy:	**1**	**2**	**3**	**4**	**5**
Calculated order of accuracy:	**0.97616**	**1.9553**	**2.935**	**3.9143**	**4.8176**

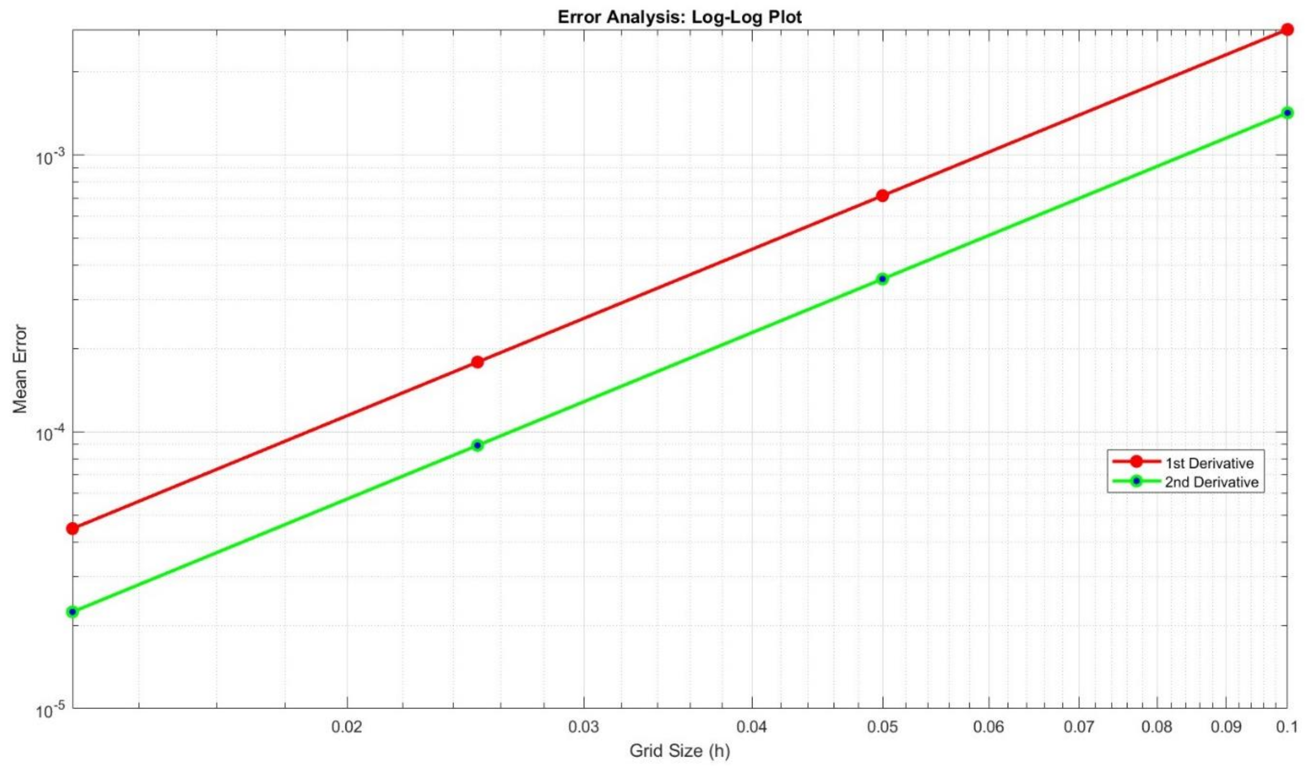
In the provided code, I did the computation for 10 iterations to be able to calculate the mean execution time for the method. The timing for different orders of accuracy does not differ significantly (Figure 2.), which is an amazing point of this method (I am not sure why for O = 1, we have a spike at the end).

This behavior is likely due to the efficient implementation of the finite difference method used to compute the derivatives. The code uses coefficients and stencils that are precomputed and reused for different accuracy orders. This avoids redundant calculations during each iteration, contributing to computational efficiency. Also, the code utilizes vectorized operations and matrix manipulations in functions like f_derivative and Coeff. Vectorization in MATLAB allows for parallelized computations, leading to faster execution compared to explicit loops. (I used MATLAB’s profile to see how much time is devoted to different parts of the code, and it seems the Coeff function used more time than other parts of the code.)

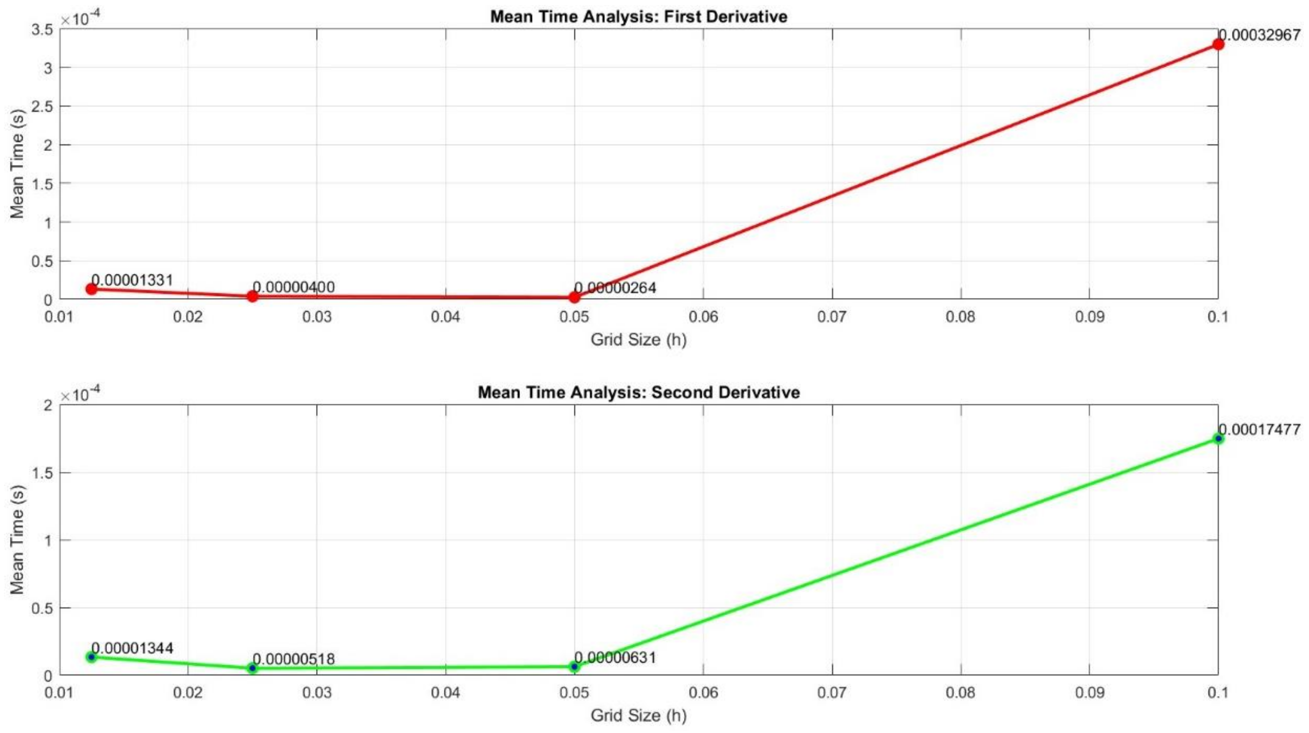


“Figure 2. The mean time for O = 1:5_ 10 iterations, Hassan’s method”

In the next section, the results of centered finite difference are explained. The mean errors vs grid sizes, and mean time for 10 iterations is plotted to better understand how this method performs.



"Figure 3. Mean errors,O = 2,Centered Finite Difference method"



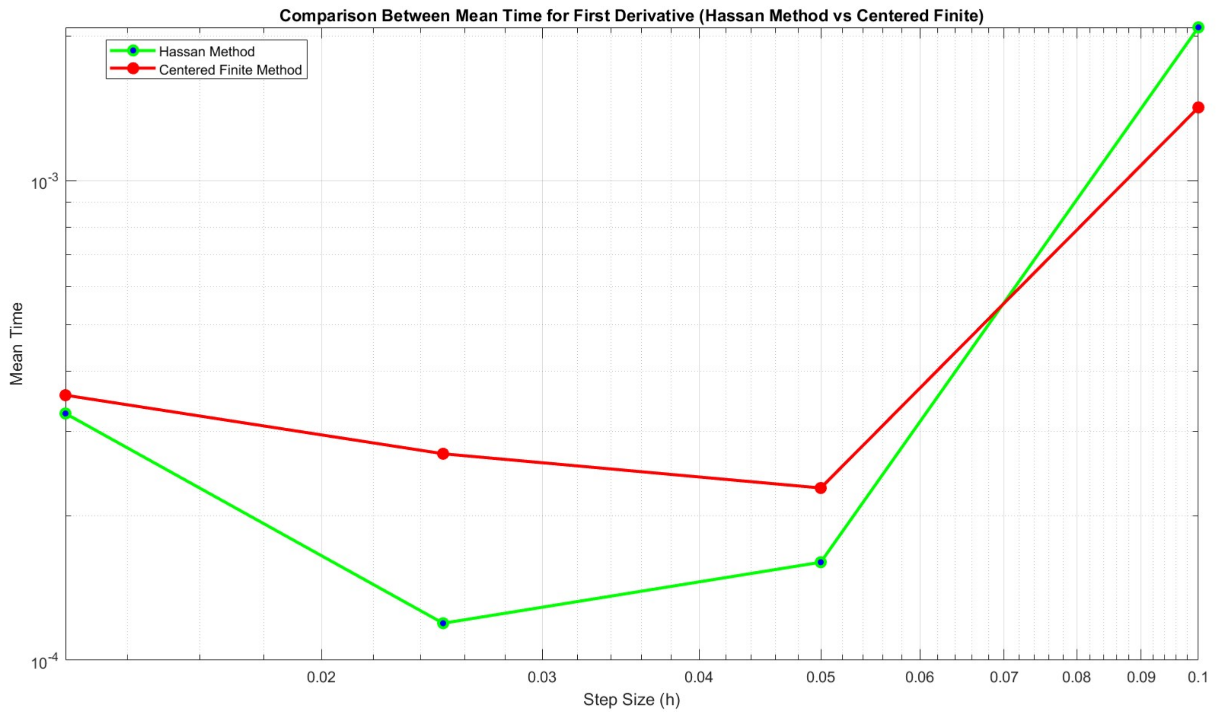
"Figure 4. Mean time,O = 2,Centered Finite Difference method"

The calculated order accuracy of the centered finite difference for both the first and second derivative is 1.9587, so the order of accuracy of this method is almost 2. Therefore, I must compare the errors and time of this method to order of accuracy 2 of Hassan's method. Below, there are tables of this comparison.

"Table 3. Comparison Mean time between Hassan's method and the Centered finite method"

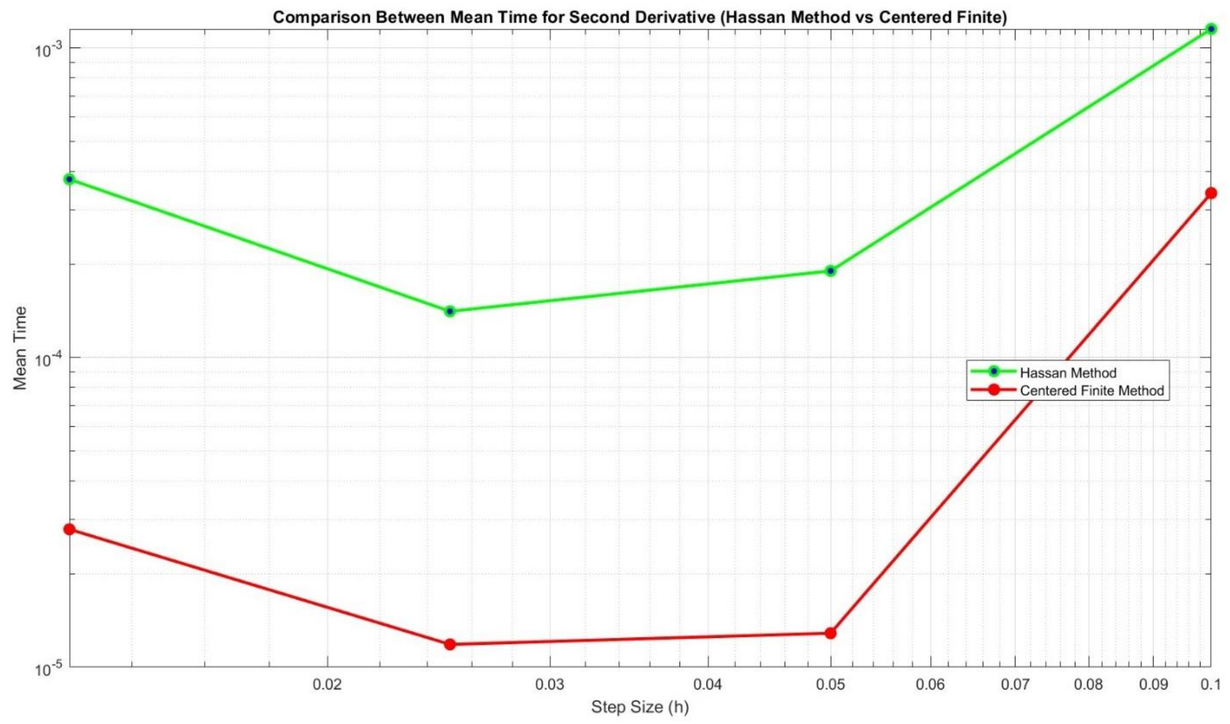
Step sizes (h)	0.1	0.05	0.025	0.0125
Mean Time: First derivative (Hassan's method)	0.00014188	0.0001205	0.00014207	0.00048786
Mean Time: Second derivative (Hassan's method)	0.00014799	0.0001197	0.00019235	0.00024708
Mean Time: First derivative (Center method)	0.0001868	4.55e-06	4.79e-06	1.748e-05
Mean Time: Second derivative (Center method)	0.00014227	1.023e-05	6.87e-06	1.968e-05

From the table and the plot, we can see that in first derivative Hassan's method is faster (as we expected) than the centered finite method except for the bigger step sizes ($h > 0.07$).



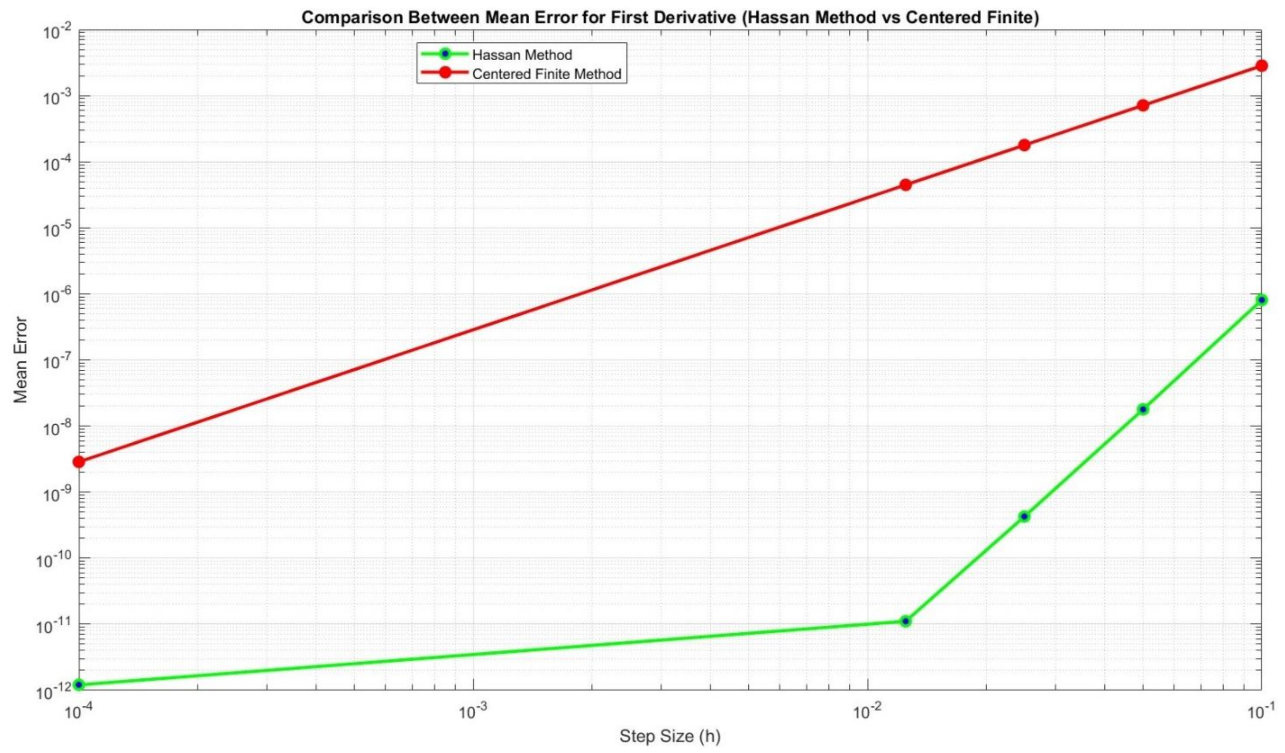
"Figure 5. Comparison Mean time between Hassan's method and the Centered finite method_First Derivative"

Although, in the second derivative, it seems that the center finite method is faster than Hassan's method, but the difference is not significant. I am not quite sure why this happened, Hassan's method should bit the other method. Maybe, it is because in Hassan's main code there are "for" loops that may increase the execution time.



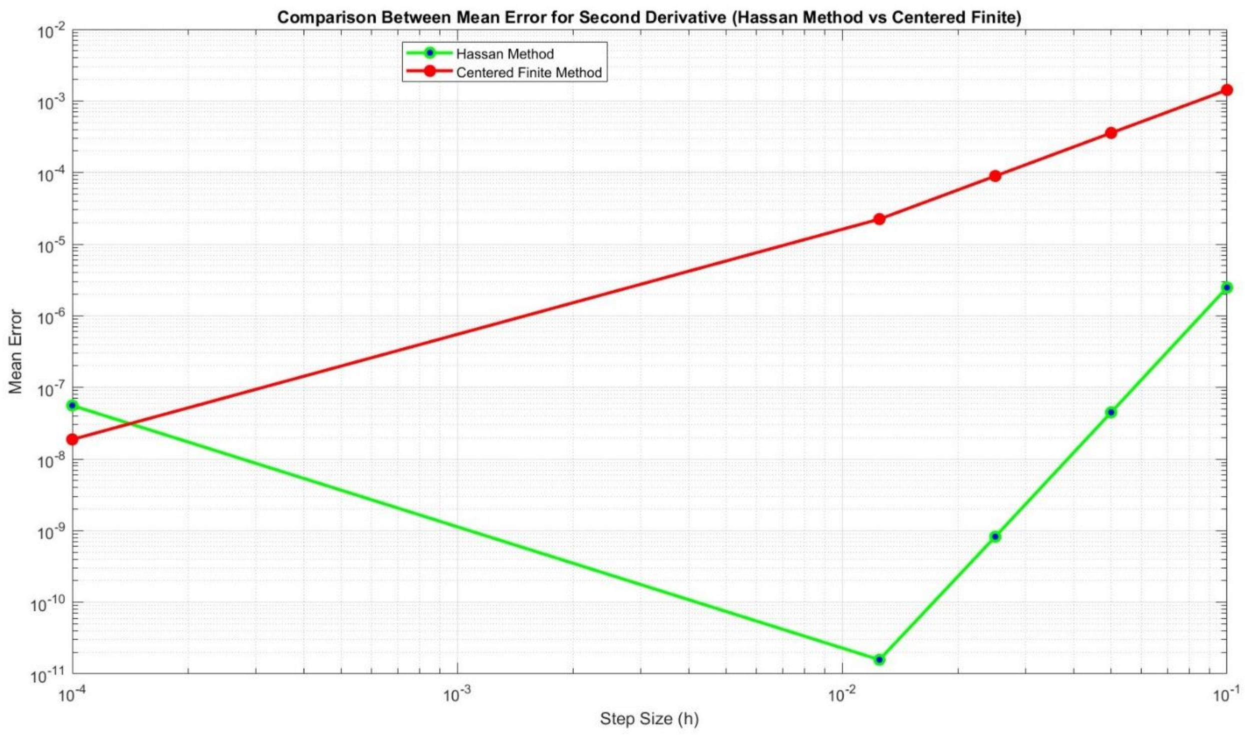
"Figure 6. Comparison Mean time between Hassan's method and the Centered finite method_Second Derivative"

Now, the comparison between the two methods' errors: When I was comparing them $h\_values = [0.1, 0.05, 0.025, 0.0125]$, it seemed that the centered finite method's errors are lower than Hassan's method. Therefore I added a 0.0001 step size, and then Hassan's method's errors were significantly lower than the centered method, and this is what we expected of the novel and efficient method.



"Figure 7. Comparison Mean error between Hassan's method and the Centered finite method_First Derivative"

Based on the results, the mean errors for the first derivative of the two methods are close, and for the second derivative they are slightly different, and the centered finite method has lower errors. This is a little odd since we expect that Hassan's method to performs better. Althogh, when the step sizes are smaller Hassan's method and centered method do not differ much and this is a good sign.



"Figure 8. Comparison Mean error between Hassan's method and the Centered finite method_Second Derivative"

Problem 2:

Equation 2:

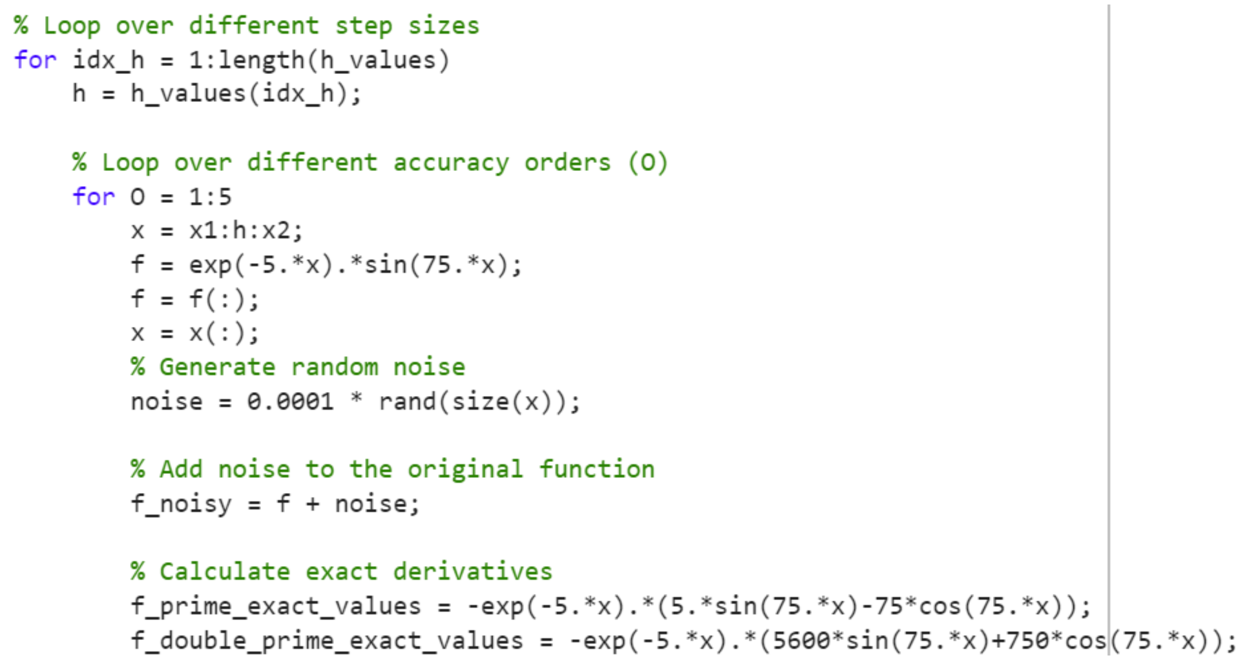
Using the function:

$$f\,(x) \;=\; e^{-5x}\,sin(75x) \qquad \text{(EQ 2)}$$

estimate the first derivative using the inverse Vandermonde matrix with step size $h \;=\; 0.01$ on the domain $x \in [0, 1]$. Many times, the derivative must be estimated from observational data which may be noisy, and this noise can significantly affect the accuracy of a numerical scheme. Add some noise to the initial function data and discuss how well the method works for higher-order derivatives with noisy data. Compare the speed of the solution with MATLAB's technique for finding the derivative of data. How do they compare? How about when there are more data points ($h \;=\; 0.001, 0.0001$).

Method 1: Inverse of the Vandermonde (Hassan's method)

Everything in the code is the same as Equation 1, except the function and its derivatives definitions:



```
% Loop over different step sizes
for idx_h = 1:length(h_values)
    h = h_values(idx_h);

    % Loop over different accuracy orders (O)
    for O = 1:5
        x = x1:h:x2;
        f = exp(-5.*x).*sin(75.*x);
        f = f(:);
        x = x(:);
        % Generate random noise
        noise = 0.0001 * rand(size(x));

        % Add noise to the original function
        f_noisy = f + noise;

        % Calculate exact derivatives
        f_prime_exact_values = -exp(-5.*x).*(5.*sin(75.*x)-75*cos(75.*x));
        f_double_prime_exact_values = -exp(-5.*x).*(5600*sin(75.*x)+750*cos(75.*x));
```

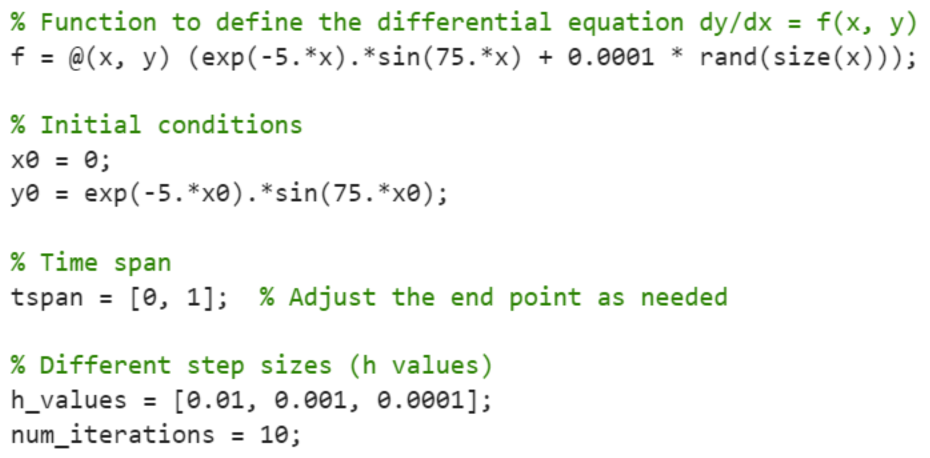
Code Snippet 11: "Setup"

Method 2 and 3: ODE45 and Poly approximation

I tried implementing two methods since I was not very familiar with either of these methods, so I implemented and investigated these two methods to practice and better understand them and be able to compare them with Hassan's method.

ODE45: The MATLAB code for this method performs a numerical analysis of a given differential equation using the ode45 solver. The goal is to study the behavior of the errors and computation times for the first and second derivatives of a function for different step sizes (h values).

1. **Differential equation, initial conditions, step sizes, and Iterations:**



```
% Function to define the differential equation dy/dx = f(x, y)
f = @(x, y) (exp(-5.*x).*sin(75.*x) + 0.0001 * rand(size(x)));

% Initial conditions
x0 = 0;
y0 = exp(-5.*x0).*sin(75.*x0);

% Time span
tspan = [0, 1];  % Adjust the end point as needed

% Different step sizes (h values)
h_values = [0.01, 0.001, 0.0001];
num_iterations = 10;
```

Code Snippet 12: "Setup"

2. **ODE Solver:**

The ODE is solved using the **ode45** solver, and the numerical solution is stored in the variables **x** and **y**.

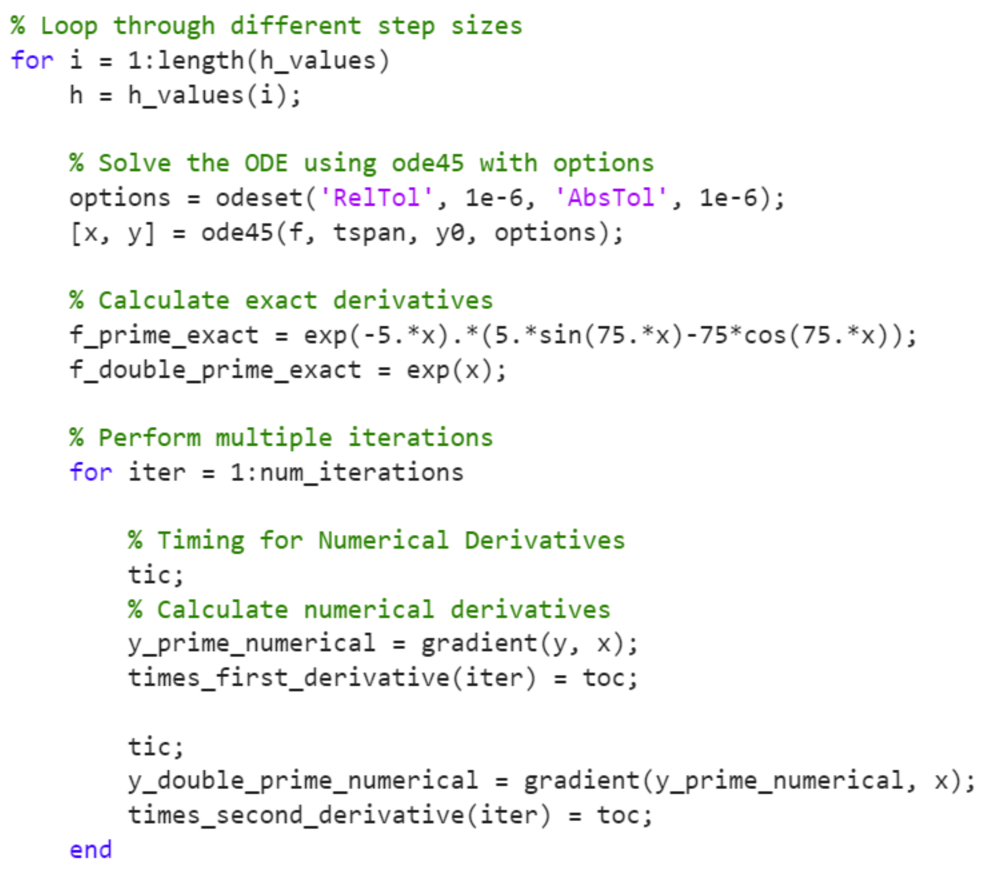
3. **Exact Derivatives:**

Exact derivatives (**f_prime_exact** and **f_double_prime_exact**) are calculated based on the analytical solution.

4. **Numerical Derivatives and Timing:**

Numerical derivatives are calculated using the **gradient** function.

The script measures the computation time for both the first and second derivatives using the **tic** and **toc** functions.



```
% Loop through different step sizes
for i = 1:length(h_values)
    h = h_values(i);

    % Solve the ODE using ode45 with options
    options = odeset('RelTol', 1e-6, 'AbsTol', 1e-6);
    [x, y] = ode45(f, tspan, y0, options);

    % Calculate exact derivatives
    f_prime_exact = exp(-5.*x).*(5.*sin(75.*x)-75*cos(75.*x));
    f_double_prime_exact = exp(x);

    % Perform multiple iterations
    for iter = 1:num_iterations

        % Timing for Numerical Derivatives
        tic;
        % Calculate numerical derivatives
        y_prime_numerical = gradient(y, x);
        times_first_derivative(iter) = toc;

        tic;
        y_double_prime_numerical = gradient(y_prime_numerical, x);
        times_second_derivative(iter) = toc;
    end
```

Code Snippet 13: "ODE solver, and derivatives calculations"

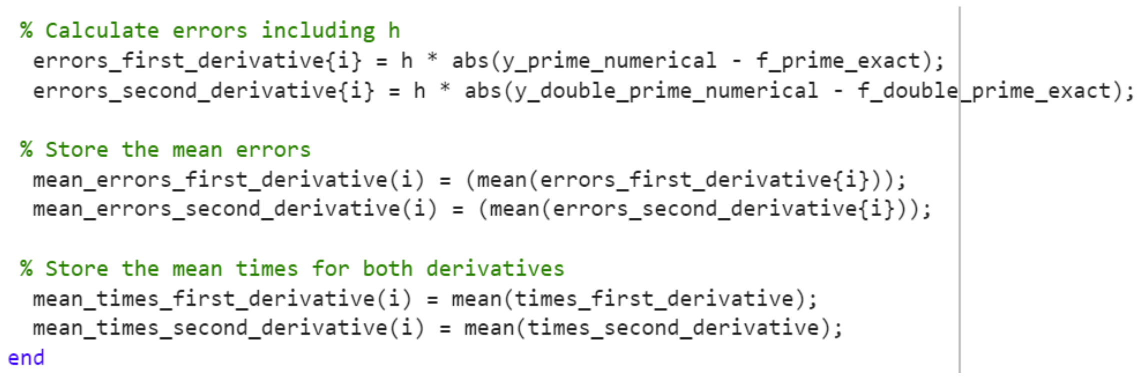
I was not sure how to include different step sizes in the method, so I tried to include them just in error calculations.

5. **Error Calculation:**

Errors are calculated by comparing the numerical derivatives with the exact derivatives, and they are stored in cell arrays (**errors_first_derivative** and **errors_second_derivative**).

6. **Mean Errors, Times, and visualization:**

Mean errors and mean computation times are calculated based on the repeated iterations for each step size.



```
  % Calculate errors including h
   errors_first_derivative{i} = h * abs(y_prime_numerical - f_prime_exact);
   errors_second_derivative{i} = h * abs(y_double_prime_numerical - f_double_prime_exact);

  % Store the mean errors
   mean_errors_first_derivative(i) = (mean(errors_first_derivative{i}));
   mean_errors_second_derivative(i) = (mean(errors_second_derivative{i}));

  % Store the mean times for both derivatives
   mean_times_first_derivative(i) = mean(times_first_derivative);
   mean_times_second_derivative(i) = mean(times_second_derivative);
end
```

Code Snippet 14: "Mean error and Mean time calculation"

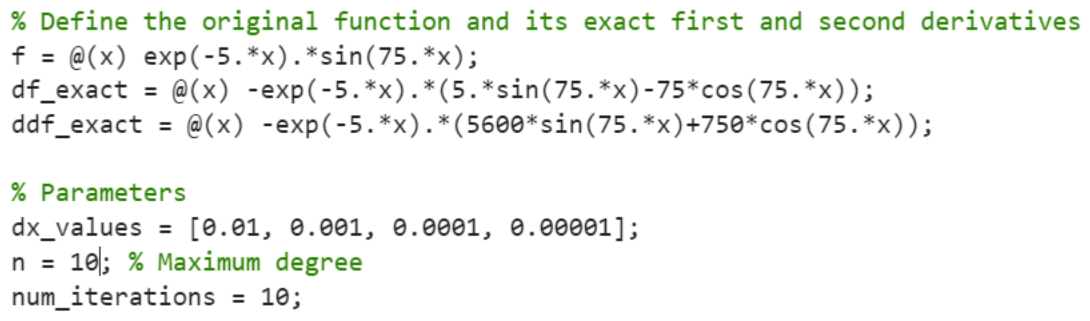
Poly: The **polyfit** and **polyder** functions are used in this method. **polyfit** fits a polynomial to the given data, and **polyder** calculates the derivative of a polynomial. These functions are commonly used in polynomial regression, where you fit a polynomial to a set of data points. In this case, it's being applied to approximate derivatives of a function at given points.

1. **Function Definition:**

The code defines a function **f(x)** and its exact first and second derivatives **df_exact(x)** and **ddf_exact(x)**.

2. **Step Sizes and Parameters:**

Different step sizes (**dx_values**), the maximum degree for polynomial fit (**n**), and the number of iterations for mean time calculation (**num_iterations**) are specified.



```
% Define the original function and its exact first and second derivatives
f = @(x) exp(-5.*x).*sin(75.*x);
df_exact = @(x) -exp(-5.*x).*(5.*sin(75.*x)-75*cos(75.*x));
ddf_exact = @(x) -exp(-5.*x).*(5600*sin(75.*x)+750*cos(75.*x));

% Parameters
dx_values = [0.01, 0.001, 0.0001, 0.00001];
n = 10; % Maximum degree
num_iterations = 10;
```

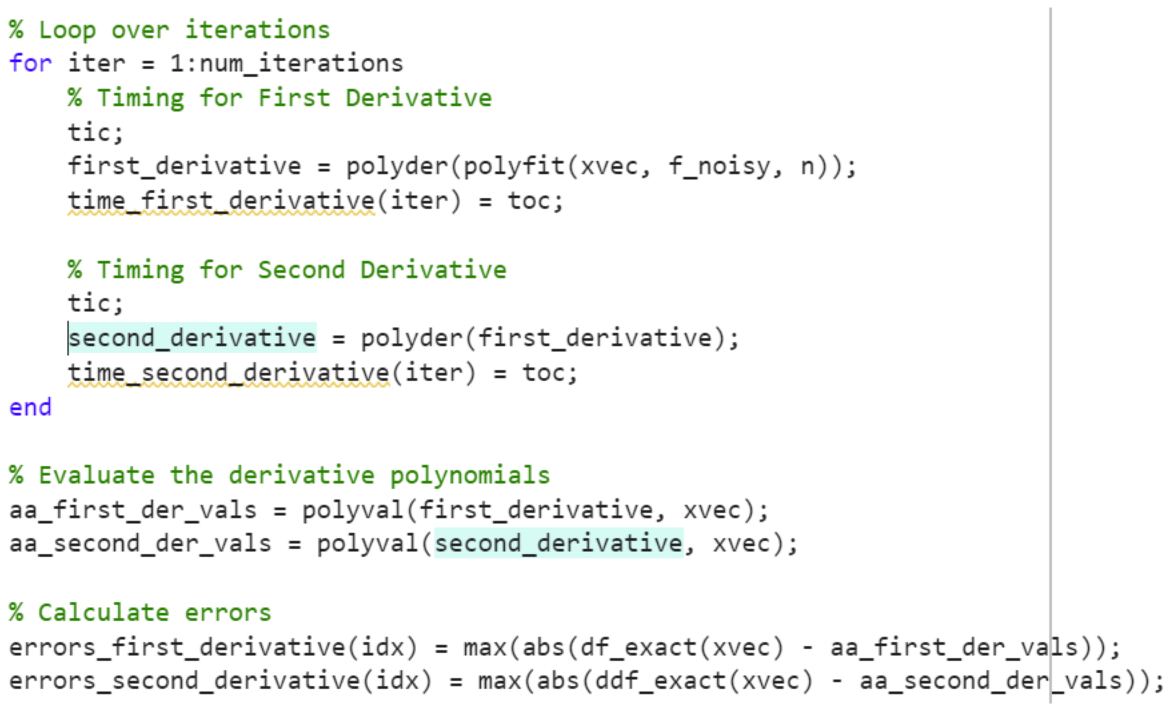
Code Snippet 15: "Setup"

3. **Main Loop:**

The code iterates over different step sizes, and for each step size, it performs polynomial fitting and calculates the derivative polynomials using **polyfit** and **polyder**. It also measures the time taken for each iteration.

4. **Error and mean time calculation:**

The code calculates the errors in the first and second derivatives by comparing them with the exact derivatives. The mean times for the first and second derivatives are calculated over multiple iterations.



```
% Loop over iterations
for iter = 1:num_iterations
    % Timing for First Derivative
    tic;
    first_derivative = polyder(polyfit(xvec, f_noisy, n));
    time_first_derivative(iter) = toc;

    % Timing for Second Derivative
    tic;
    second_derivative = polyder(first_derivative);
    time_second_derivative(iter) = toc;
end

% Evaluate the derivative polynomials
aa_first_der_vals = polyval(first_derivative, xvec);
aa_second_der_vals = polyval(second_derivative, xvec);

% Calculate errors
errors_first_derivative(idx) = max(abs(df_exact(xvec) - aa_first_der_vals));
errors_second_derivative(idx) = max(abs(ddf_exact(xvec) - aa_second_der_vals));
```

Code Snippet 16: "Error and mean time calculation"

Results:

The below table is the calculated order of accuracies from errors vs step size to evaluate Hassan's method:

The calculated orders are approximately what we expected them to be, but I believe because of the nature of function 2, the order of accuracies is not as accurate as function 1.

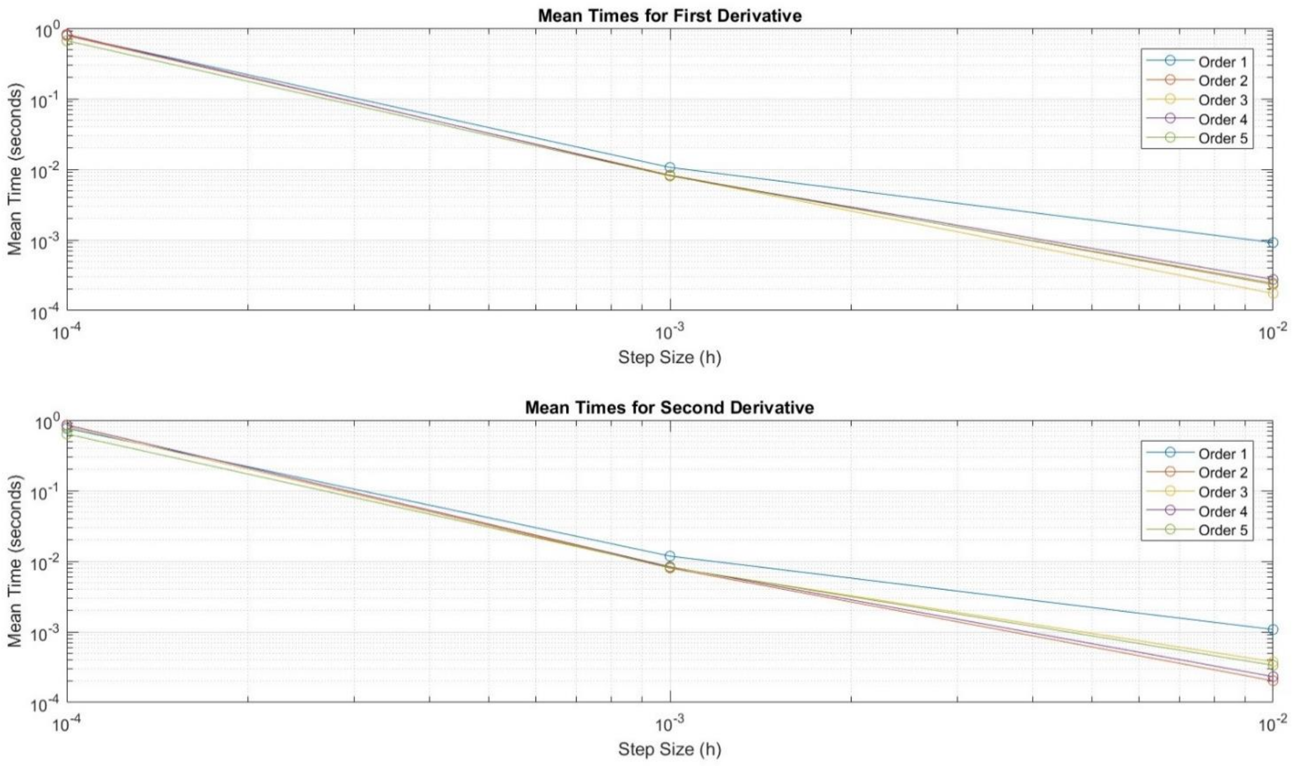
"Table 6. Comparison between actual and calculated O, Second derivative"

Actual order of accuracy	**1**	**2**	**3**	**4**	**5**
Calculated order of accuracy	**0.98**	**1.92**	**3.21**	**3.56**	**5.13**

"Table 5. Comparison between actual and calculated O, First derivative"

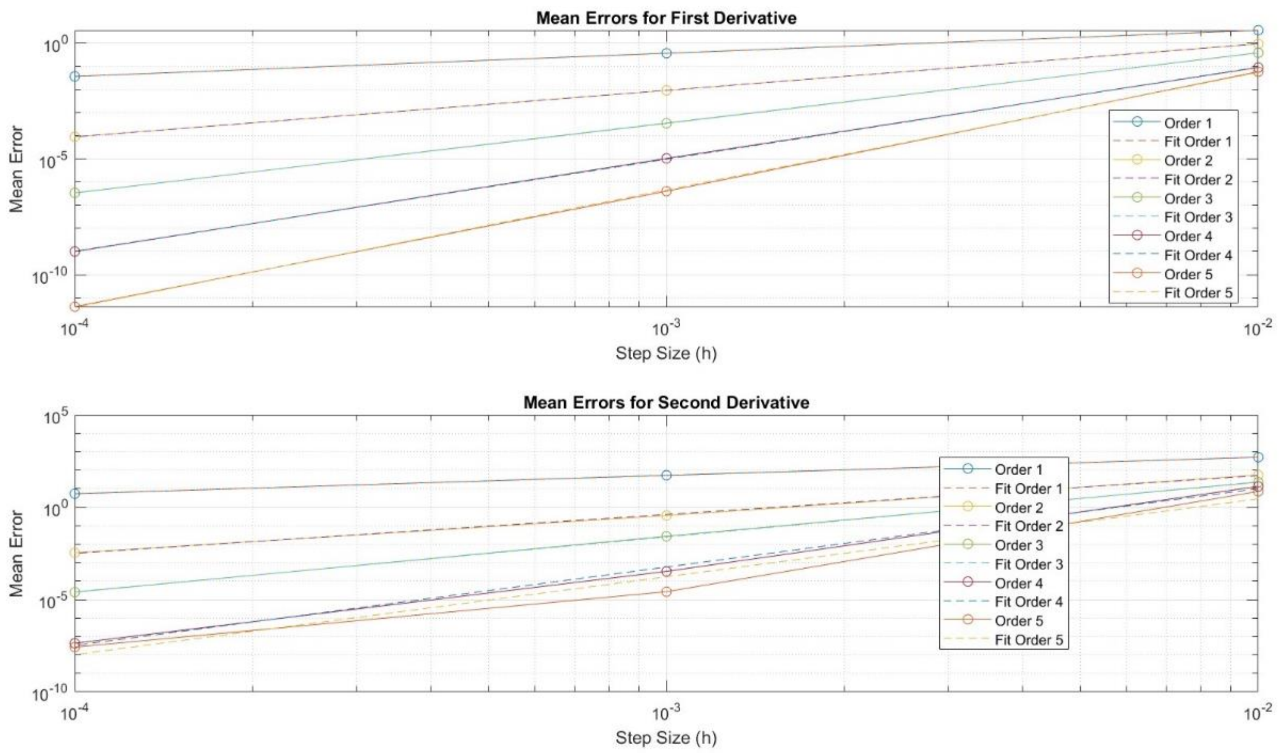
The actual order of accuracy:	**1**	**2**	**3**	**4**	**5**
Calculated order of accuracy:	**0.95**	**2.22**	**2.46**	**4.13**	**4.64**

The next figure is the mean time for this method for the first and second derivatives:



"Figure 8. The mean time for O = 1:5, 10 iterations, Hassan's method"

Again, the execution times for different orders of accuracy do not differ from each other, and this is the beauty of Hassan's method. Also, smaller step sizes, take more time and more work than bigger step sizes.



"Figure 9. Mean errors for O = 1:5, Hassan's method"

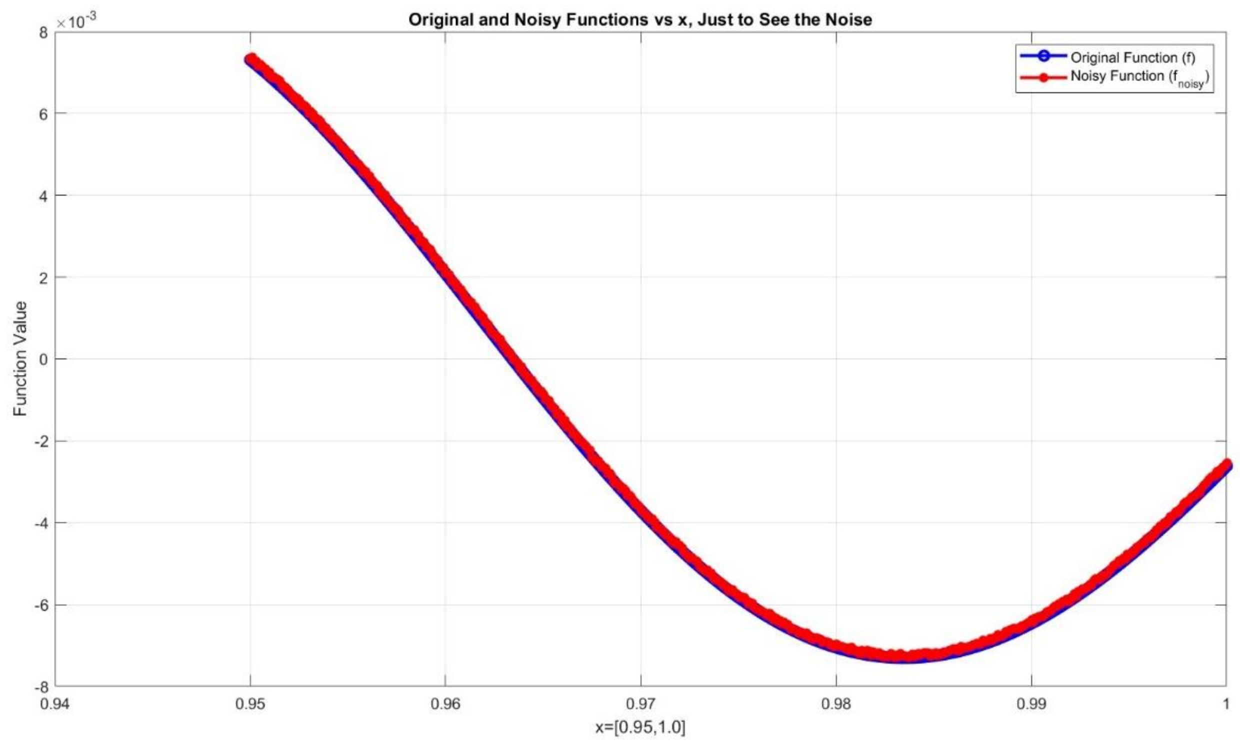
The errors for h = 0.01 are big, after I added the h = 0.001, 0.0001 the errors decreased significantly. Therefore, as the data points increased and step sizes decreased the errors were smaller and we have better efficiency.

Now, we are going to add noise to the original function and see how Hassan's method is going to perform:

The noise function:

$$noise = 0.0001 * rand(size(x)); \qquad \text{(EQ 3)}$$

The next figure is a part of original function and noisy function to visualize the effect of noise.

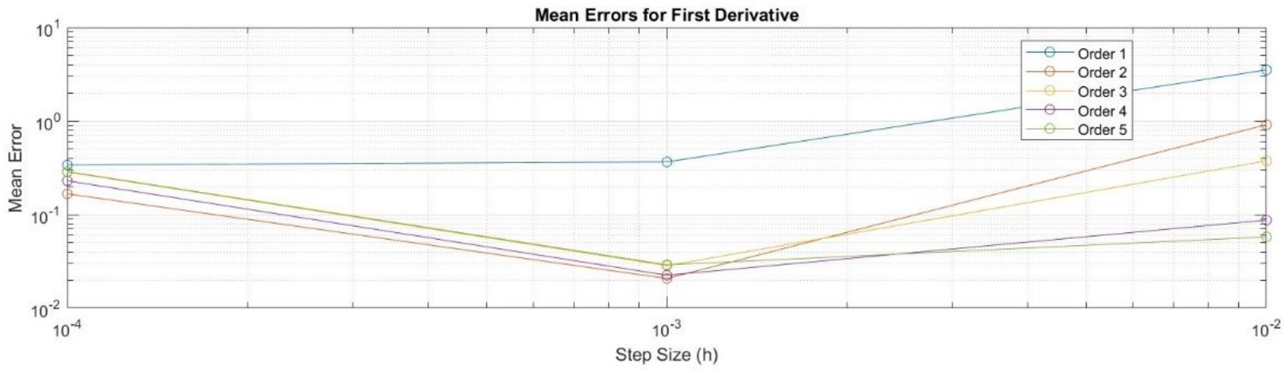


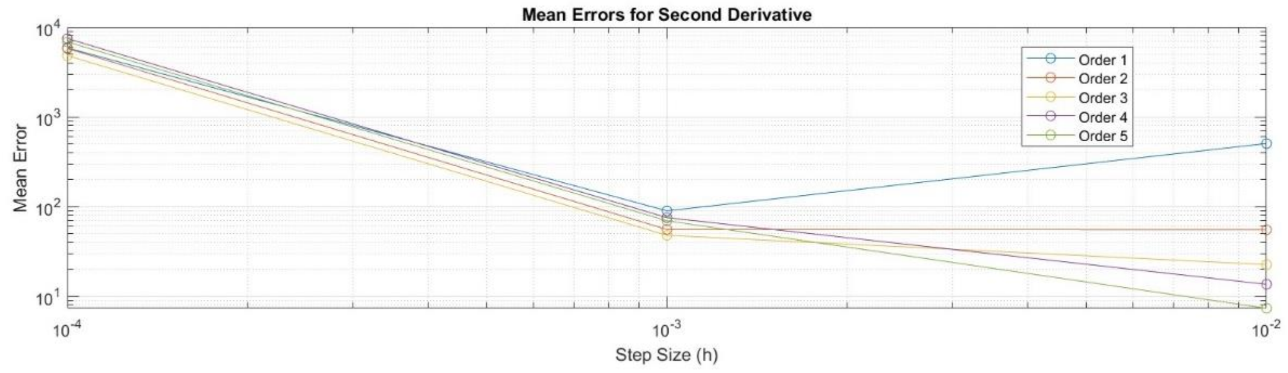
"Figure 10. Part of the original and noisy function"

When we add noise to the function, we introduce variability or random fluctuations to the values of the function. This noise can significantly affect the accuracy of derivative calculations, especially for higher-order derivatives. Therefore, when I calculate the derivatives of this noisy function, the noise contributes to the variations in the derivative values. The first derivative is sensitive to the noise added to the function. The noise introduces variations in the slope of the function, leading to errors in the first derivative calculation. Even if the noise is relatively small, it can still affect the accuracy of the first derivative, resulting in larger errors compared to the case without noise. When we calculate the second derivative, we are amplifying the effects of noise present in the first derivative. The second derivative is more sensitive to high-frequency variations, and noise often contains high-frequency components. This sensitivity leads to larger errors in the second derivative, especially when noise is present.

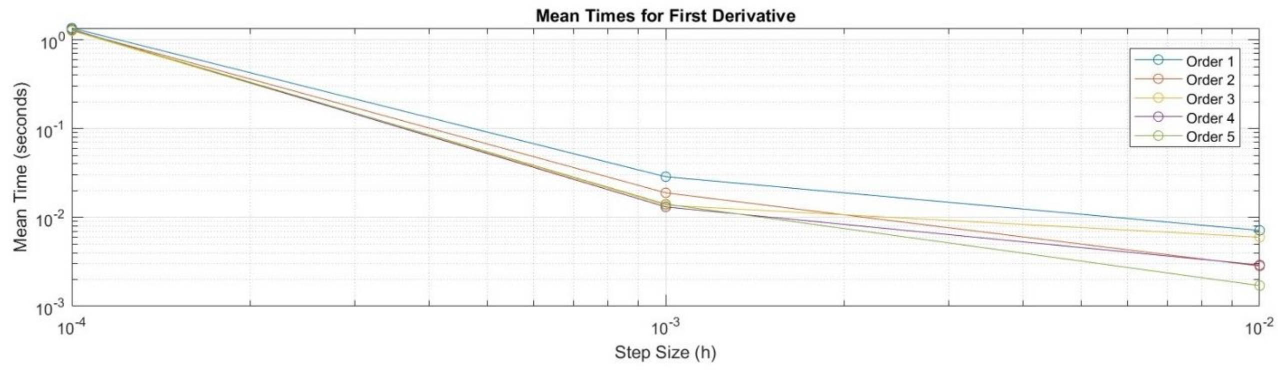
To mitigate the impact of noise on derivative calculations, we can consider using smoothing or filtering techniques on the noisy function before computing derivatives. These techniques can help reduce high-frequency noise and improve the accuracy of derivative estimates.

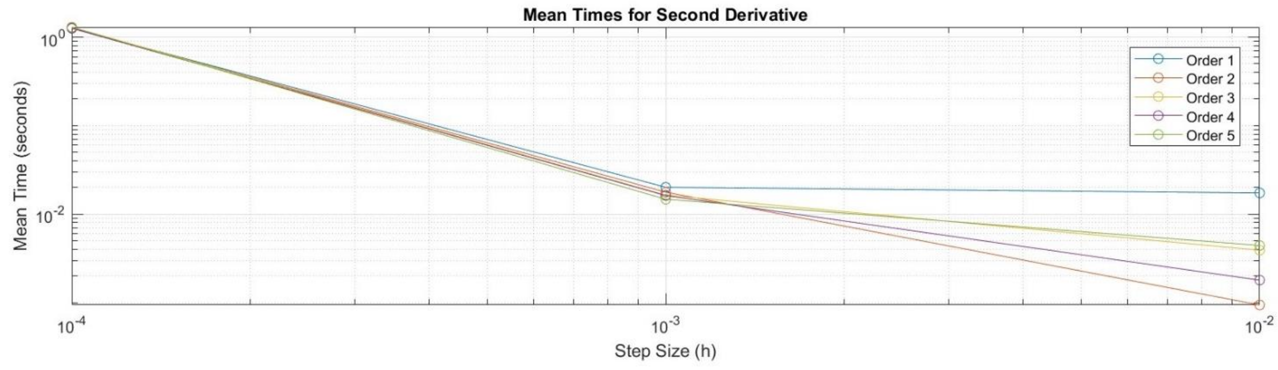
The next figure show the mean error and mean time for the noisy function.





"Figure 11. Mean errors for O = 1:5, Hassan's method_Noisy function"

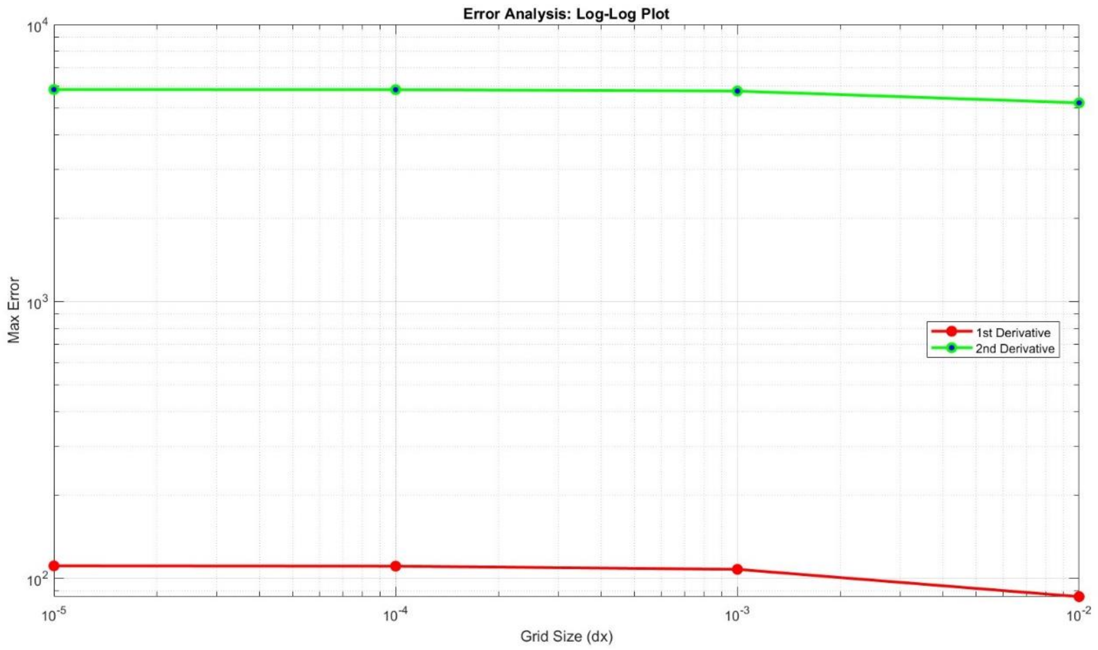




"Figure 12. Mean time for O = 1:5, Hassan's method_Noisy function"

In the next part, the results of methods 2 and 3 (ODE45 & Poly) are shown.

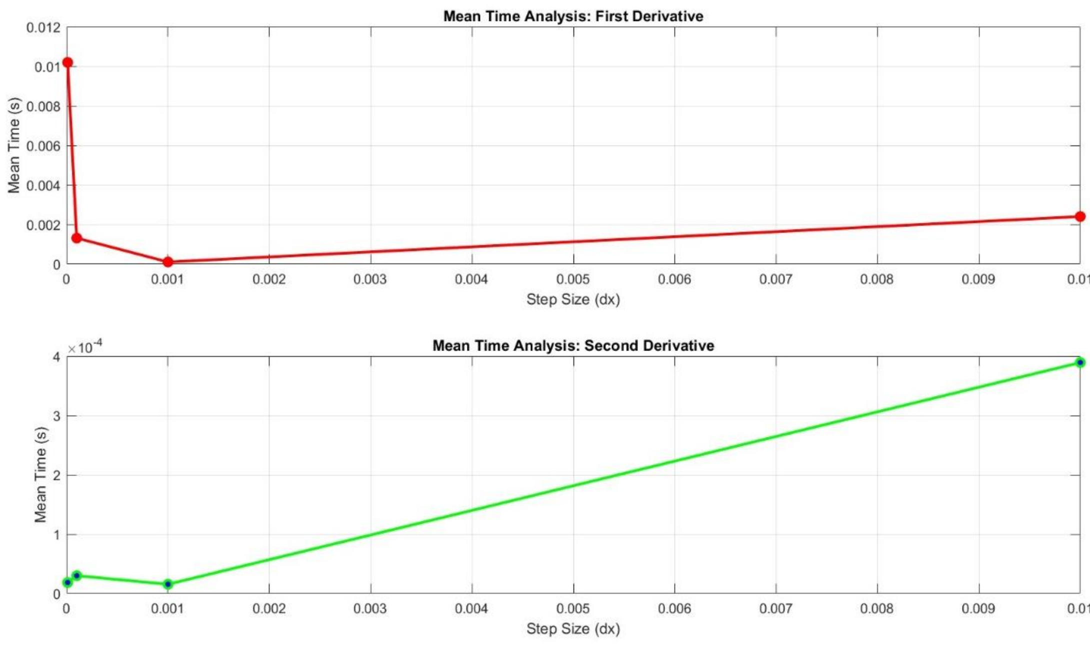
The next figures show the mean errors of ODE45 and Poly.



“Figure 13. Mean errors, Poly’s method_Noisy function”

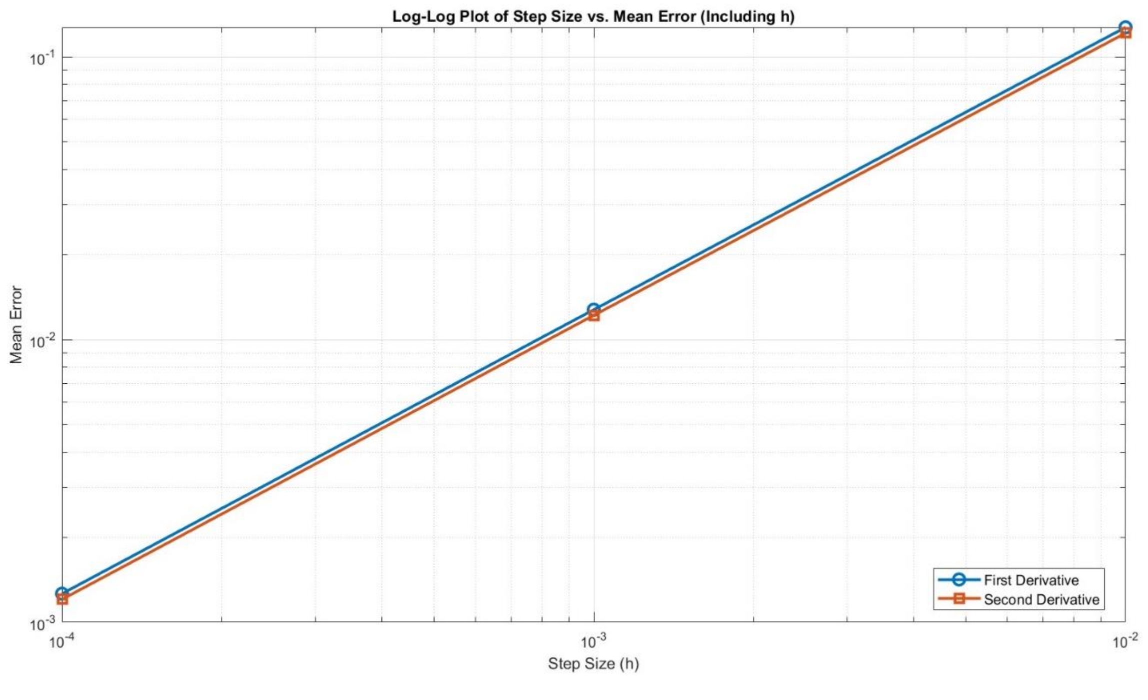
The errors of the poly method are very big and it is not efficient at all. The big errors in the computed derivatives can be influenced by several factors. For instance, the addition of random noise to the function values introduces variability, making it more challenging to accurately estimate derivatives. The polyfit function fits a polynomial to the data, and higher-degree polynomials may lead to oscillations and overfitting, especially when dealing with noisy data. Also, finite differences, which are used implicitly in the polynomial fitting process, introduce errors due to the discrete nature of the data points. Smaller step sizes (**dx**) can help reduce this error but might be limited by machine precision. Moreover, the process of differentiating polynomials can amplify numerical errors, especially for high-degree polynomials. This can lead to inaccurate results, particularly for the second derivative.

To improve the accuracy of derivative estimates, we can use more advanced numerical differentiation methods (like Hassan’s method). Or Employing techniques like regularization to mitigate overfitting. Also, we can experiment with different noise levels, step sizes, and polynomial degrees.

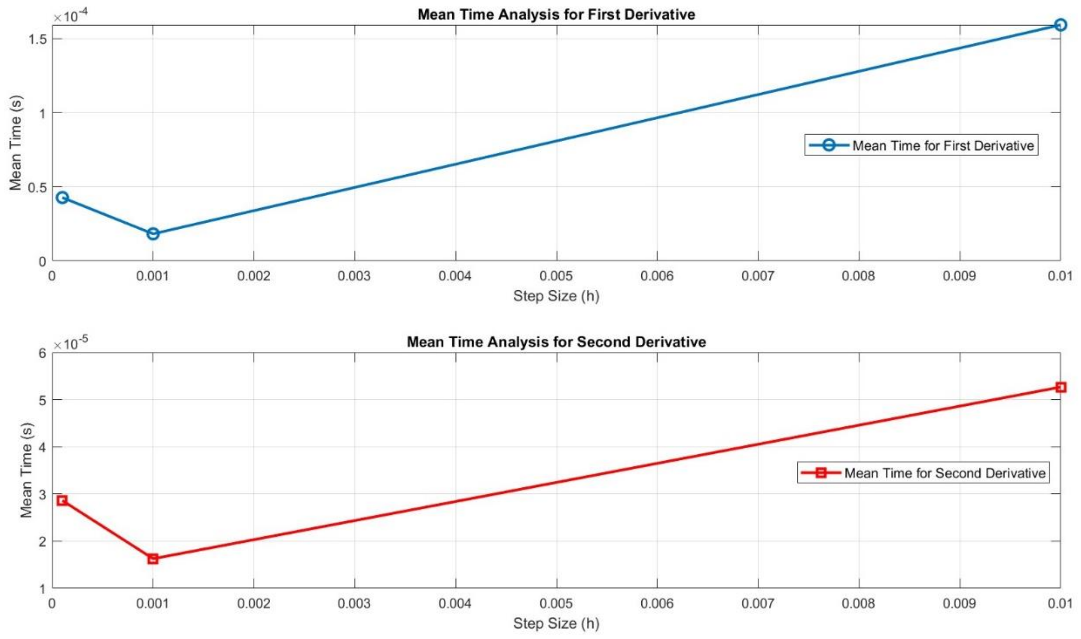


"Figure 14. Mean time, Poly's method_Noisy function"

It's a little odd but the mean time for the first derivative is bigger than mean time of second derivative. Also, as the step size gets bigger it takes more time to calculate the first and second derivative as we knew.



"Figure 15. Mean errors, ODE's method_Noisy function"



“Figure 16. Mean time, ODE’s method_Noisy function”

Upon thorough investigation and comparison of various methods for computing derivatives with different orders of accuracy, Hassan's method demonstrates notable advantages and efficiency. Specifically, Hassan's approach exhibits superior performance in terms of accuracy and computational efficiency when compared to alternative methods. The methodology proposed by Hassan proves to be a robust and reliable choice for obtaining derivatives with varying levels of precision. The efficiency gains and accuracy make it a favorable option for applications that require precise differentiation across different orders.

Last note:

I would like to express my gratitude for all your help and guidance throughout this semester. This project has been a valuable learning experience for me. The implementation of this novel method has not only expanded my knowledge but has also proven to be highly beneficial. This project provided an opportunity to delve into a new methodology, allowing me to gain practical experience in its implementation. Overall, the project has reinforced my understanding of various methods that we learned this semester and has allowed me to revisit and enhance my MATLAB skills, contributing to a deeper and more comprehensive grasp of the subject matter.

Also, I tried my best to write a good report with all the tips that you mentioned for each HW.

Thank you for your support, which has been instrumental in this semester.

Reference:

[1] H. Z. Hassan, A. A. Mohamad, and G. E. Atteia, “An algorithm for the finite difference approximation of derivatives with arbitrary degree and order of accuracy,” *J Comput Appl Math*, vol. 236, no. 10, pp. 2622–2631, Apr. 2012, doi: 10.1016/j.cam.2011.12.019.