Testing Adaptive Step-Size Algorithms for an Implicit Euler Solver Using the Newton-Raphson Method

Submitted By:
Aaron Jay Hinkle & Sophia Felicia Salome Döring

Final Project for: 3301 Numerical Methods of Simulation
Submission Date: March 1, 2023

Abstract

This report presents a project aimed at implementing and comparing the efficiency and accuracy of three different numerical methods for solving initial value problems (IVPs). One of the key contributions of the report is the implementation of the adaptive step size algorithms using the MATLAB programming language. The code is annotated and presented in detail, in a clear and easy-to-follow format. The methods compared were a constant step size implicit Euler solver using the Newton-Raphson method, an adaptive step size algorithm based on the ratio between the first and second derivative, and an adaptive step size algorithm based on minimizing the mean square error (MSE). Presented herein is a thorough discussion of the results for three different test IVPs, along with analytical solutions for comparison. In each case, the adaptive step size algorithms produced more accurate results than the constant step size method, with the MSE-based algorithm generally requiring the fewest iterations and resulting in the fastest computation time. The results showed that the adaptive step size algorithms produced more accurate results than the constant step size method, with the algorithm based on the second derivative showing the greatest improvement in accuracy, albeit at the cost of longer computation time. The algorithm based on minimizing the mean square error showed similar accuracy to the constant step size algorithm but required fewer iterations and was faster. The report also includes tables and figures that present the results for each method applied to three different test IVPs, as well as a critique of the results and suggestions for future research. Overall, the report provides a useful analysis of different numerical methods for solving IVPs. The authors present their results clearly and thoroughly, and provide valuable insights into the strengths and weaknesses of each method. The code presented in the report could be a valuable resource for other researchers or students interested in implementing these methods in their own work.

Keywords: Implicit Euler Method, Newton-Raphson Method, Adaptive Step Size, MATLAB, Step-Size Algorithm, Numerical Simulation, Numerical Solver, Ordinary Differential Equations

Contents

Abstract **i**

Contents **ii**

1 Introduction **1**
1.1 Implicit Euler Method . . . 1
1.2 Newton-Raphson Method . . . 1
1.3 Step-Size . . . 2
1.4 Adaptive Step Size . . . 2
1.4.1 Adaptive Step Size Control Method . . . 2
1.4.2 Predictor-Corrector Method . . . 3
1.5 Project . . . 4

2 Methodology **5**
2.1 Constant Step Size . . . 5
2.2 Second Derivative . . . 6
2.3 Mean Square Error . . . 7
2.4 Newton-Raphson Method . . . 8
2.5 Test Problems . . . 9
2.5.1 Analytical Solutions . . . 10
2.6 Accuracy and Computation Time . . . 14

3 Results **15**
3.1 Implicit Euler Method with a Constant Step Size . . . 15
3.2 Adapting Step Size Based on 2nd Derivative . . . 15
3.3 Adapting Step Size by Minimizing MSE Using Gradient Descent . . . 16
3.4 IVP 1 . . . 17
3.5 IVP 2 . . . 18
3.6 IVP 3 . . . 19

4 Discussion **21**
4.1 IVP 1 . . . 21
4.2 IVP 2 . . . 22
4.3 IVP 3 . . . 22
4.4 Analysis and Critique . . . 23
4.4.1 Newton–Raphson Method . . . 23
4.4.2 MSE Algorithm . . . 24

4.4.3 Second Derivative Algorithm 25
4.5 Difficulties and Complications 25

5 Conclusion **27**

Bibliography **29**

A Appendix A **30**
A.1 Code 30
A.1.1 Implicit Euler with Constant Step Size 30
A.1.2 Implicit Euler with Step Size Based on Second Derivative 32
A.1.3 Implicit Euler with Step Size Based on MSE 34
A.1.4 Analytical Solutions of Test IVPs 36
A.1.5 IVP2 36
A.1.6 IVP3 36
A.2 Organizational Documents 37

List of Figures **38**

List of Tables **39**

1 | Introduction

Implicit Euler Method

The implicit Euler method is used to solve differential equations in the form $u_t = f(u, t)$. It relies on the assumption that close to a point, a function and its tangent have nearly the same value. Graphically, the idea of the Euler scheme is to take the current value of U and its derivative at the current time, and to assume that it is approximately linear for some small change in time h [7]. In other words, drawing a secant line from $f(t)$ at a slope of $f'(t)$. The implicit Euler scheme simply uses the slope at the end of the line approximation, instead of at the beginning of it like the explicit version of the method [1]. When the function $f(u, t)$ is non-linear, then the implicit Euler method results in a set of non-linear equations that need to be solved for each time step. The Newton-Raphson method can be used for this purpose.
From the following truncated Taylor series expansion,

$$y_n \equiv y(t_{(n+1)} - h) = y(t_{(n+1)}) - h \cdot \frac{dy}{dt}|_{(t_{(n+1)})} + O(h^2),$$

comes the backward Euler scheme

$$y_{(n+1)} = y_n + h \cdot f(y_{(n+1)}, t_n + h).$$

This can be rearranged, set equal to zero and renamed as follows

$$F(y_{(n+1)}) = y_{(n+1)} - y_n - h \cdot f(y_{(n+1)}, t_n + h) = 0$$

where $y_{(n+1)} = x$. This means that

$$F(x) = x - y_n - h \cdot f(x, t_n + h) = 0.$$

The Newton-Raphson method can be applied to find the root of $F(x)$, which in turn becomes the solution to the differential equation at the next timestep [7].

Newton-Raphson Method

The Newton-Raphson method is a well-known technique for solving equations in the form $F(x) = 0$ by successive approximation. An initial guess x_0 is chosen such that $F(x_0)$ is reasonably close to 0. The x-intercept of the tangent line evaluated at that point is then

used as the guess for the next iteration of the algorithm, which is repeated until some suitable level of tolerance is reached [1]. The recursive form of the method is

$$x_{(i+1)} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Step-Size

The Euler Method is a first-order method, meaning the error per step (*local error*) is proportional to the square of the step size. However, the stability of the solver is not guaranteed for all step sizes. The criterion for stability depends on the ODE being considered. If the solution varies rapidly over a portion of the time interval and slowly over the remaining time, then using a fixed time step is inefficient. Using a variable step size is particularly important when there is large variation in the magnitude of the derivative of the function, but if it is implemented efficiently, then it can also serve to reduce computation time since fewer time steps are required for relatively "smooth" functions [7].

Methods for implementing a variable step size require a means for estimating the next time step. If the error at time t_n is known, then it can be used as the criterion for acceptance or rejection based on some level of tolerance. Furthermore, if the step is accepted, then the error can be used to estimate the next time step.

Adaptive Step Size

Adaptive step sizes are useful for numerical integrators because they allow the algorithm to automatically adjust the step size based on the accuracy of the solution. This can improve the efficiency of the integration, since the step size can be increased when the solution is accurate and decreased when the solution is less accurate. This can reduce the total number of steps required to reach the desired tolerance, which can result in a significant reduction in computation time.

Adaptive step size methods are particularly useful for solving systems of differential equations, since the error in the solution can vary significantly depending on the complexity of the system and the initial conditions [1, 7]. By automatically adjusting the step size based on the error, adaptive step size methods can ensure that the solution is accurate while minimizing the total computation time.

Adaptive Step Size Control Method

One commonly used variable step size method for implicit Euler solvers that use Newton's method for each step is the adaptive step size control method. This method adjusts the step size in each iteration based on the convergence of Newton's method and the error tolerance specified by the user. If the solution does not converge within a certain number of iterations or the error is above the tolerance, the step size is reduced and the iteration is repeated. If the solution converges within the specified tolerance, the step size is increased

for the next iteration. This method can improve the efficiency of the solver by avoiding unnecessary steps and reducing the total computation time [2].

The Jacobian matrix and residual vector for the system being solved are used in Newton's method to calculate the update for the solution. The error is calculated as the difference between the current and previous solutions, normalized by the previous solution. If the error is within the specified tolerance, the step size is increased by a factor of α. If the error is above the tolerance, the step size is reduced by a factor of α and the iteration is repeated with the new step size. The process is repeated until the error is within the tolerance or the maximum number of iterations is reached [7].

One potential drawback of the adaptive step size control method is that it can be computationally expensive, since it requires recalculating the Jacobian matrix and residual vector and solving the system of equations using Newton's method in each iteration. Additionally, if the step size is adjusted too frequently, it can lead to additional overhead and reduced efficiency [1]. Another potential drawback is that the method may not always converge to the desired tolerance, especially if the initial step size is too large or if the system being solved is highly nonlinear. In these cases, the step size may need to be reduced significantly to achieve convergence, which can further increase the total computation time [7]. The choice of the step size control parameter α can also affect the performance of the method. If α is set too low, the step size may be adjusted too frequently, leading to reduced efficiency. If α is set too high, the step size may not be adjusted frequently enough, and the error may not converge to the desired tolerance within the maximum number of iterations [2].

Predictor-Corrector Method

Another commonly used variable step size method for implicit Euler solvers that use Newton's method is the predictor-corrector method. This method involves taking two steps at each iteration: a "predictor" step and a "corrector" step.

In the predictor step, an approximate solution is obtained using a rough estimate of the step size. This can be done using a simple forward Euler step or a more accurate method such as the trapezoidal rule [7]. In the corrector step, the solution is improved by using a more accurate estimate of the step size based on the difference between the approximate solution from the predictor step and the true solution. This can be done using a modified form of Newton's method, where the Jacobian matrix is approximated using the difference between the approximate and true solutions [1]. The step size can then be adjusted based on the error between the approximate and true solutions. If the error is within the desired tolerance, the step size can be increased for the next iteration. If the error is above the tolerance, the step size is reduced and the iteration is repeated.

This method can be more efficient than the adaptive step size control method, since it only requires one solution using Newton's method in each iteration, rather than recalculating the Jacobian and residual at each step. However, it can still be computationally expensive, especially if the step size needs to be adjusted frequently [2].

Project

One of the main drawbacks of the Implicit Euler method is that it requires the use of a fixed step size, which can be inefficient in certain cases. Variable step size algorithms can improve the efficiency of the Implicit Euler method by adjusting the step size at each iteration based on certain criteria. The aim of this project is to compare the performance of three different step size algorithms for an Implicit Euler solver that uses the Newton Method at each iteration.

The analysis will compare three novel algorithms that were developed for this project. To evaluate the performance of the algorithms, a set of test problems with known analytical solutions will be used. This will allow comparison between the computed solutions and the true solutions in order to measure the accuracy and efficiency of the new algorithms. The results of this comparison will provide insights into the strengths and weaknesses of the different algorithms and assist in identifying the best solver choice for a given IVP.

This project necessitates a search for methods which allow a variable time step to be taken efficiently. To create such methods, a means to estimate the next time step is needed. One method is to use the behavior of the differential equation itself to determine the step size. If the solution is smooth and continuous, then the step size can be increased for the sake of computational effort. If however, the function is erratic and changing frequently or suddenly with respect to time, then the step size should be decreased in order to maintain accuracy. A second method is based on the error of the solution. If an estimate for the error made at time t_n can be obtained, then the magnitude of the error can be the criteria by which the time step is accepted or rejected; and if the step is accepted, to determine the value of the next time step. This paper will analyse two methods that were based on these two strategies. The first adaptive step size algorithm is based on the second derivative, and the second is based on the mean square error.

2 | Methodology

A program was written for each method to solve three different ODEs with known analytical solutions. The programming of the different methods was coded in MATLAB (MATLAB. (2022). version 9.13.0 (R2022b). Natick, Massachusetts: The MathWorks Inc.). For every method a subfile was created, where the function to solve and the initial condition were updated for every IVP that was solved.

Constant Step Size

The program of the constant step sized was structured as described in section 1.1. The overall structure of the program can be seen in 2.1. The initial conditions, the boundaries, and the function to solve were adjusted to the ODE to solve. The constant step size h for all calculations and results presented was 0.01. All programs had a maximum of 10000 newton steps. The detailed description of the function of the Newton-Raphson can be found in Section 2.4.

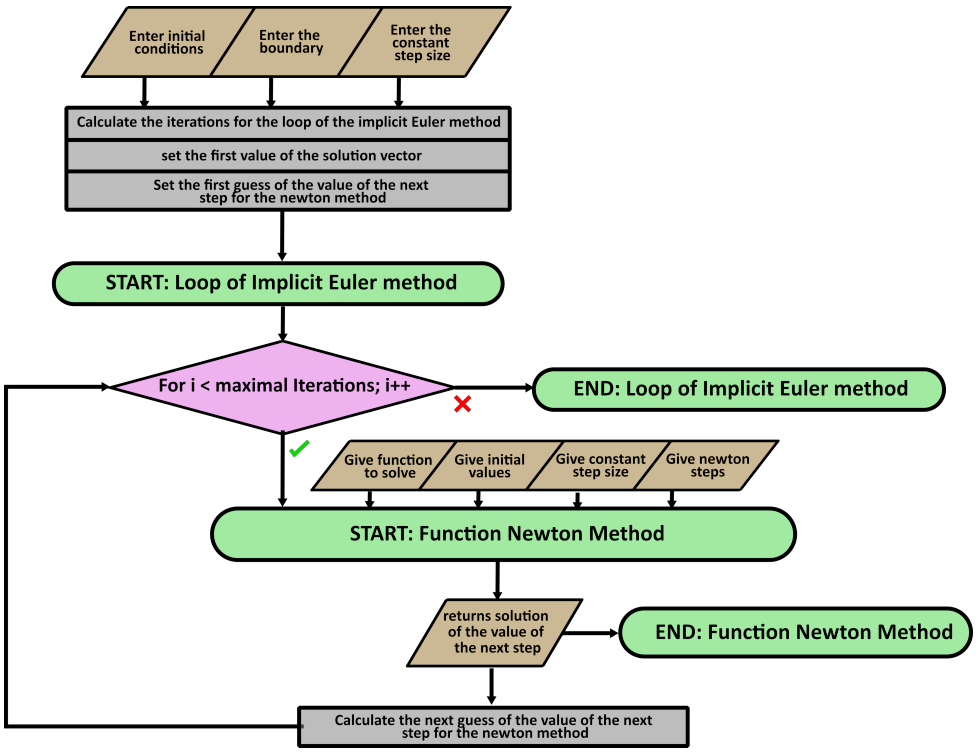

Figure 2.1: Flowchart for the implicit Euler method with a constant step size.

Second Derivative

When the second derivative is large compared to the first derivative, it indicates that the rate of change of the function is large, which means that the function is changing rapidly. In this case, using a smaller step size can help to ensure that the solution is accurate and stable. The idea behind using an adaptive step size is to adjust the step size h in such a way that the solution is accurate and stable, while minimizing the computational cost. One way to do this is to use the second derivative as a measure of the "smoothness" of the function. When the second derivative is large, it indicates that the function is changing rapidly, so a smaller step size may be needed to accurately capture the behavior of the function. On the other hand, when the second derivative is small, it indicates that the function is changing slowly, so a larger step size can be used without compromising the accuracy of the solution.

The program for the second derivative method was structured as shown in 2.2. The maximal step size h of this method was set to be the same as the constant step size 0.001. The structure of the Newton-Raphson Method function is described in detail in chapter 2.4.

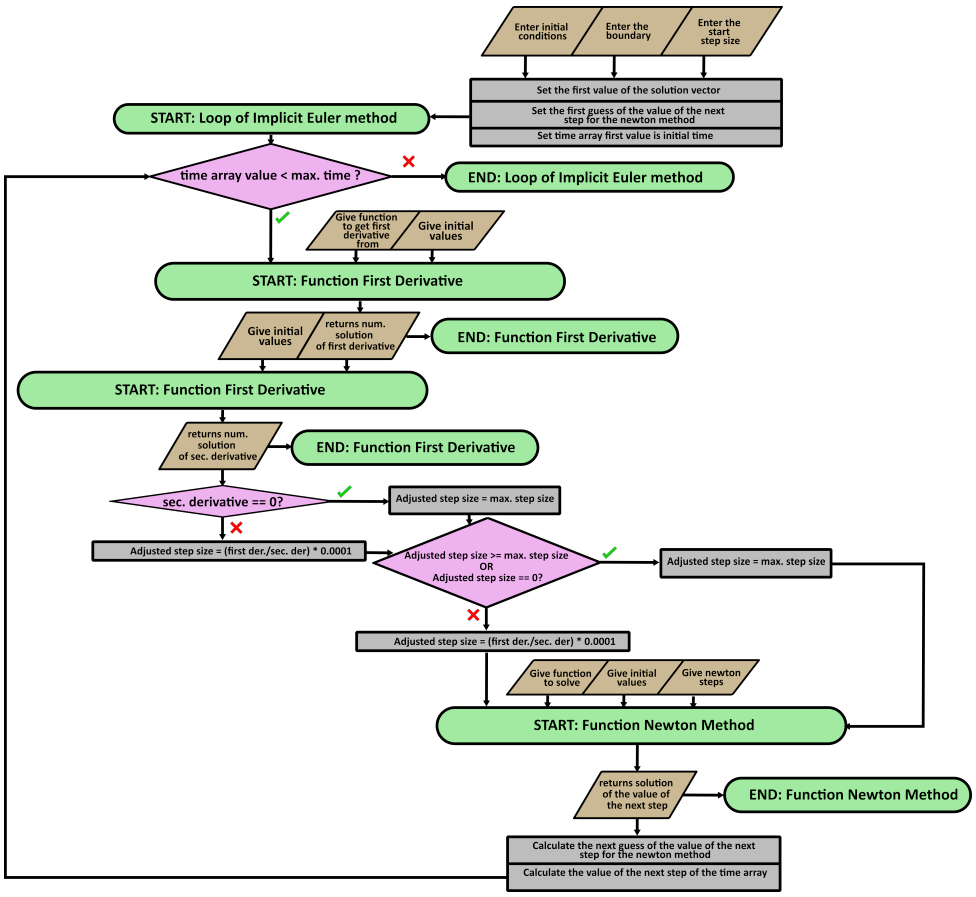

Figure 2.2: Flowchart for the implicit Euler method with step size based on the second derivative.

Mean Square Error

The mean squared error (MSE) is a measure of the difference between the true value and the estimated value of a quantity. Seen in Equation (2.1), it is defined as the average of the squares of the differences between the true value and the estimated value. In other words, the MSE is the average of the squares of the errors made in the estimates. The MSE is commonly used in statistical modeling and is a measure of the overall fit of a model to a set of data. It is often used to compare the performance of different models, with a lower MSE indicating a better fit. The MSE can be calculated as the sum of the squared errors divided by the number of samples, or as the sum of the squared differences between the true and estimated values divided by the number of samples minus one. The formula for the MSE is:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{2.1}$$

where y_i is the true value of the ith sample, $\hat{y}_i$ is the estimated value of the ith sample, and n is the total number of samples. The formula for the root mean squared error (RMSE) is the square root of the MSE:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}, \tag{2.2}$$

In the context of an implicit Euler solver, the RMSE can be used to measure the difference between the approximate solution obtained using the implicit Euler method and the true solution to the differential equation. To use the RMSE as a measure of error in an adaptive step size algorithm, the h is initialized at a small value and the differential equation is solved using that step size. The RMSE between the approximate solution obtained in this step and the true solution is then computed. If the RMSE is above a certain threshold, then this indicates that the approximate solution is not accurate enough, and the step size needs to be reduced in order to improve the accuracy of the solution. h is then reduced by some factor and the differential equation is solved again using the new step size.

On the other hand, if the RMSE is below the threshold, then this indicates that the approximate solution is accurate enough, and h can be increased in order to reduce the computational cost of the algorithm. This process is repeated until the RMSE is within the desired tolerance. The idea is to adjust the step size so that the RMSE is minimized, while still maintaining an acceptable level of accuracy.

In the program, the MSE is used to find the optimal h that minimizes the error to 0.001. To minimize the MSE function by adjusting h, the gradient descent method was used:

$$h_{i+1} = h_i - \alpha * f'(h) \tag{2.3}$$

Where h_{i+1} is the next guess of the location of a minimum, h_i is the last guess, α is the step size, and $f'(h)$ is the derivative of the target function. If the absolute difference between h_{i+1} and h_i is smaller than the error, then the location of the minimum has likely been found.

The program of the MSE method was structured as shown in 2.3. The initial step size for this method was set to be the same as for the constant step size, 0.001. The structure of the Newton-Raphson Method function is described in detail in chapter 2.4. The derivative of the function was calculated numerically.

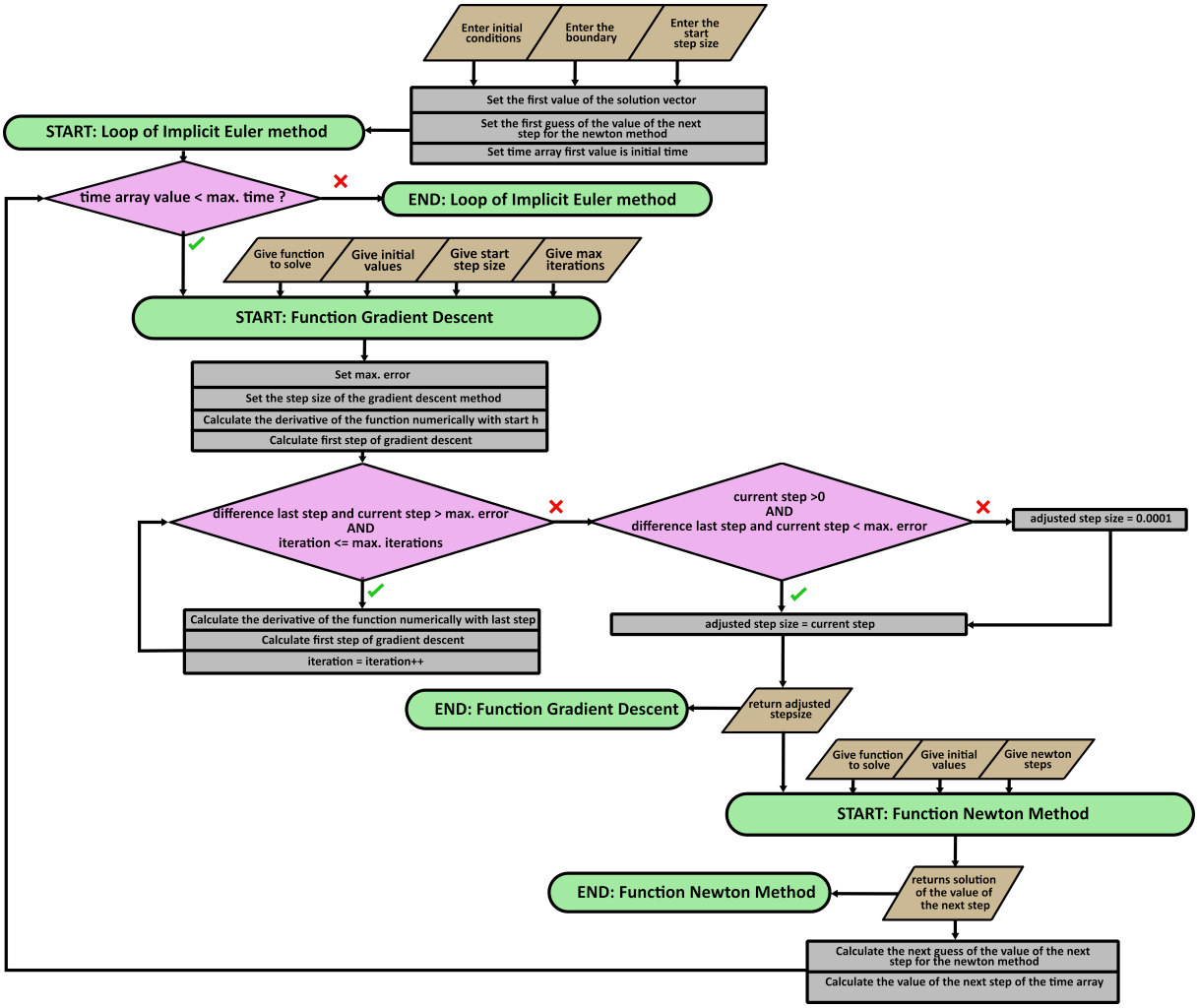

Figure 2.3: Flowchart for the implicit Euler method with step size based on minimizing MSE.

Newton-Raphson Method

The Newton-Raphson method was used at each iteration for all three implementations of the implicit Euler method, and the same programmed function was used in each of them. The general description of the Newton-Raphson method can be found in chapter 1.2. The structure of the function can be seen in 2.4.

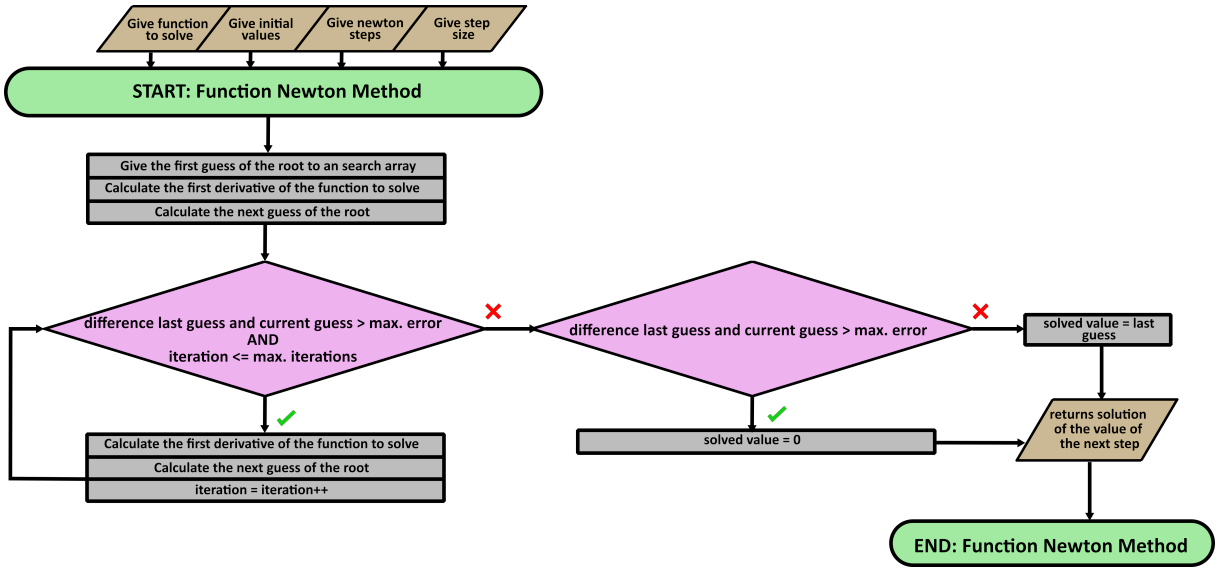

Figure 2.4: Flowchart for the Newton-Raphson Method program.

Test Problems

A few sample initial value problems (IVPs) with known analytical solutions were chosen to test the variable step size algorithms:

1. Linear differential equation:

$$\begin{aligned} y'(t) + 4 \cdot y(t) - 8 \cdot t &= 0 \\ y(0) &= 1 \end{aligned} \tag{2.4}$$

2. Nonlinear differential equation:

$$\begin{aligned} y' - \cos(t) y^2 &= 0 \\ y(0) &= 1 \end{aligned} \tag{2.5}$$

3. ODE with variable coefficients:

$$\begin{aligned} y'(t) + 2 \cdot t \cdot y(t) &= 0 \\ y(0) &= 1 \end{aligned} \tag{2.6}$$

These IVP's were chosen because they have different properties (e.g. linear vs nonlinear, variable coefficients), which can affect the performance of the numerical solvers. These examples will be used to compare the behavior of the different algorithms and to see which ones perform the best for each problem.

Analytical Solutions

IVP 1: Linear IVP

$$y'(t) + 4y(t) - 8t = 0, \quad y(0) = 1$$

This is a first-order linear differential equation that can be solved using an integrating factor. An integrating factor is a function that can be multiplied with the differential equation to transform it into an equation that can be solved by integration. The integrating factor for this equation is given by

$$\mu(t) = e^{4t}.$$

Multiplying the differential equation by the integrating factor gives

$$4e^{4t}y'(t) + 4e^{4t}y(t) = 8te^{4t}.$$

This can be rewritten as

$$\frac{d}{dt}(e^{4t}y(t)) = 8te^{4t}.$$

Integrating both sides of the equation with respect to t gives

$$e^{4t}y(t) = \int 8te^{4t}dt.$$

Then, integration by parts can be used.

$$\int 8te^{4t}dt = 2te^{4t} - \int 2e^{4t}dt = 2te^{4t} - \frac{1}{2}e^{4t} + C$$

where C is the constant of integration. Substituting this back into the previous equation gives

$$e^{4t}y(t) = 2te^{4t} - \frac{1}{2}e^{4t} + C.$$

Using the initial condition $y(0) = 1$, results in

$$e^{0}y(0) = 2 \cdot 0 \cdot e^{4 \cdot 0} - \frac{1}{2}e^{4 \cdot 0} + C,$$

which simplifies to $C = \frac{3}{2}$. Substituting this value of C back into the equation gives

$$e^{4t}y(t) = 2te^{4t} - \frac{1}{2}e^{4t} + \frac{3}{2}.$$

Dividing both sides by e^{4t} gives the final particular solution

$$y(t) = 2t - \frac{1}{2} + \frac{3}{2}e^{-4t}.$$

The analytical solution to the linear IVP can be seen in Figure 2.5. The code for generating the image can be found in Appendix A

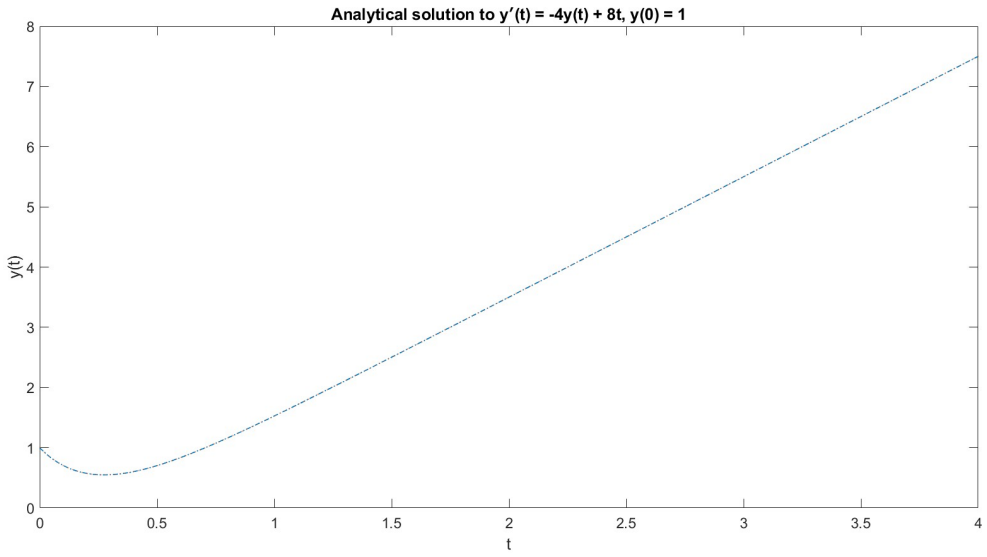

Figure 2.5: Analytical solution of (2.4)

IVP 2: Nonlinear IVP

$$y' - \cos(t)y^2 = 0, \quad y(0) = 1$$

The first step is to separate the variables t and y by multiplying both sides of the equation by dt and dividing both sides by y^2, resulting in

$$\frac{dy}{y^2} = \cos(t)dt$$

This equation can be written in the form of a separable differential equation, $M(y)dy = N(x)dx$, where $M(y) = \frac{1}{y^2}$ and $N(x) = \cos(x)$. Integrating both sides of the equation gives

$$\int \frac{1}{y^2}dy = \int \cos(x)dx$$

The integral on the left-hand side can be evaluated using the substitution $u = y^{-1}$, resulting in

$$-\frac{1}{y} = \sin(x) + C$$

where C is the constant of integration. Solving for y gives:

$$y = -\frac{1}{\sin(x) + C}$$

To find the particular solution for this differential equation, the initial condition $y(0) = 1$ is substituted into the equation resulting in

$$1 = -\frac{1}{\sin(0) + C} = -\frac{1}{C}$$

Hence, $C = -1$. Substituting this value of C back into the equation for y gives:

$$y = \frac{1}{1 - \sin(x)}$$

Therefore, the general solution to the given differential equation is:

$$y = \frac{1}{1 - \sin(x)}$$

which can be verified by plugging in the initial condition $y(0) = 1$. Note that this solution is only valid in the domain where $\sin(x) \neq 1$.

The analytical solution to the non-linear IVP can be seen in Figure 2.6. The code for generating the image can be found in Appendix A

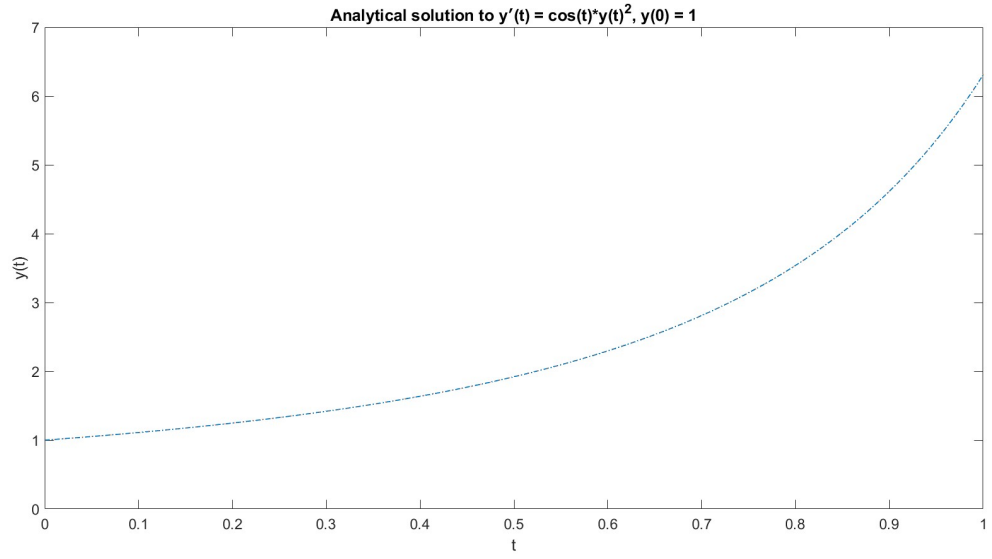

Figure 2.6: Analytical solution of (2.5)

IVP 3: ODE IVP with variable coefficients

$$y'(t) + 2t \cdot y(t) = 0, \quad y(0) = 1$$

This equation is linear but has variable coefficients due to the $2t$ term. It is a good test case for adaptive step size algorithms because the solution decays rapidly as t increases

so the numerical solver must adjust the step size appropriately to accurately capture the solution.

Separation of variables can be used to solve this IVP as well since it can be written as a separable equation in the form

$$\frac{dy}{dt} = -2t \cdot y.$$

Separating the variables gives

$$\frac{1}{y}dy = -2tdt.$$

Integrating both sides gives

$$\ln|y| = -t^2 + C,$$

where C is a constant of integration.

Applying the initial condition $y(0) = 1$ gives

$$\ln|1| = -0^2 + C,$$

so $C = 0$. Thus,

$$\ln|y| = -t^2,$$

which can be rewritten as

$$|y| = e^{-t^2}.$$

Since $y(0) = 1$, $|y(0)| = 1$, so the positive solution is used

$$y(t) = e^{-t^2}.$$

The analytical solution to the IVP with variable coefficients can be seen in Figure 2.7.

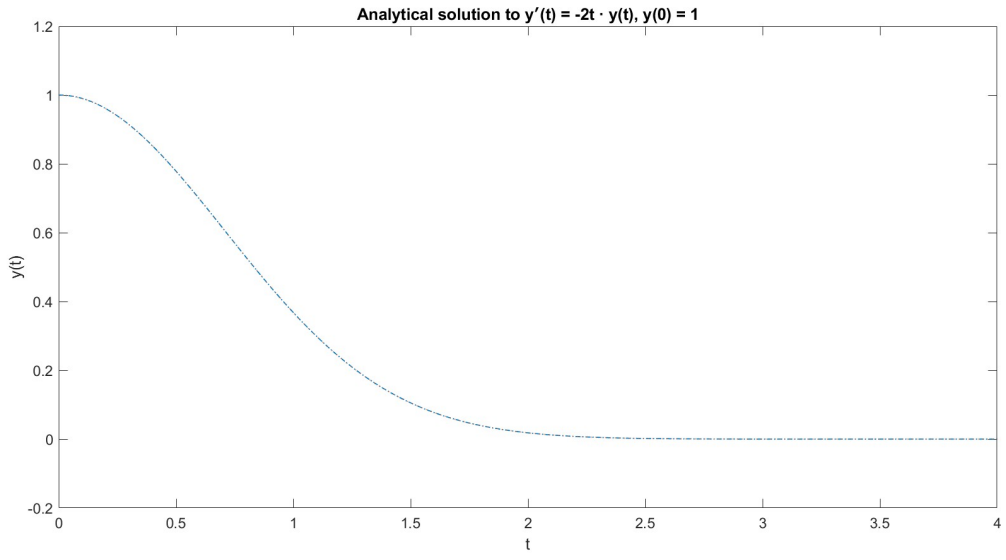

Figure 2.7: Analytical solution of (2.6)

Accuracy and Computation Time

To compare the different methods, the accuracy and the computation time of each method on each IVP is calculated. To calculate the accuracy, the average MSE was calculated for each step in a method. For computation time, the time to run the script was measured, also within the code. The reported computation time is the average time of three trials with each method on each IVP.

3 | Results

Implicit Euler Method with a Constant Step Size

The base script, called `ImplEuler_Constant_h.m` implements an implicit Euler numerical solver that discretizes the time domain using a constant step size `const_h` to approximate the solution to a differential equation defined by the function `dx_dt`. The code also calculates the MSE between the approximate solution obtained with the Euler solver and the actual solution of the differential equation.

The code first sets up the initial values for the time domain `t_start` and `t_end`, the step size `const_h`, and the initial condition for the solution `x_start`. The code also defines the number of iterations for the Newton method `step_newt`.

In the next step, the code sets up a loop that uses the Euler method to calculate the approximate solution of the differential equation. The loop iterates over the time domain and for each iteration, it uses the Newton method to solve the implicit equation obtained during the Euler step by repeatedly updating the value of x_i until the absolute value of f'(x_i) is less than the tolerance level `tol`. The guess for the Newton method is set as the previous solution plus `const_h`, and the solution for the Newton method is used as the new approximation of the solution.

The code then calculates the actual solution of the differential equation and the MSE between the approximate solution and the actual solution. Finally, the code visualizes the results by plotting the approximate and actual solutions and the MSE. The code also includes three helper functions. The `dx_dt` function defines the differential equation that needs to be solved. The `dx_dt_sol` function calculates the actual solution of the differential equation. The `newton` function implements the Newton method for finding the roots of the implicit equation obtained by the implicit Euler method. The `MSE` function calculates the Mean Square Error between the values.

Adapting Step Size Based on 2nd Derivative

The second script utilizes the same implementation of the Implicit Euler method, but this time the step size is adaptive based on the second derivative. To find the minimum of a function, the root of its first derivative (which corresponds to the point where the function changes from decreasing to increasing) can be used. A minimum is determined

by verifying that the second derivative is positive at that point. If the first derivative is small compared to the second derivative, the graph goes near a minima or maxima, and therefore an adjustment of h is warranted to reduce the error and increase stability.

The script starts by setting the initial values, function, and boundaries. Then, it solves the differential equation initially with a constant step size h using a while loop. Inside the loop, it calculates the first and second derivatives and adjusts h accordingly. Finally, it calculates the real values of x and the MSE and visualizes the solution.

The code defines several functions, including:
• `dx_dt`: calculates the derivative of the differential equation at a given point.
• `dx_dt_sol`: calculates the real solution of the differential equation.
• `dx_dt_prime`: calculates the derivative of `dx_dt` numerically using a small step.
• `newton`: implements the Newton-Raphson method to find the root of the differential equation.

The code also uses the `MSE` function to calculate the mean squared error between the calculated and real values of x. The solution is visualized using a tiled layout with two plots, one for the solution and the other for the MSE.

Adapting Step Size by Minimizing MSE Using Gradient Descent

Once again, the implicit Euler method is implemented using the Newton method at each time step, but this time, the goal is to adjust the step size h in a such a way as to minimize the MSE by using gradient descent.

Gradient descent is a widely used optimization algorithm that can be used to minimize a given objective function. The basic idea is to iteratively update the parameters of the objective function in the direction of the negative gradient of the function, which corresponds to the steepest descent direction of the function [9].

More formally, the update rule for gradient descent can be expressed as:

$$\theta = \theta - \alpha \cdot \nabla J(\theta)$$

where θ represents the current set of parameters, α is the learning rate that determines the step size of each iteration, J is the objective function to be minimized, and $\nabla J(\theta)$ is the gradient of the objective function with respect to the parameters θ.

At each iteration, the algorithm computes the gradient of the objective function with respect to the current set of parameters and updates the parameters by moving in the direction of the negative gradient with a step size determined by the learning rate. The algorithm continues until a stopping criterion is met, such as the convergence of the ob-

jective function or a maximum number of iterations [9].

In the first part of the code, some initial values are set for x, t, h, and the step size for Newton's method, which is used in the gradient descent function. The error is also set to a small value.

Then, the while loop starts for the Euler method. At the beginning of the loop, the time `t_start` and the first element of the `h`-array are initialized. The solution array `x_sol` is also initialized with the starting value of `x_start`. Within the while loop, the `gradient descent` function is called to find the value of h that minimizes the MSE. The `MSE` function takes two inputs: the current solution value and the true solution value, which is calculated using the function `dx_dt_sol`. The MSE is then minimized using gradient descent by adjusting the value of h. The adjusted value of h is stored in the `h`-array and is used to calculate the next value of x using Newton's method. The new value of t is then calculated using the adjusted value of h and the loop continues until `t_array(i)` is greater than `t_end`, at which point the loop stops.

The second part of the code calculates the true values of x and the MSE for the calculated values of `x_sol`. This is done using a for loop that iterates over the calculated values of `x_sol` and compares them with the true values of x, which are calculated using `dx_dt_sol`. The MSE is once again calculated using the `MSE` function, and the results are plotted as before with the numerical solution and the true solution in the first plot, and the MSE in the second.

IVP 1

The results of the first IVP are shown in Figure 3.1 and Table 3.3.

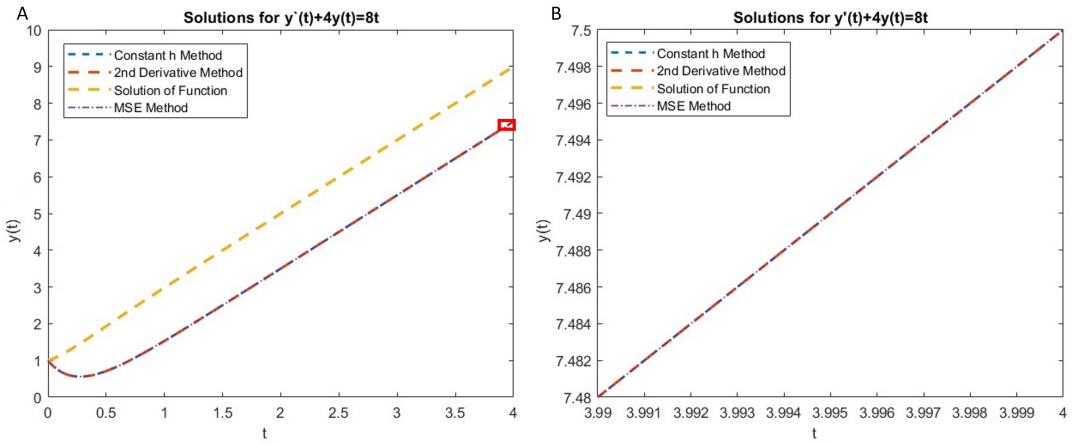

Figure 3.1: Results of the three methods applied on the Function (2.4). Graph A shows the results of the methods and the real solution in the domain t ([0,4]); Graph B is a close-up of the red rectangle to see the difference between the methods.

Table 3.1: **Results of IVP 1.** The number of iterations for solving IVP 1, the average computation time, and the accuracy of each method.

Method	Iterations	Computation time (s)	Accuracy
Constant step size	401	0.527	1.999
Second derivative	402	0.537	1.999
Mean square error	384	0.519	1.988

Figure 3.2 shows the calculation of the MSE for every calculated point. For every point, each method was compared with the analytically calculated value of x at the same time step.

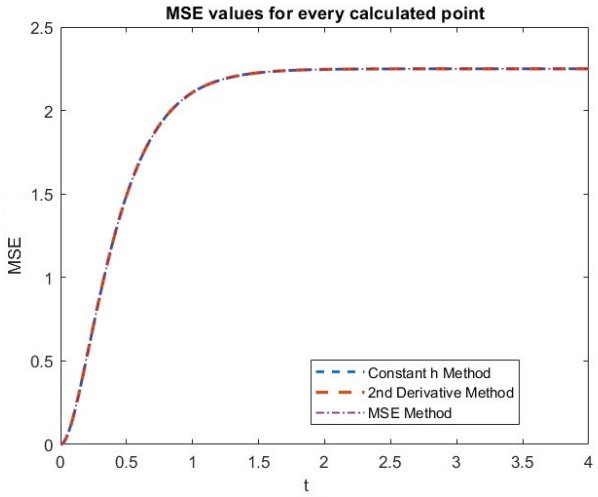

Figure 3.2: Results of the MSE for every step for all methods on Equation (2.4).

IVP 2

The results of the first IVP are shown in Figure 3.3 and Table 3.2.

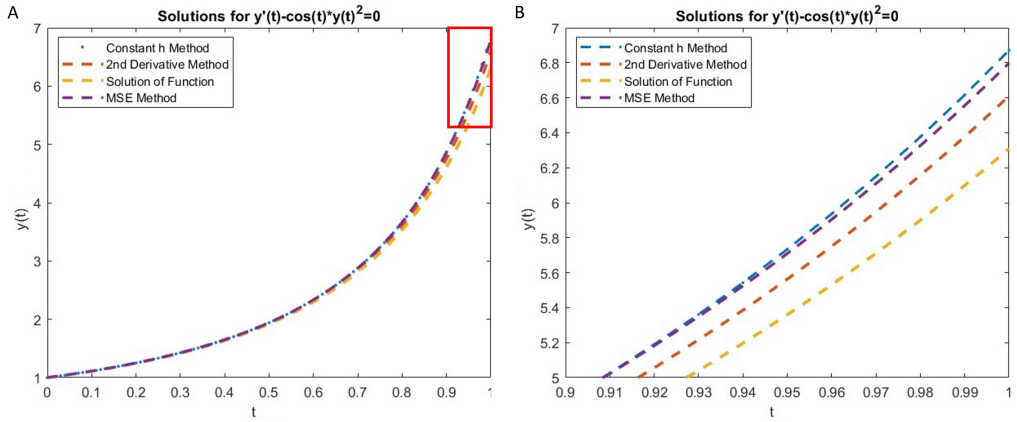

Figure 3.3: Results of the three methods applied on the Function (2.6). Graph A shows the results of the methods and the real solution within the boundary of t ([0,1]); Graph B is a close up of the red rectangle to see the difference between the methods.

Table 3.2: **Results of IVP 2.** The number of iterations for solving IVP 2, the average computation time, and the accuracy of each method.

Method	Iterations	Computation time (s)	Accuracy
Constant step size	101	0.699	0.0219
Second derivative	3022	0.741	0.0028
Mean square error	983	5.198	0.1276

Figure 3.4 shows the calculation of the MSE for every calculated point. For every point, each method was compared with the analytically calculated value of x at the same time step.

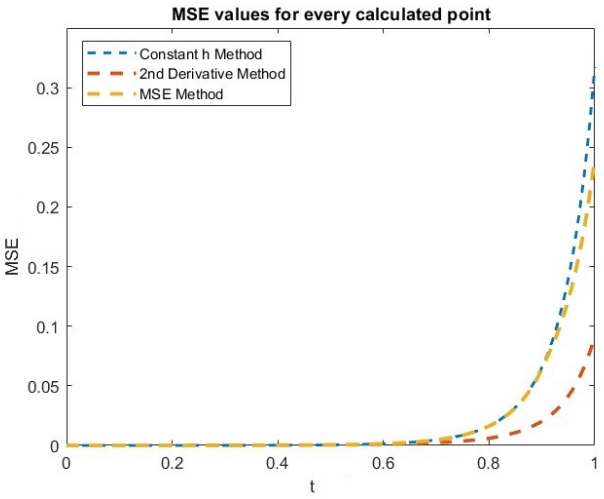

Figure 3.4: Results of the MSE for every step for all methods on Equation (2.5).

IVP 3

The results of the first IVP are shown in Figure 3.5 and Table 3.3.

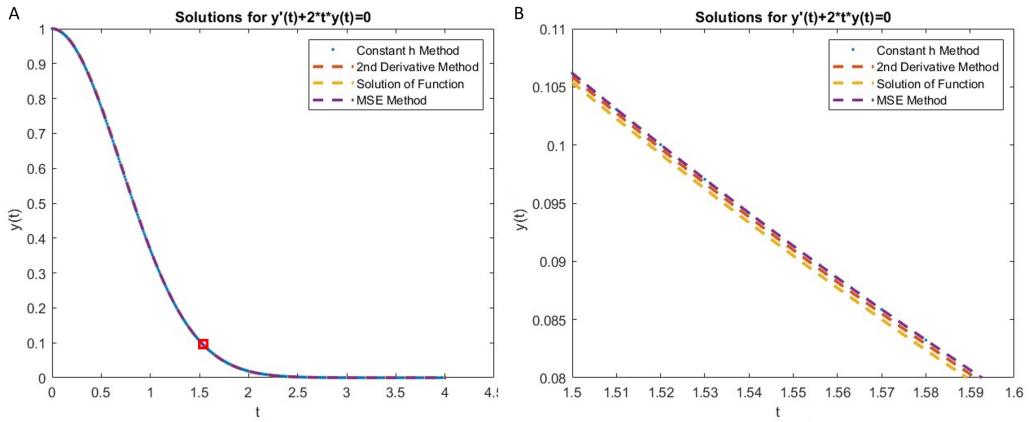

Figure 3.5: Results of the three methods applied on the Function (2.6). Graph A shows the results of the methods and the real solution in the domain t ([0,4]); Graph B is a close-up of the red rectangle to see the difference between the methods.

Table 3.3: **Results of IVP 3.** The number of iterations for solving IVP 3, the average computation time, and the accuracy of each method.

Method	Iterations	Computation time (s)	Accuracy
Constant step size	401	0.535	$1.604 \cdot 10^{-6}$
Second derivative	664	0.539	$2.870 \cdot 10^{-7}$
Mean square error	402	0.555	$1.599 \cdot 10^{-6}$

Figure 3.6 shows the calculation of the MSE for every calculated point. For every point, each method was compared with the analytically calculated value of x at the same time step.

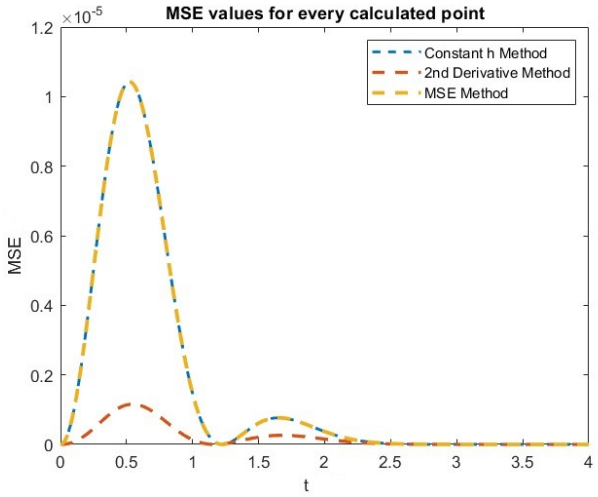

Figure 3.6: Results of the MSE for every step for all methods on IVP Equation (2.6).

4 | Discussion

The aim of this project was to compare the efficiency and accuracy of three different numerical methods for solving initial value problems. An implicit Euler solver using the Newton-Raphson method at each iteration was implemented with a constant step size and it was compared with two adaptive step size algorithms when solving the same set of differential equations. The first step-size algorithm is based on the ratio between the first and second derivative; and the second algorithm is based on minimizing the mean square error. This section is a discussion of the results that were obtained from applying these methods to three test IVPs and analyzing the efficiency and accuracy of the numerical solutions compared to the analytical solutions.

IVP 1

The first test IVP was solved using the three numerical methods and the analytical solution. Table 3.3 and Figure 3.1 show the results. The results of the constant step size and the second the derivative for the first test IVP are almost identical except for the computation time. The reason for this is that the second derivative method uses a constant, maximal step size if the second derivative (which is calculated numerically) is 0. This is the case for this IVP. Therefore the accuracy and the number of iterations are nearly the same, since the maximal step size is the same as the constant step size. Because of a rounding error of the step size of the second derivative, the program additionally does one extra step. The additional computation time is a result of the additional computation of the first and second derivative.

The MSE method shows a higher accuracy as the average MSE is smaller. Additionally, the computation time is lower because of the smaller number of iterations. The reason for the lower number of iteration and higher accuracy is that in the beginning, the adaptation of h to the MSE leads to the almost same step sizes as the constant step size, the step size seems close enough to a minimal error. In the section of the IVP where the ODE shows approximately linear behaviour, the MSE method shows larger step sizes than the constant step size, because they do not lead to a significant increase of the error. Therefore, fewer iterations are needed. The program is not sensitive to the propagation of the error. Therefore a optimal h is found that does not significantly increase the error and does not take into account the overall error from the previous steps.

Figure 3.2 shows that in the beginning, when all methods seem to undershoot the real values, the error increases. After this overshoot, the graph of the analytical solution and the graphs of all methods show a very similar course. Because of the error propagation,

the value of the error for every point in that section is high, but barely increases.

IVP 2

As it can be seen in the results of IVP 2 (Table 3.2 and Figure 3.3), the choice of the method for selecting the step size has a significant influence on the results of the implicit Euler method. The constant step size algorithm was 6 % faster than the second derivative method and it required only 3 % as many iterations. However, the accuracy of the second derivative method was seven times higher. The reason why the accuracy is higher is due to the high number of iterations in the sections of domain where the error of the other two methods increases dramatically, which also leads to a higher propagation of error in the end. In contrast, the MSE method, even though it shows a higher number of iterations, exhibited a lower accuracy than the constant step size method. Additionally, the computation time is seven times higher. This is because, in the beginning of the calculation, the step size leading to a minimal MSE is close in value to the constant step size. Early in the domain, error propagation leads to the situation that no h can be found which leads to the minimal MSE later in the domain. Thus after running the loop for searching the minimum, the program is using a very small h. By using a small step size, especially in the end of the domain where large error occurs, the overall average error increases tremendously leading to less accurate results.

This problem of the MSE method can also be observed in Figure 3.4 as the values of the MSE method at the last section of graph show a lower MSE than the values of the constant step size. But there are more steps taken by the MSE value in this section so the average MSE increases tremendously. The second derivative method shows a smaller error propagation than both of the other methods.

IVP 3

Finally, the results of IVP 3 were analyzed. Table 3.3 shows the results obtained for each method, and Figure 3.5 shows the numerical solutions and the analytical solution. The results show that, compared to the solutions of IVP 1, all methods have a much higher accuracy since the average MSE is low. This difference is probably because at the beginning of the numerical solutions of the first IVP, all the methods show a big discrepancy from the analytical solution. This is not the case for the third IVP. The constant step size and the MSE methods show very similar solutions. The MSE has one step size more which leads to a slightly higher accuracy than the constant step size. This is because the initial step size for finding the optimal step size with a minimal MSE for the MSE method, is the same as the constant step size. Because the MSE is so small anyway, the step size is adjusted only once to see if the maximal tolerance of a error is satisfied and the calculated step size for comparing with the maximal tolerance is taken, which in this case is slightly smaller than the constant step size. Therefore one step more is needed to get to the boundary. The higher computation time is a result of the additional calculation of the minimal MSE with the gradient descent.

Once again, The second derivative method showed higher accuracy and took a larger num-

ber of iterations, especially in the beginning and when the curve of the graph flattens. The step size of the method adjusts as the second derivative increases. This has a significant impact on the accuracy as these sections are highly sensitive to error in the Implicit Euler Method. The computation time is also lower than with the MSE method, which leads to the assumption that the additional calculations needed for the MSE method need more time than of the second derivative method. Even though the second derivative method is more than five times more accurate for this IVP, the question remains if it is necessary if the error in general is very small for all methods. This probably depends on the application.

The Figure 3.6 shows that there is no error propagation as the for three methods the error shows to peaks but flattens in the end. The reason for this the characteristics of the function. The graph shows a change curvature with one turning point. The Implicit Euler shows the systematic error of undershooting the real curvature of a graph [3, 5]. A change of curvature within the course of the graph can lead to a decrease of the error propagation, which is the case for the third IVP.

Analysis and Critique

Overall, the results suggest that the use of adaptive step size algorithms can significantly improve the accuracy of the numerical solutions for certain types of IVPs, while maintaining reasonable computation times. The choice of adaptive step size algorithm may depend on the particular characteristics of the problem being solved, such as the degree of nonlinearity or the presence of sharp changes in the solution.

It should also be noted that the results presented here are based on a small set of test problems, and further analysis may be required to generalize these findings to a wider range of problems. Additionally, the choice of numerical method and algorithm may depend on other factors such as the availability of parallel computing resources or the desired level of precision.

Newton–Raphson Method

The use of the Newton-Raphson Method to find the next step in the the Implicit Euler Method is a common procedure, but can lead to some problems. For finding the root, the derivative of the function is needed. For this the ODE needs to be homogeneous, reducible to a first-order system, and differentiable [11]. In the implementation, the derivative was calculated numerically, which could have lead to an additional error in every calculation for the MSE. By using the numerical calculation of the first derivative the computation time can be reduced.

For using the Newton-Raphson Method, the first step needs a guess of the root. If the guess is too far from the real root, the method does not converge or overshoots, which can lead to not finding the root. To avoid this in the program, the next guess for the root is assumed to be as far away as the value of the last step of the Implicit Euler Method from the step before. In processing the test function, this method could have been a reason

the Newton-Raphson Methods always converged. To be certain if this method is efficient, it would be important to test it on more functions. Additionally, if no root is found, the program will set the value of the root to 0, which can lead to some problems of converging of the Newton-Raphson-Method for the next steps of the Implicit Euler Method as well, as the guess of the next root is based on the difference of the last step to the step before.

MSE Algorithm

The method of finding a minima by means of gradient descent was used in this program. The usage of gradient descent is common and known for its stability and high probability of convergence. [8] However, there are some other possibilities of finding a minima e.g. applying the Newton-Raphson method on the first and second derivative of the function to find the minimum of it. As the program already included the Newton-Raphson method, it would have been faster to use it also for finding the minimum. However, using gradient descent was ensuring a higher success rate since it enables specifically finding a minimum when applying the Newton-Raphson method, rather than a maximum *or* minimum. Additionally, only the first derivative had to be calculated numerically by the program for gradient descent, and not the first and second derivatives, which also reduces a possible error, because of the numerical assumption.

The approach of adapting step size by minimizing MSE using gradient descent is not a particularly effective one since the actual solution is required for comparison. A numerical solver that is limited to solving differential equations that already have known analytical solutions, does not have much value as a solver of continuous functions. However, this method *can* be effective when solving discrete systems, especially for solving some types of inverse problems, including those that are ill-posed or involve noisy data [6]. The basic idea is to formulate the inverse problem as an optimization problem, where the objective function is the MSE between the measured data and the predicted data based on a given set of model parameters. The goal is then to find the set of model parameters that minimizes this objective function [9].

One advantage of this approach is that it allows for the incorporation of prior information or constraints on the model parameters, such as smoothness or sparsity, by adding regularization terms to the objective function [4]. Another advantage is that it can be implemented using standard optimization techniques, such as the gradient descent method used in this project, which can be computationally efficient and scalable to large-scale problems [10].

However, there are also some limitations and challenges associated with this approach. One limitation is that the performance of the algorithm can be sensitive to the choice of optimization parameters, such as the learning rate and the stopping criteria [9]. In addition, the algorithm may converge to a local minimum rather than the global minimum, depending on the initial conditions and the topology of the objective function [4]. To mitigate these issues, it may be necessary to use adaptive learning rates or more sophisticated optimization methods, such as stochastic gradient descent or second-order methods [6].

Furthermore, the success of this approach is highly dependent on the quality and accuracy of the measured data, as well as the assumptions and simplifications made in the model [10]. In particular, if the model is too simple or does not capture all the relevant physics or interactions, the algorithm may produce biased or inaccurate estimates of the model parameters [4]. In such cases, it may be necessary to use more complex models or incorporate additional data or prior information to improve the accuracy and robustness of the algorithm.

Second Derivative Algorithm

Overall, the algorithm based on the second derivative is a more precise solver for the test IVPs. Since functions with a change of curvature seem to lead to errors in the Implicit Euler Method, the second derivative method explicitly tries to decrease step size in these areas, as it adjusts the steps size based on the ratio between first and second derivative. However, the second derivative method has some limitations and disadvantages as well. Like the MSE method, the second derivative method is only applicable to a specific class of differential equations, namely those which are homogeneous, can be reduced to a first-order system, have a second derivative that does not equal 0, and are differentiable [11]. This means that the method cannot be used to solve other types of differential equations, such as non-homogeneous equations. It also means the method cannot be applied to discrete systems. The second derivative can be difficult to calculate, especially for complex functions. If the function is very complicated, finding the second derivative analytically may be difficult or impossible. In such cases, numerical methods may be required, which can be time-consuming and computationally intensive. This method also requires the specification of the initial conditions, which can be challenging in some cases, especially if the function is not well-understood or if there is noise in the data [11]. In some cases, multiple sets of initial conditions may need to be tried before a solution can be found.

The program written to solve the first and second derivative uses a numerical solution, These lead to an additional error, but also to a reduced computational time.

Difficulties and Complications

Significant challenge was encountered when attempting to modify code that was designed to solve first order ordinary differential equations (ODEs) into code that could solve higher order ODEs. This proved to be a formidable task that required a great deal of time and effort from both members of the group.

Dozens of work hours were spent trying to adapt the working code to be able to solve the test IVPs that had been chosen. Different methods of solving higher order ODEs were researched and implementation into the existing codebase was attempted, but unfortunately met with limited success.

One of the main challenges was in understanding the mathematical nuances of the nu-

merical methods. The basic principles of the algorithm were graspable, but the specific details proved to be more difficult to comprehend and time-consuming to learn. A great deal of time was spent pouring over the code line by line, trying to implement the necessary changes without disrupting the logic and flow of the program. Unfortunately, most attempts only resulted in a cascade of errors and additional complications.

Despite best efforts, ultimately, the goal of creating a more robust and user-friendly program that could solve higher order ODEs was not achieved. As a result, different test IVPs had to be chosen for comparison. While this was a disappointment, the project was still successfully completed and meaningful results were generated. Moving forward, the authors will continue to explore different methods and approaches to solving ODEs, in the hopes of improving upon this work.

5 | Conclusion

In conclusion, this study has shown that adaptive step size algorithms can offer significant advantages over the traditional constant step size method for numerical simulations. Specifically, the algorithm based on the second derivative demonstrated superior accuracy in solving the IVPs when compared to the constant step size algorithm. This improvement in accuracy was achieved with only a marginal increase in computation time. The algorithm based on minimizing the mean square error also performed well, though not as well as the second derivative algorithm in some cases.

One of the key strengths of adaptive step size algorithms is their ability to adjust the step size according to the local behavior of the solution, resulting in a more efficient and accurate simulation. However, the effectiveness of these algorithms may depend on the specific characteristics of the problem being solved. For example, in problems with steep or rapidly varying solutions, a smaller step size might be necessary to accurately capture the solution behavior, while in problems with smooth solutions, a larger step size might be more appropriate. Therefore, the choice of adaptive step size algorithm, and the specific parameters used, should be tailored to the problem at hand.

Another strength of adaptive step size algorithms is their ability to handle stiff differential equations. In contrast, constant step size methods can struggle with stiff equations, requiring a very small step size that results in a large number of iterations and increased computation time. Adaptive step size algorithms can adjust the step size to better capture the behavior of the solution, resulting in a more efficient simulation.

One potential weakness of adaptive step size algorithms is their increased computational complexity compared to constant step size methods. This is because the adaptive algorithms require additional calculations to determine the optimal step size. However, these results show that this increased computational complexity can be offset by the superior accuracy of the adaptive algorithms, making them a viable option for many applications.

In summary, this study demonstrates the advantages of using adaptive step size algorithms for numerical simulations, particularly for solving stiff differential equations or problems with rapidly varying solutions. The accuracy of the implicit Euler solver depends on the step size used in the calculation. If the step size is too large, the accuracy of the solution may be compromised. Additionally, the method may not be able to capture certain features of the solution, such as oscillations or sharp peaks, which can limit its usefulness in some applications. The choice of specific algorithm and parameters will depend

on the characteristics of the problem being solved, and careful consideration should be given to balancing the computational cost with the desired level of accuracy. While these adaptive step size algorithms can be powerful tools for solving certain types of differential equations, they do have limitations and may not be the best choice for all problems. As with any numerical method, it is important to carefully consider the strengths and weaknesses of an approach before applying it to a particular problem.

Bibliography

[1] Chapra, S. C. and Canale, R. P. (2015). *Numerical Methods for Engineers*. McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121, 7 edition.

[2] Feldman, J. (1999). *Variable Step Size Methods*. https://personal.math.ubc.ca/ CLP/. Retrieved from https://personal.math.ubc.ca/∼feldman/math/vble.pdf.

[3] Fjordholm, U. S. (2018). Numerical methods for ordinary differential equations. Technical report.

[4] Gardner, W. A. (1984). Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique. *Signal Processing*, 2(6). doi:10.1016/0165-1684(84)90013-6.

[5] Hairer, E. and Lubich, C. (2010). Numerical solution of ordinary differential equations.

[6] Mustapha, A., Mohamed, L., and Ali, K. (2020). An overview of gradient descent algorithm optimization in machine learning: Application in the ophthalmology field. *In Communications in Computer and Information Science*, pages 349–359.

[7] Peterson, J. (2017). *Numerical Methods for Differential Equations*. Department of Scientific Computing, Florida State University, Tallahassee, FL 32304, United States.

[8] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[9] Skouras, K., Goutis, C., and Bramson, M. J. (1994). Estimation in linear models using gradient descent with early stopping. *Statistics and Computing*, 4(4):271–278. doi:10.1007/bf00156750.

[10] Turin, A. (2020). Gradient descent from scratch.

[11] Zhang, L., Zhang, C., and Liu, M. (2020). Variable-step-size second-order-derivative multistep method for solving first-order ordinary differential equations in system simulation. *Advances in Complex Systems. A Multidisciplinary Journal*, 11(01). doi:10.1142/s1793962320500014.

A | Appendix A

Code

Implicit Euler with Constant Step Size

```
%==========================================================================
% PROJECT FOR NMoS: INVESTIGATION OF THE INFLUENCE OF AN ADJUSTABLE
%                   STEP-SIZE IN THE BACKWARD EULER METHOD
%--------------------------------------------------------------------------
%Three methods of adjusting h are compared in their computational time
%accuracy and stability:
%   1. constant h
%   2. using the second derivative
%   3. adjusting the h in a way to minimize the MSE by using gradient
%       descent
%
%To test these methods functions with a known solution are tested.
%--------------------------------------------------------------------------
%In this file the Implicit Euler Method (IEM) is done with the constant h
%==========================================================================


%--------------------------------------------------------------------------
% PUT IN INITIAL VALUES AND BOUNDARIES
%--------------------------------------------------------------------------
tic
clear all;
x_start = 1;
t_start = 0;
t_end = 4;
const_h = 0.01;
step_newt = 10000;

%--------------------------------------------------------------------------
% START SOLVING WITH CONSTANT H
%--------------------------------------------------------------------------

t_array = t_start:const_h:t_end; %setting the whole time array
n = numel(t_array); %finding the number for the loop for finding x values
x_sol(1) = x_start; %solution of x with the newton method
x_guess(1)= x_sol(1)+const_h;%guess of x for the newton method

for i=1:n-1 %LOOP FOR EULER STEPS
   %solution of x with the newton method
    x_sol(i+1) = newton(@dx_dt,x_sol(i),x_guess(i),t_array(i)+const_h, ...
        const_h,step_newt);
   %guess of x for the newton method
    x_guess(i+1) = x_sol(i+1)+(x_sol(i+1)-x_sol(i));
end %LOOP FOR EULER STEPS

%--------------------------------------------------------------------------
% REAL VALUES OF X AND MSE
%--------------------------------------------------------------------------

for i=1:n-1 %LOOP FOR REAL VALUES
    x_true(1) = x_start; %x_true = solution array of x
    mse_value(1) = 0;
    x_true(i+1) = dx_dt_sol(t_array(i+1),x_start);
    mse_value(i+1) = MSE(x_sol(i+1),x_true(i+1));
end %LOOP FOR REAL VALUE

mse_average = sum(mse_value)/numel(t_array);

%--------------------------------------------------------------------------
% Visualization of the Solution
%--------------------------------------------------------------------------
tiledlayout(1,2)
nexttile
plot(t_array,x_sol,"--",'Color',[0, 0.5, 0.3],'LineWidth',1);
hold on
```

```
plot(t_array,x_true,"--",'Color',[0.5, 0, 0.3],'LineWidth',1);
xlabel("t");
ylabel("x");
nexttile
plot(t_array,mse_value,"--",'Color',[0.3, 0, 0.5],'LineWidth',1)
toc



%==========================================================================
% FUNCTIONS
%==========================================================================

%--------------------------------------------------------------------------
% FUNCTION TO SOLVE AND FUNCTION FOR REAL SOLUTION (CHANGE THESE FOR YOUR
% PURPOSE)
%--------------------------------------------------------------------------

function func = dx_dt(t,x)
%Three test functions (ODE)
  %func = 8*t-4*x;
  %func = x^2*cos(t);
   func = -2*t*x;

end

function f_solution = dx_dt_sol(t,x_0)
%Three solutions functions (ODE)
    %f_solution = 2*t+x_0-t*exp(-4*t);
    %f_solution = 1/(1-sin(t));
    f_solution = exp(-t^2);
end


%--------------------------------------------------------------------------
% FUNCTION FOR NEWTON METHOD
%--------------------------------------------------------------------------
function x_newt = newton(dx_dt,x_i, x_guess, t, h,step_newt)
  prime_factor = 0.00000001; %For getting the derivative
  error = 0.00000001; %Tolerance for finding the root

  %First newton step
  j=1;
  x_search(j) = x_guess; %x_search = array of search for right x
  newton =  x_i + h * dx_dt(t,x_search(j))- x_guess;
  primestep = (x_i + h * dx_dt(t,x_search(j)+prime_factor) - ...
      (x_search(j)+prime_factor));
  prime = (primestep-(newton))/prime_factor;
  j = 2;
  x_search(j) = x_search(j-1)-newton/prime;

  %LOOP FOR NEWTON STEPS
  while (abs(x_search(j)-x_search(j-1))>error) & j ≤ step_newt
    newton =  x_i + h * dx_dt(t,x_search(j)) - x_search(j);
    prime = ((x_i + h * dx_dt(t,x_search(j)+prime_factor) - ...
        (x_search(j)+prime_factor))-(newton))/prime_factor;
    x_search(j+1)=x_search(j)-newton/prime;
    j=j+1;
  end%LOOP FOR NEWTON STEPS
  %IF STATEMENT FOR NEWTONSTEPS (if not successful)
  if abs(x_search(j)-x_search(j-1))>error
      x_newt  = 0;
  else
      x_newt  = x_search(j);
  end%IF STATEMENT FOR NEWTONSTEPS

end

%--------------------------------------------------------------------------
% FUNCTION FOR MEAN SQUARE ERROR
%--------------------------------------------------------------------------

function MSE = MSE(x_estimate,x_true)
    MSE = (x_estimate-x_true)^2;
end
```

Implicit Euler with Step Size Based on Second Derivative

```
%==========================================================================
% PROJECT FOR NMoS: INVESTIGATION OF THE INFLUENCE OF AN ADJUSTABLE
%                   STEP-SIZE IN THE BACKWARD EULER METHOD
%--------------------------------------------------------------------------
%Three methods of adjusting h are compared in their computational time
%accuracy and stability:
%   1. constant h
%   2. using the second derivative
%   3. adjusting the h in a way to minimize the MSE by using gradient
%       descent
%
%To test these methods functions with a known solution are tested.
%--------------------------------------------------------------------------
%In this file the Implicit Euler MEthod (IEM) is done with second
%derivative
%
%The idea is that if the first derivative is small comapared to the second
%derivative the graph goes near a minima or maxima and therefore a
%djustment of h is needed to reduce the error and increase stability
%==========================================================================


%--------------------------------------------------------------------------
% PUT IN INITIAL VALUES, FUNCTION AND BOUNDARIES
%--------------------------------------------------------------------------
tic
clear all;
x_start = 1;
t_start = 0;
t_end = 4;
h_max = 0.01;
step_newt = 10000;


%--------------------------------------------------------------------------
% START SOLVING WITH CONSTANT H
%--------------------------------------------------------------------------
t_array(1) = t_start;
i = 1;
x_sol(1) = x_start;
x_guess(1)= x_sol(1)+h_max;%guess of x for the newton method
while t_array(i)<t_end %LOOP FOR EULER STEPS


    f_prime = dx_dt_prime(@dx_dt,t_array(i),x_sol(i));
    f1(i)=f_prime;
    f_2prime = (dx_dt_prime(@dx_dt,t_array(i)+0.001,x_sol(i)+0.001)- ...
        dx_dt_prime(@dx_dt,t_array(i),x_sol(i)))/0.1;
    f2(i)=f_2prime;
    if f_2prime==0 || f_prime == 0
        der2_h(i)=h_max;
    else
       der2_h(i) = abs(f_prime/f_2prime)*0.0001;
    end

    if der2_h > h_max
        der2_h(i)=h_max;
    else
        der2_h(i) = abs(f_prime/f_2prime)*0.0001;
    end

    x_sol(i+1) = newton(@dx_dt,x_sol(i),x_guess(i),t_array(i)+der2_h(i), ...
        der2_h(i),step_newt);
    t_array(i+1)=t_array(i)+der2_h(i);
    x_guess(i+1) = x_sol(i+1)+(x_sol(i+1)-x_sol(i));%guess of x for the newton method
    i = i+1;
end %LOOP FOR EULER STEPS


%--------------------------------------------------------------------------
% REAL VALUES OF X AND MSE
%--------------------------------------------------------------------------

for j=1:i-1 %LOOP FOR REAL VALUES
    x_true(1) = x_start; %solution of x with the newton method
    mse_value(1) = 0;
    x_true(j+1) = dx_dt_sol(t_array(j+1),x_start);
    mse_value(j+1) = MSE(x_sol(j+1),x_true(j+1));
end %LOOP FOR REAL VALUE

mse_average = sum(mse_value)/numel(t_array);
%--------------------------------------------------------------------------
% Visualisation of the Solution
%--------------------------------------------------------------------------
tiledlayout(1,2)
nexttile
plot(t_array,x_sol,"--",'Color',[0, 0.5, 0.3],'LineWidth',1);
hold on
plot(t_array,x_true,"--",'Color',[0.5, 0, 0.3],'LineWidth',1);
```

```
xlabel("t");
ylabel("x");
nexttile
plot(t_array,mse_value,"--",'Color',[0.3, 0, 0.5],'LineWidth',1);

toc



%==========================================================================
% FUNCTIONS
%==========================================================================

%--------------------------------------------------------------------------
% FUNCTION TO SOLVE AND FUNCTION FOR REAL SOLUTION (CHANGE THESE FOR YOUR
% PuRPOSE)
%--------------------------------------------------------------------------
function func = dx_dt(t,x)
  %func = 8*t-4*x;
  % func = x^2*cos(t);
   func = -2*t*x;

end

function f_solution = dx_dt_sol(t,x_0)
    %f_solution = 2*t+x_0-t*exp(-4*t);
    %f_solution = 1/(1-sin(t));
    f_solution = exp(-t^2);
end

%--------------------------------------------------------------------------
% FUNCTION FOR GETTING DERIVATIVE NUMERICALLY
%--------------------------------------------------------------------------

function func_prime = dx_dt_prime(funct,t,x)
prime_step = 0.0001;
func_prime = (funct(t+prime_step,x+prime_step)- funct(t,x))+prime_step;

end

%--------------------------------------------------------------------------
% FUNCTION FOR NEWTON METHOD
%--------------------------------------------------------------------------
function x_newt = newton(dx_dt,x_i, x_guess, t, h,step_newt)
  prime_factor = 0.00000001; %For getting the derivative
  error = 0.00000001; %Tolerance for finding the root

  %First newton step
  j=1;
  x_search(j) = x_guess; %x_search = array of search for right x
  newton =  x_i + h * dx_dt(t,x_search(j))- x_guess;
  primestep = (x_i + h * dx_dt(t,x_search(j)+prime_factor) - ...
      (x_search(j)+prime_factor));
  prime = (primestep-(newton))/prime_factor;
  j = 2;
  x_search(j) = x_search(j-1)-newton/prime;

  %LOOP FOR NEWTON STEPS
  while (abs(x_search(j)-x_search(j-1))>error) & j ≤ step_newt
    newton =  x_i + h * dx_dt(t,x_search(j)) - x_search(j);
    prime = ((x_i + h * dx_dt(t,x_search(j)+prime_factor) - ...
        (x_search(j)+prime_factor))-(newton))/prime_factor;
    x_search(j+1)=x_search(j)-newton/prime;
    j=j+1;
  end%LOOP FOR NEWTON STEPS
  %IF STATEMENT FOR NEWTONSTEPS (if not successful)
  if abs(x_search(j)-x_search(j-1))>error
      x_newt  = 0;
  else
      x_newt  = x_search(j);
  end%IF STATEMENT FOR NEWTONSTEPS

end


%--------------------------------------------------------------------------
% FUNCTION FOR MEAN SQUARE ERROR
%--------------------------------------------------------------------------

function MSE = MSE(x_estimate,x_true)
    MSE = (x_estimate-x_true)^2;
end
```

Implicit Euler with Step Size Based on MSE

```
%==========================================================================
% PROJECT FOR NMoS: INVESTIGATION OF THE INFLUENCE OF AN ADJUSTABLE
%                   STEP-SIZE IN THE BACKWARD EULER METHOD
%--------------------------------------------------------------------------
%Three methods of adjusting h are compared in their computational time
%accuracy and stability:
%   1. constant h
%   2. using the second derivative
%   3. adjusting the h in a way to minimize the MSE by using gradient
%       descent
%
%To test these methods functions with a known solution are tested.
%--------------------------------------------------------------------------
%In this file the Implicit Euler MEthod (IEM) is done with adjusting the h
% in a way to minimize the MSE by using gradient descent
%==========================================================================


%--------------------------------------------------------------------------
% PUT IN INITIAL VALUES, FUNCTION AND BOUNDARIES
%--------------------------------------------------------------------------
tic
clear all;
x_start = 1;
t_start = 0;
t_end = 4;
h_start = 0.01;
step_newt = 10000;

%--------------------------------------------------------------------------
% START SOLVING WITH ADJUSTMENT BY MINIMIZING MSE
%--------------------------------------------------------------------------
t_array(1) = t_start;
i = 1;
h(1) = h_start;
x_sol(1) = x_start;
while t_array(i)<t_end %LOOP FOR EULER STEPS

    h_new = gradient_descent(@MSE,h_start,x_sol(i),t_array(i),@dx_dt_sol, ...
        x_start,step_newt);
    h(i+1)=h_new;
    x_sol(i+1)=newton(@dx_dt,x_sol(i),x_sol(i)+0.001,t_array(i)+h_new, ...
        h_new,step_newt);
    t_array(i+1)=t_array(i)+h(i+1);
    i = i+1;

end %LOOP FOR EULER STEPS

%--------------------------------------------------------------------------
% REAL VALUES OF X AND MSE
%--------------------------------------------------------------------------

for j=1:i-1 %LOOP FOR REAL VALUES
    x_true(1) = x_start;
    mse_value(1) = 0;
    x_true(j+1) = dx_dt_sol(t_array(j+1),x_start);
    mse_value(j+1) = MSE(x_sol(j+1),x_true(j+1));
end %LOOP FOR REAL VALUE


mse_average = sum(mse_value)/numel(t_array);
%--------------------------------------------------------------------------
% Visualisation of the Solution
%--------------------------------------------------------------------------
tiledlayout(1,2)
nexttile
plot(t_array,x_sol,"--",'Color',[0, 0.5, 0.3],'LineWidth',1);
hold on
plot(t_array,x_true,"--",'Color',[0.5, 0, 0.3],'LineWidth',1);
xlabel("t");
ylabel("x");
nexttile
plot(t_array,mse_value,"--",'Color',[0.3, 0, 0.5],'LineWidth',1);

toc



%==========================================================================
% FUNCTIONS
%==========================================================================

%--------------------------------------------------------------------------
% FUNCTION TO SOLVE AND FUNCTION FOR REAL SOLUTION (CHANGE THESE FOR YOUR
% PuRPOSE)
%--------------------------------------------------------------------------
function func = dx_dt(t,x)
  func = 8*t-4*x;
  % func = x^2*cos(t);
```

```
90      %func = -2*t*x;
91
92  end
93
94  function f_solution = dx_dt_sol(t,x_0)
95      f_solution = 2*t+x_0-t*exp(-4*t);
96      %f_solution = 1/(1-sin(t));
97      %f_solution = exp(-t^2);
98  end
99
100
101 %------------------------------------------------------------------------
102 % FUNCTION FOR GRADIENT DESCENT
103 %------------------------------------------------------------------------
104
105 function h_right = gradient_descent(funct,h_start,x_sol,t,x_true,x_start,step_newt)
106     min(1)= h_start;
107     prime_factor = 0.00001;
108     k = 2;
109     alpha = 0.01;
110     error = 0.001;
111     f_prime = (funct(newton(@dx_dt,x_sol,x_sol+0.001,t+min(1)+prime_factor,min(1)+ ...
112         prime_factor,step_newt),x_true(t+min(1),x_start))- ...
113         funct(newton(@dx_dt,x_sol,x_sol+0.001,t+min(1),min(1),step_newt), ...
114         x_true(t+min(1),x_start)))/0.001;
115     min(k) = min(1) - alpha * f_prime;
116     while abs(min(k-1)-min(k))>error & k <= 10000
117         f_prime = (funct(newton(@dx_dt,x_sol,x_sol+0.001,t+min(k)+prime_factor, ...
118             min(k)+prime_factor,step_newt),x_true(t+min(k),x_start))- ...
119             funct(newton(@dx_dt,x_sol,x_sol+0.001,t+min(k),min(k),step_newt), ...
120             x_true(t+min(k),x_start)))/prime_factor;
121         min(k+1) = min(k) - alpha * f_prime;
122         k=k+1;
123     end
124     if min(k)>0 & abs(min(k-1)-min(k))<error
125         h_right=min(k);
126     else
127         h_right=0.0001;
128     end
129
130 end
131
132 %------------------------------------------------------------------------
133 % FUNCTION FOR NEWTON METHOD
134 %------------------------------------------------------------------------
135 function x_newt = newton(dx_dt,x_i, x_guess, t, h,step_newt)
136   prime_factor = 0.00000001; %For getting the derivative
137   error = 0.00000001; %Tolerance for finding the root
138
139   %First newton step
140   j=1;
141   x_search(j) = x_guess; %x_search = array of search for right x
142   newton =  x_i + h * dx_dt(t,x_search(j))- x_guess;
143   primestep = (x_i + h * dx_dt(t,x_search(j)+prime_factor) - ...
144       (x_search(j)+prime_factor));
145   prime = (primestep-(newton))/prime_factor;
146   j = 2;
147   x_search(j) = x_search(j-1)-newton/prime;
148
149   %LOOP FOR NEWTON STEPS
150   while (abs(x_search(j)-x_search(j-1))>error) & j <= step_newt
151     newton =  x_i + h * dx_dt(t,x_search(j)) - x_search(j);
152     prime = ((x_i + h * dx_dt(t,x_search(j)+prime_factor) - ...
153         (x_search(j)+prime_factor))-(newton))/prime_factor;
154     x_search(j+1)=x_search(j)-newton/prime;
155     j=j+1;
156   end%LOOP FOR NEWTON STEPS
157   %IF STATEMENT FOR NEWTONSTEPS (if not successful)
158   if abs(x_search(j)-x_search(j-1))>error
159       x_newt  = 0;
160   else
161       x_newt  = x_search(j);
162   end%IF STATEMENT FOR NEWTONSTEPS
163
164 end
165
166 %------------------------------------------------------------------------
167 % FUNCTION FOR MEAN SQUARE ERROR
168 %------------------------------------------------------------------------
169
170 function MSE = MSE(x_estimate,x_true)
171     MSE = (x_estimate-x_true)^2;
172 end
```

Analytical Solutions of Test IVPs

IVP1

```
% Define the analytical solution
y_exact = @(t) 2*t - 1/2 + 3*exp(-4*t)/2;

% Set up the time range for the plot
t = linspace(0,4,200);

% Compute the analytical solution at the time steps used by the ODE solver
y_exact_vals = y_exact(t);

% Plot the analytical solution
figure
plot(t, y_exact_vals, '-.', 'Color', [0 0.45 0.74], 'LineWidth', 1, 'Marker', '.', ...
    'MarkerSize', 1, 'MarkerEdgeColor', [0.85 0.33 0.1], 'MarkerFaceColor', [0.85 0.33 0.1]);
title('Analytical solution to  y  (t) = -4y(t) + 8t, y(0) = 1');
xlabel('t', 'FontSize', 12);
ylabel('y(t)', 'FontSize', 12);
axis([0 4 0 8]);
```

IVP2

```
% Define the analytical solution
y_exact = @(t) 1./(1-sin(t));

% Set up the time range for the plot
t = linspace(0,1,100);

% Compute the analytical solution at the time steps used by the ODE solver
y_exact_vals = y_exact(t);

% Plot the analytical solution
figure
plot(t, y_exact_vals, '-.', 'Color', [0 0.45 0.74], 'LineWidth', 1, 'Marker', '.', ...
    'MarkerSize', 1, 'MarkerEdgeColor', [0.85 0.33 0.1], 'MarkerFaceColor', [0.85 0.33 0.1]);
title('Analytical solution to  y  (t) = cos(t)*y(t)^2, y(0) = 1');
xlabel('t', 'FontSize', 12);
ylabel('y(t)', 'FontSize', 12);
axis([0 1 0 7]);
```

IVP3

```
% Define the analytical solution
y_exact = @(t) exp(-t.^2);

% Set up the time range for the plot
t = linspace(0,4,200);

% Compute the analytical solution at the time steps used by the ODE solver
y_exact_vals = y_exact(t);

% Plot the analytical solution
figure
plot(t, y_exact_vals, '-.', 'Color', [0 0.45 0.74], 'LineWidth', 1, 'Marker', '.', ...
    'MarkerSize', 1, 'MarkerEdgeColor', [0.85 0.33 0.1], 'MarkerFaceColor', [0.85 0.33 0.1]);
title('Analytical solution to  y  (t) = -2t    y(t), y(0) = 1');
xlabel('t', 'FontSize', 12);
ylabel('y(t)', 'FontSize', 12);
axis([0 4 -0.2 1.2]);
```

Organizational Documents

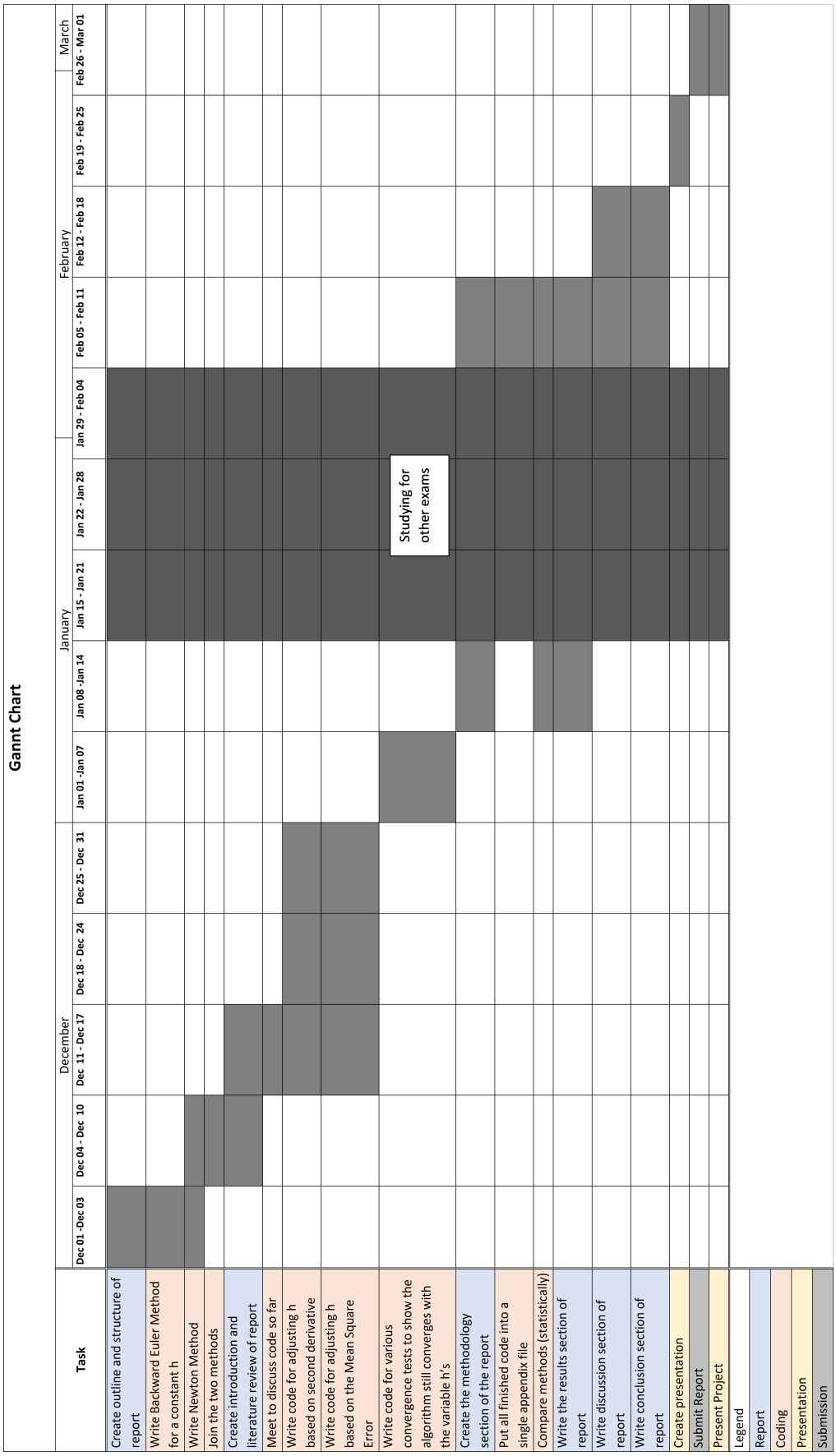

Figure A.1: Organizational plan for project

List of Figures

2.1 Flowchart for the implicit Euler method with a constant step size. 5
2.2 Flowchart for the implicit Euler method with step size based on the second derivative. 6
2.3 Flowchart for the implicit Euler method with step size based on minimizing MSE. 8
2.4 Flowchart for the Newton-Raphson Method program. 9
2.5 Analytical solution of (2.4) 11
2.6 Analytical solution of (2.5) 12
2.7 Analytical solution of (2.6) 13

3.1 Results of the three methods applied on the Function (2.4). Graph A shows the results of the methods and the real solution in the domain t ([0,4]); Graph B is a close-up of the red rectangle to see the difference between the methods. 17
3.2 Results of the MSE for every step for all methods on Equation (2.4). . . . 18
3.3 Results of the three methods applied on the Function (2.6). Graph A shows the results of the methods and the real solution within the boundary of t ([0,1]); Graph B is a close up of the red rectangle to see the difference between the methods. 18
3.4 Results of the MSE for every step for all methods on Equation (2.5). . . . 19
3.5 Results of the three methods applied on the Function (2.6). Graph A shows the results of the methods and the real solution in the domain t ([0,4]); Graph B is a close-up of the red rectangle to see the difference between the methods. 19
3.6 Results of the MSE for every step for all methods on IVP Equation (2.6). 20

A.1 Organizational plan for project 37

List of Tables

3.1 Results of IVP 1. 18
3.2 Results of IVP 2. 19
3.3 Results of IVP 3. 20

Final Project Journal

By: Aaron Hinkle 22113

For: 3301 Numerical Methods of Simulation

Submitted: 1 March 2023

December 2, 2022:

Created the structure of the Project Journal and started thinking about the tasks and time that will be required to accomplish the project.

December 3, 2022:

Started a Gannt Chart so we can get and stay organized.

- Create outline and structure of report (1 week)
- Write Backward Euler Method for a constant h (1 week)
- Write Newton Method (1 week)
- Join the two methods (1 week)
- Create introduction and literature review of report (2 weeks)
- Meet to discuss code (1 day)
- Write code for adjusting h based on second derivative (3 weeks)
- Write code for adjusting h based on the Mean Square Error (3 weeks)
- Write code for various convergence tests to show the algorithm still converges with the variable h's (1 week)
- Create the methodology section of the report (2 weeks)
- Put all finished code into a single appendix file (1 week)
- Compare methods (statistically) (2 weeks)
- Write the results section of report (2 weeks)
- Write discussion section of report (2 weeks)
- Write conclusion section of report (2 weeks)
- Create presentation (1 week)
- Submit Report (1 day)
- Present Project (1 day)

December 5, 2022:

Finished the Gannt Chart and sent it to Sophia. She approves overall and we are off and running. We will spend this next week reviewing, reading, and collecting our thoughts and ideas.

December 8, 2022:

I spent the day refining code from the exercises for the implicit solver using Newton's Method. Mostly just MATLAB practice since I am rusty.
Calculate a derivative
https://de.mathworks.com/matlabcentral/answers/30313-how-to-calculate-a-derivative
Symbolic Math Toolbox
https://de.mathworks.com/matlabcentral/answers/445457-how-to-install-and-enable-symbolic-math-toolbox
Installing the add-on
https://de.mathworks.com/matlabcentral/answers/101885-how-do-i-install-additional-toolboxes-into-an-existing-installation-of-matlab
Reinstall Matlab with the add-on
https://de.mathworks.com/matlabcentral/answers/98886-how-do-i-install-matlab-or-other-mathworks-products

December 9, 2022:

Started writing the Step Size section of the Introduction chapter of the report.
**https://mmsallaboutmetallurgy.com/wp-content/uploads/2019/01/numerical-methods.pdf
**https://people.sc.fsu.edu/~jpeterson/nde_book4.pdf
**https://personal.math.ubc.ca/~CLP/CLP2/clp_2_ic/ap_variable.html

December 11, 2022:

Finished the Step Size section of the Introduction chapter of the report.
**https://mmsallaboutmetallurgy.com/wp-content/uploads/2019/01/numerical-methods.pdf
**https://people.sc.fsu.edu/~jpeterson/nde_book4.pdf
**https://personal.math.ubc.ca/~CLP/CLP2/clp_2_ic/ap_variable.html

December 12, 2022:

Professor Struck wants us to compare our methods to some other adaptive step size methods, so I spent the day researching. This is a huge topic and I only understand things at the surface level so I started with the textbooks today. Hopefully I will find a few options before meeting with Sophia on Wednesday.
**https://mmsallaboutmetallurgy.com/wp-content/uploads/2019/01/numerical-methods.pdf
**https://people.sc.fsu.edu/~jpeterson/nde_book4.pdf
**https://personal.math.ubc.ca/~CLP/CLP2/clp_2_ic/ap_variable.html

December 13, 2022:

I decided on the adaptive step size control method and the predictor-corrector method. Both are commonly used and I already understand the basics of one of them so that's a plus. Today I just made some notes about how they work so I can talk to Sophia about them tomorrow.

December 14, 2022:

Met with Sophia to discuss the project so far. For the early stages, we will essentially split the labor between the report and the coding. I will start doing the literature review and she will start programming. We were initially going to use Struck's solution for the base algorithm with a constant step size, but she thinks it would be better if we made our own, and it will make coding the other two algorithms easier, which are good points. So, she is going to write all three algorithms from scratch.

December 17, 2022:

Wrote intro section about adaptive step size control method.
https://en.wikipedia.org/wiki/Adaptive_step_size
**https://people.sc.fsu.edu/~jpeterson/nde_book4.pdf
**https://personal.math.ubc.ca/~feldman/math/vble.pdf

December 18, 2022:

Wrote intro section about predictor corrector method.
https://en.wikipedia.org/wiki/Adaptive_step_size
**https://people.sc.fsu.edu/~jpeterson/nde_book4.pdf
**https://personal.math.ubc.ca/~feldman/math/vble.pdf

December 21, 2022:

Sophia wants to do the report in Latex. I don't prefer it because I'm not great at it, so this is a good opportunity to improve my skills. I went through dozens of templates looking for something decent and I eventually found one to use:
https://www.overleaf.com/latex/templates/politecnico-di-milano-deib-phd-thesis-template/ydsvtyzwxfdk
Some of the places I looked:
https://www.overleaf.com/gallery/tagged/report
https://www.latextemplates.com/cat/academic-journals
https://en.wikibooks.org/wiki/LaTeX/Scientific_Reports
https://scientificallysound.org/2019/02/19/1396/

December 22, 2022:

Started transferring work so far into Latex
https://tex.stackexchange.com/questions/52276/inline-equation-in-latex-with-text
https://tex.stackexchange.com/questions/111580/removing-an-unwanted-page-between-two-chapters

December 23, 2022 - December 26, 2022:

Christmas

December 27, 2022:

Still converting the stuff I already did from Word to Latex. Latex is a little annoying (tedious), but at least Overleaf is pretty user-friendly.
https://tex.stackexchange.com/questions/111580/removing-an-unwanted-page-between-two-chapters
https://tex.stackexchange.com/questions/39988/space-before-chapters-and-contents
https://epsviewer.org/onlineviewer.aspx
https://www.overleaf.com/learn/latex/Bibtex_bibliography_styles

December 28, 2022 - January 1, 2023:

New Years

January 2, 2023:

I finally got all my previous work into Latex. I also made the whole structure of the document including the abstract, chapters, and appendix. Now we should just be able to drop content into it as we generate it. The only thing I have left to add is the bibliography.

January 6, 2023:

Wrote bibliography in Overleaf. That was more complicated than I expected. Thank goodness for nice people on StackExchange!
https://www.overleaf.com/learn/latex/Bibtex_bibliography_styles
https://tex.stackexchange.com/questions/17445/how-can-i-change-the-references-to-reference-in-the-thebibliography-environm
https://tex.stackexchange.com/questions/36396/how-to-properly-write-multiple-authors-in-bibtex-file
https://tex.stackexchange.com/questions/12597/renaming-the-bibliography-page-using-bibtex

January 7, 2023:

Started methodology section today: chose test IVPs for testing and wrote MATLAB code for visualizing their analytical solutions.
https://tex.stackexchange.com/questions/198432/using-the-tab-command
https://tex.stackexchange.com/questions/254785/e-vs-exp-in-display-mode
https://www.symbolab.com/solver/ordinary-differential-equation-calculator/y%E2%80%B2%5Cleft(t%5Cright)%2B4y%5Cleft(t%5Cright)%3D8t%2C%20y%5Cleft(0%5Cright)%3D1?or=input
https://garsia.math.yorku.ca/MPWP/LATEXmath/node9.html
https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols
https://www.overleaf.com/learn/latex/Subscripts_and_superscripts
https://tex.stackexchange.com/questions/20538/what-is-the-right-order-when-using-frontmatter-tableofcontents-mainmatter

January 8, 2023:

Wrote the intro to the Methodology chapter today. Also uploaded the MATLAB code from yesterday to Appendix A after figuring out how to format code using the mcode package
https://www.overleaf.com/latex/templates/highlighting-matlab-code-in-latex-with-mcode/nhtksndnsmmx
https://tex.stackexchange.com/questions/343723/where-put-mcode-sty
https://github.com/TeXworks/texworks/issues/828

January 9, 2023:

Wrote the Second Derivative section of the Methodology chapter.
https://de.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package
https://courses-archive.maths.ox.ac.uk/node/view_material/49161#:~:text=To%20include%20your%20Matlab%20file,m%7D.

January 11, 2023:

Wrote the Mean Square Error section of the Methodology chapter.
https://de.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package
https://courses-archive.maths.ox.ac.uk/node/view_material/49161#:~:text=To%20include%20your%20Matlab%20file,m%7D.
https://www.google.com/search?q=how+to+add+a+.sty+into+the+same+folder+as+your+latex+document&sxsrf=AJOqlzUkdWfoD-b9Hw1DH6YFPawDUeNJmQ:1673174702397&source=lnms&tbm=vid&sa=X&ved=2ahUKEwjn8KG-5bf8AhXLzqQKHW6kDrUQ_AUoAXoECAEQAw&biw=1536&bih=664&dpr=1.25#fpstate=ive&vld=cid:98f67153,vid:R5uFpUdnMA0

January 13, 2023:

I wrote some code today for the adaptive step size control method and the predictor corrector method, but it's a pretty hacky (my style lol) and I'm not sure if it's actually working correctly. I guess we probably won't include it in the project, but I spent my day doing it so I figured I would mention it.

January 14, 2023 - February 6, 2023:

Studying for and taking exams for other classes.

February 7, 2023:

We are getting pretty behind where we wanted to be in the Gannt Chart. Studying for exams just took more than we expected, but those are over now, so it's time to kick things into overdrive. I talked to Sophia today; she said she is basically done with the code, but still has a few bugs to work out to make the programs easier to use. She said she should have them ready for testing by the end of the week.

February 9, 2023:

I went back over the Introduction chapter making it look nicer and fixing some of the equations I messed up before. I spent the rest of the day getting the Results chapter of the report structured and ready for some tables and diagrams.

February 12, 2023:

Got the code for all three algorithms from Sophia. Took a couple hours to understand everything she did and then started writing up the descriptions of the algorithms in the Results chapter of the report.

February 15, 2023:

As written, the algorithms are solving y'=2x right now. Started trying to modify it to solve our test IVPs. It's so hard to work on somebody else's code!
I spent most of the day sifting through programming textbooks and old lecture notes as well as Reddit, StackExchange, Quora and Mathworks.com...no luck yet.

February 16, 2023:

The code is working for the linear IVP, but we are still stuck on the other two. The code is written to solve a first order ODE, but IVP 2 and 3 are second order. It seems like it should be so trivial to modify the algorithms to work for a linear system, but I suck at coding and I am struggling. The only thing that makes me feel better is the fact that Sophia can't figure it out either and she is *good* at coding...although that shouldn't make me feel better since we still don't have working code lol.

February 17, 2023:

Sophia is trying to modify the base script and I am working on the 2nd derivative algorithm since it's a little less complicated than the MSE one. When either of us is successful, we will do the same for the other two algorithms so that each algorithm solves all three test IVPs. A friend told me about a tool called chat GPT which can apparently write code for you, so now I am trying to figure out how to use that to get some help. No success yet, but from what I have seen so far, it's actually a pretty amazing tool.
https://chat.openai.com/chat

February 18, 2023:

I feel so close to getting this stupid thing to work I could cry. I have made and unmade so many changes, large and small, that it's getting difficult to keep track of what I have tried already. Every time I figure out how to deal with one error, three more show up that I have never heard of. I had high hopes for chat GPT at first, and it *is* an amazing tool for some things, but it's useless when it comes to writing code, at least MATLAB code for these purposes. Giving up on that pathway and going back to StackExchange and Mathworks now.

February 19, 2023:

Another full day spent trying to modify the 2nd derivative code. Starting to get discouraged.

February 20, 2023:

Met with Sophia again today. We decided we would try for one more day to modify the scripts to solve the test IVPs...if we are unsuccessful, we will meet again tomorrow and choose two more first order IVPs to test instead. I tried for three more hours with no success. Giving up. I guess we only have a first order ODE solver.

February 21, 2023:

Talked to Sophia and we just decided to cut our losses and come up with some easier test IVPs. We are going to stick with linear, non-linear and variable coefficients so I spent the afternoon looking for some suitable ones that are first order. Once I chose a couple, I remade the Analytical Solutions section of the Methodology chapter of the report. I also adapted the MATLAB code to generate solutions of the new IVPs and added those to the report as well.
https://mathdf.com/dif/#
https://www.wolframalpha.com/
https://www.symbolab.com/solver/ordinary-differential-equation-calculator

February 22, 2023:

Sophia got the algorithms working for IVP 3, but the one I chose for IVP 2 isn't working. So, we just decided to change it once more. I found another non-linear ODE and I think this one is even better since it includes a periodic trig function, which we did not have before. Once selected, I remade the analytical solution and the code for the graph and got them in the report.
https://mathdf.com/dif/#
https://www.wolframalpha.com/
https://www.symbolab.com/solver/ordinary-differential-equation-calculator

February 23, 2023:

Woot! We finally have some results to write about! Today I started getting stuff into the Results chapter of the report. I also added the basic structure of the Discussion chapter.

February 24, 2023:

Sophia added a bunch more stuff to the Results and Discussion chapters and the report is looking good. Today I started working on the Conclusion a bit, but I think she was working on it at the same time so, to avoid confusion in Overleaf, I decided to write the Abstract. This project is complete; it was time. :)

February 25, 2023:

Spent today going back through the entire report, editing and polishing. English isn't Sophia's first language so grammar and flow come down to me. She is still working to get some of the diagrams and stuff into the report. I also fixed one of the images that annoyingly wouldn't show up. It's minor, but I spent over an hour trying to figure it out.
https://tex.stackexchange.com/questions/214532/how-to-end-wrapfigure-environment

February 26, 2023:

The report is basically done so today I made the presentation slide show. I hope the professor doesn't just want a PDF because I spent a lot of time organizing and detailing the presentation notes as well. I think Sophia may want to add a couple slides of her awesome flow diagrams for the code, but we are basically now ready to present.
https://www.freecodecamp.org/news/eulers-method-explained-with-examples/
https://wethestudy.com/mathematics/newton-raphson-method-how-calculators-work/
https://rosettacode.org/wiki/Euler_method

February 27, 2023:

Today was all about finishing touches for my part. I spent some time formatting this document so it would be presentable, and I spent the rest going through the minor details (e.g. making sure variables are italicized and dimensions aren't, or cleaning up titles and axis labels etc.). Sophia has a bit more to add to the report tonight and I will do a final check tomorrow to make sure we are submission-ready.

February 28, 2023:

Did a final proof read through the entire document. Now compiling everything for submission.

Final Note:

I realize as I read back through this project log, that numerous times I made statements such as "I wrote the ... chapter of the report," but I want to stress that those statements should only imply that I got to them first. Sophia added a lot of really valuable input and content for the report, so I want to be clear that I am in no way taking the full credit. I hope I can honestly say that I also made some valuable contributions to the programming side of the project as well. Although Sophia technically wrote all three algorithms, I contributed in getting them to work for our IVPs and at least shared in the misery of defeat and in the triumph of success. That's gotta be worth something, right? :)

Project Journal: Testing Adaptive Step-Size Algorithms for an Implicit Euler Solver Using the Newton-Raphson Method

Submitted By:
Sophia Felicia Salome Döring

Project Journal for: 3301 Numerical Methods of Simulation
Submission Date: March 1, 2023

Week 1: 1-10 December

In week 1 the main focus was research and planning. To discuss different ways of structuring a program for implicit Euler method and ways of adjusting the step size were researched, such as Richardson extrapolation and predictor-corrector schemes. (https://people.sc.fsu.edu/ jpeterson/nde_book4.pdf(I could not find the book it is from)) Additionally, a Gannt Chart was organized by Aaron and approved by me.

Working time: 3 hours

Week 2: 11-17 December

The decision was made to split up the work in coding and writing report. For programming different Possibilities of structuring the were suggested:

1. use one main MATLAB file and create sub files with functions needed for the solving of the ODEs
2. create one file for every ODE to solve
3. create one file for every method and generalize them for every ODE to solve

The last idea was cosen, as it enables the usage of the programmed solver for other ODEs, too. To apply the same code on different ODEs the code was structured in three files for the implicit Euler with constant step size, with adjusted step size by the second derivative, and the adjusted step size by the Mean Square Error.[3–6] The first draft of the structure of implicit Euler method with a constant step size was created (see Figure 1). Even though the programming was done in MATLAB (MATLAB. (2022). version 9.13.0 (R2022b). Natick, Massachusetts: The MathWorks Inc.) which provides a root function to solve the roots as it is done by the Newton-Raphson Method. As this function gives all roots of a function and the results are the exact eigenvalues of a matrix within round off error of the companion matrix it seems not necessary to use this function as only the root close to the last step is needed. It was planned to write a Newton-Raphson Method with a numerically calculated derivative, which also provides the possibility of generalized usage of the function and does not need to be adapted to the

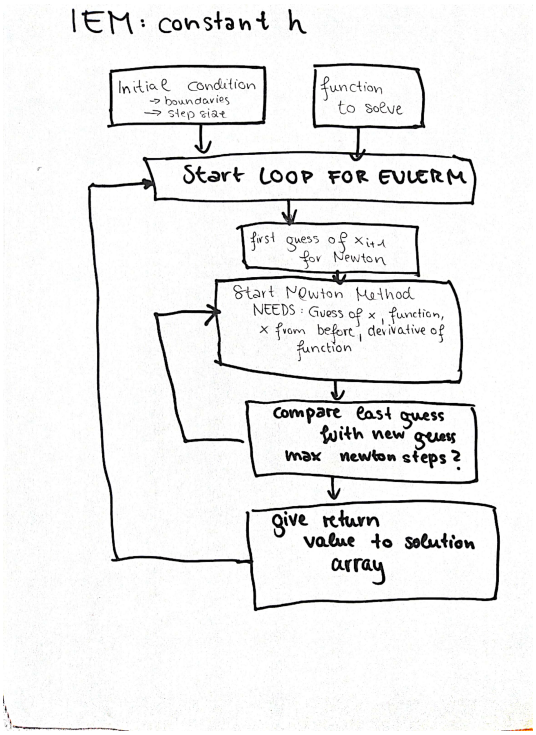

Figure 1: Draft of the structure of Implicit Euler Method for coding.

IVP that is tried to be solved.

Additionally, the first possible IVPs to solve where researched. [3] The criteria where, because of the definition of the implicit Euler method, that the ODEs are depending on x and t. Also an analytical solution of the ODE had to exist to evaluate the quality of the program and each method later.

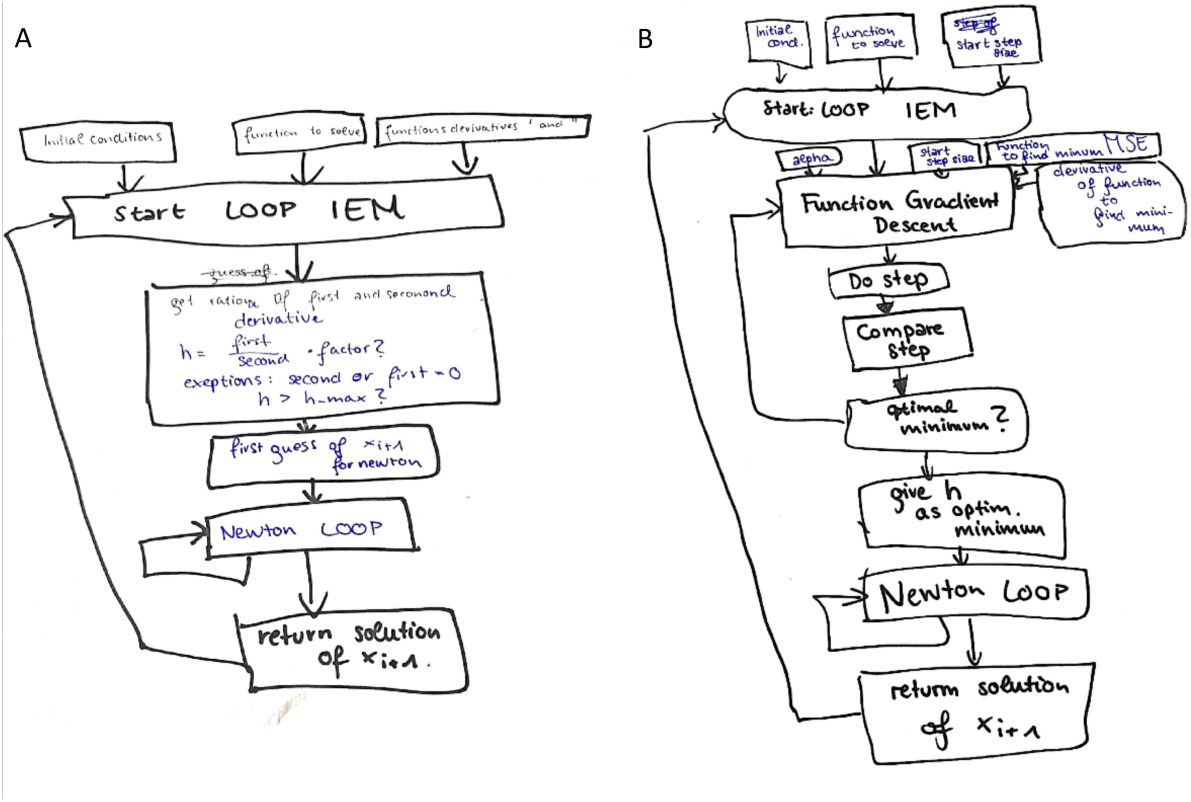

Figure 2: A shows the draft of the structure of the second derivative method, B shows the draft of the MSE method.

To program the MSE method a step size had to be calculated that minimizes the MSE. It was decided to use the gradient descent method for finding the minimum, which is often used in Machine Learning.[1] There are other forms of finding minima of a function numerically e.g. Newton-Raphson Method with first and second derivative. Gradient descent has the advantage, that only one additional numerical calculation has to be done (first derivative) compared to two, which can lead to more error. [2]

Some other ideas of organizing the code where discuss such as putting everything together in one file, creating different files for each IVPs, doing the evaluation in another subfile or not, and how to include the visualization of results in plots. In the end it was decided that the three files for every method also additionally include the evaluation with MSE. It was decided to use the sum of the MSE to evaluate the accuracy of the method.

Working time: 6 hours

Week 3: 18-24 December

The code for the constant step size was written. The example code from the lectures could have been used, but for better understanding a new code was created. In the beginning some problems occurred: The tolerance for the error was to high and the guess for the next Newton step was not precise enough, which had the consequence of long computational time and no result for the next value x. To solve this problem only the first newton step was assumed to be the *last value of x + the step size*. For every other guess of x the guess was the *last value of x + the difference between the last value x and the value of x before that*. Despite the better guess of the root a mistake in the code lead to the problem of incorrect values and a graph that was very much off compared to the analytical solution of the IVP 1. First it was assumed, that a mistakes in the calculation of the original function was the problem, but several other functions and also the endless usage of extra brackets did not help to solve the problem. The mistake could not be found during working on the code in this week.

Working time: 8 hours

Week 4-5: 25 Decemeber - 07 January

After searching for a few hours to find the mistake in the code, it was detected to be a missing minus sign before one bracket in code for the newton method. After this the code worked out to show suitable results and the code for comparing the calculated values with the real solution values (MSE) was written. The code for solving the second derivative method and comparing it with the real values was also written. Because of the function the first IVP 1 which does have a second derivative of 0, the code did not show a solution, the exception of switching to a maximal step size when first and second derivative are 0 was included. Therefore the program showed the same results as the constant step size as the maximal step size was the same as the constant step size. As there was no other solution to be expected from this function another test function from engcourses uofa.ca [3] was tested, which lead to a suitable result for this function. Because of Christmas break on the fifth week, only the notation and description of the code of the constant step size was done during that time.

Working time: 6 hours

Week 6: 8-14 January

The code for the MSE method was written. As start value for the h the constant step size was used. For finding the h at the minimum of the MSE function, the numerical calculation the derivative of the MSE function was coded. For this MSE function the

value of x+1 with the starting h had to be calculated with the Newton Method and the difference of the value from the true value had to be calculated. The derivative of the MSE function was then multiplied with the gradient descent step α and substracted from the starting value of h. If the new value of h does differ from the starting value of h more than a respective error, the new value h became the starting value. The loop steps were determined by a maximal iteration, which was decided to be the same as the max. iterations as the newton method. If there is no minimum found the step size was set to a very small step size.[2]

Working time: 7 hours

Week 7: 05-11 February

In week 7 the code was tested for all IVPs. The original IVPs that were decided to use for the program were changed as one was not meeting the needed conditions of the implicit Euler Method (depending on x and t) and the other one was a second ODE. During this week it was tried to adjust the code to solve the second ODE by creating the system of first ODE and solving the created system of ODEs with the Newton-Raphson Method and the help of the Jacobian Matrix. Even though there is a function in MATLAB for the Jacobian Matrix , it was decided to write a numerically solution of the Jacobian Matrix and the functions to generalize to application and to avoid the additional computational time by the analytical solving of the Jacobian Matrix by MATLAB. Even though it was tried for several hours to get a suitable solution of the system of differential equations no solution could be found and the mistake was still the same. Even though a solution was produced by the code it was not a suitable one for the IVP.

Working time: 20 hours

Week 8: 12-18 February

The mistake could not be solved and no suitable solution was created by the program. The last try is shown in chapter "Last try of solving the second ODE". Therefore the IVP was exchanged for a new IVP with variable coefficients. This IVP was solvable with the written code. The results were then visualized by graphs. For plotting the solutions of each method on one IVP the solutions were exported to another MATLAB code for plotting the solutions. For putting the results into the report the graphs were additionally added and a close up to see the difference between the different solution methods needed to be created for every IVP. To get the computational time the function `tic... toc` of MATLAB was included.

Last try of solving the second ODE

```
1
2  %==========================================================================
3  % PROJECT FOR NMoS: INVESTIGATION OF THE INFLUENCE OF AN ADJUSTABLE
4  %                    STEP-SIZE IN THE BACKWARD EULER METHOD
5  %--------------------------------------------------------------------------
6  %Three methods of adjusting h are compared in their computational time
7  %accuracy and stability:
8  %   1. constant h
9  %   2. using the second derivative
10 %   3. adjusting the h in a way to minimize the MSE by using gradient
11 %       descent
12 %
13 %To test these methods functions with a known solution are tested.
14 %--------------------------------------------------------------------------
15 %In this file the Implicit Euler Method (IEM) is done with the constant h
16 %==========================================================================
17
18
19 %--------------------------------------------------------------------------
20 % PUT IN INITIAL VALUES AND BOUNDARIES
21 %--------------------------------------------------------------------------
22 clear all;
23 x_start = 1;
24 u_start = 0;
25 t_start = 0;
26 t_end = 2;
27 const_h = 0.001;
28 step_newt = 10000;
29
30 %--------------------------------------------------------------------------
31 % START SOLVING WITH CONSTANT H
32 %--------------------------------------------------------------------------
33
34 t_array = t_start:const_h:t_end; %setting the whole time array
35 n = numel(t_array); %finding the number for the loop for finding x values
36
37 for i=1:n-1 %LOOP FOR EULER STEPS
38     x_sol(1) = x_start; %solution of x with the newton method
39     u_sol(1) = u_start;
40     x_guess(1)= x_sol(1)+const_h;%guess of x for the newton method
41     u_guess(1) = u_sol(1)+const_h;
42     [x_sol(i+1),u_sol(i+1)] = newton(@dx_dt,x_sol(i),x_guess(i),@du_dt,u_sol(i),u_guess(i), ...
43         t_array(i)+const_h,const_h,step_newt);%solution of x with the newton method
44     x_guess(i+1) = x_sol(i+1)+(x_sol(i+1)-x_sol(i));
45     u_guess(i+1) = u_sol(i+1)+(u_sol(i+1)-u_sol(i));%guess of x for the newton method
46 end %LOOP FOR EULER STEPS
47
48 %--------------------------------------------------------------------------
49 % REAL VALUES OF X AND MSE
50 %--------------------------------------------------------------------------
51 for i=1:n-1 %LOOP FOR REAL VALUES
52     x_true(1) = x_start; %solution of x
53     mse_value(1) = 0;
54     x_true(i+1) = dx_dt_sol(t_array(i+1),x_start);
55     mse_value(i+1) = MSE(x_sol(i+1),x_true(i+1));
56 end %LOOP FOR REAL VALUE
57 mse_sum = sum(mse_value);
58 %--------------------------------------------------------------------------
59 % Visualisation of the Solution
60 %--------------------------------------------------------------------------
61 tiledlayout(1,2)
62 nexttile
63 plot(t_array,x_sol,"--",'Color',[0, 0.5, 0.3],'LineWidth',1);
64 hold on
65 plot(t_array,x_true,"--",'Color',[0.5, 0, 0.3],'LineWidth',1);
66 xlabel("t");
67 ylabel("x");
68 nexttile
69 plot(t_array,mse_value,"--",'Color',[0.3, 0, 0.5],'LineWidth',1);
70
71
72
73
74
75 %==========================================================================
76 % FUNCTIONS
77 %==========================================================================
78
79 %--------------------------------------------------------------------------
80 % FUNCTION TO SOLVE AND FUNCTION FOR REAL SOLUTION (CHANGE THESE FOR YOUR
81 % PuRPOSE)
82 %--------------------------------------------------------------------------
83 function funcx = dx_dt(u,t)
84   funcx = u;
85
86 end
87 function funcu = du_dt(u,x,t)
88     funcu = -x^2
89 end
```

```
90
91  function f_solution = dx_dt_sol(t,x_0)
92      f_solution = (-1)/(t-1);
93      %f_solution = (1/(sqrt(2)))*(t^(sqrt(2)))+(1/(sqrt(2)))*(t^(-sqrt(2)));
94  end
95
96
97
98  %-------------------------------------------------------------------------
99  % FUNCTION FOR NEWTON METHOD
100 %For solving the Newton step for several functions depending on several
101 %variables a vector containing both function needs to be implemented of
102 %which you then try to f nd the root.
103 %The newton step (as a logical explaination) than is:
104 %next newton step (vector of the two variables) =
105 % = last newton step (vektor of the two variables)+ inverse of (the
106 % jacobian matrix * the vector of the functions)*the vector of the
107 % functions with last newton step variables
108 % The functions to get the roots for are implicit euler steps from both
109 % first order ordinary functions
110 % to get the jacobian the normal jacobian function provided by matlab is
111 % not use as all steps are tried to be solved numerically and nor
112 % symbolically (time consuming)
113 %
114 %
115 %-------------------------------------------------------------------------
116 function [x_newt u_newt] = newton(dx_dt,x_i, x_guess,du_dt,u_i,u_guess, ...
117     t, h,step_newt)
118 %1. Implementation of variables
119   prime_fact = 0.0001; %For getting the drivative
120   error = 0.000000000001; %Tolerance for finding the root
121
122  %2.First newton step
123
124   search = zeros(2,step_newt);%search is the matrix containing each
125   % vector step for finding the root
126   search(:,1) = [x_guess;u_guess];%The first search step are the
127   % first guesses of x and u
128   t=[x_guess;u_guess]
129   j=1;
130   %Implementing the numerical jacobian, all four elements of the jacobian
131   %times the function vector are aclculated separately
132   f1_x=((eul_dx_dt(@dx_dt,x_i,search(1,j)+prime_fact,search(2,j),h))- ...
133       (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
134   f1_u=((eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j)+prime_fact,h))- ...
135       (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
136   f2_x=((eul_du_dt(@du_dt,u_i,search(2,j),search(1,j)+prime_fact,h,t))- ...
137       (eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)))/prime_fact;
138   f2_u=((eul_du_dt(@du_dt,u_i,search(2,j)+prime_fact,search(1,j),h,t))- ...
139       (eul_du_dt(@du_dt,u_i,search(2,j),search(2,j),h,t)))/prime_fact;
140   Df = [f1_x f1_u; f2_x f2_u]
141   inv(Df)
142   fer=[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h);
143       eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]
144   g=inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h);
145       eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]
146   t1=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
147       eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)])
148   search(:,1)
149   search(:,j+1)=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
150       eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]);
151   j = 2;
152  %3.newton steps
153   while (abs(search(1,j)-search(1,j-1))>error) & (abs(search(2,j)- ...
154           search(2,j-1))>error) & j < step_newt %LOOP FOR NEWTON STEPS
155     f1_x=((eul_dx_dt(@dx_dt,x_i,search(1,j)+prime_fact,search(2,j),h))- ...
156         (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
157     f1_u=((eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j)+prime_fact,h))- ...
158         (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
159     f2_x=((eul_du_dt(@du_dt,u_i,search(2,j),search(1,j)+prime_fact,h,t))- ...
160         (eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)))/prime_fact
161     f2_u=((eul_du_dt(@du_dt,u_i,search(2,j)+prime_fact,search(1,j),h,t))- ...
162         (eul_du_dt(@du_dt,u_i,search(2,j),search(2,j),h,t)))/prime_fact
163     Df = [f1_x f1_u; f2_x f2_u];
164     r=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
165         eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)])
166     search(:,j+1)=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
167         eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]);
168     j=j+1;
169   end%LOOP FOR NEWTON STEPS
170   if (abs(search(1,j)-search(1,j-1))<error) &
171       (abs(search(2,j)-search(2,j-1))<error) %IF STATEMENT FOR NEWTONSTEPS
172       % (if not successful)
173       x_newt  = search(1,j);
174       u_newt = search(2,j);
175   else
176       x_newt  = 0;
177       u_newt = 0;
178   end%IF STATEMENT FOR NEWTONSTEPS
179
180     function eul_dx_dt = eul_dx_dt(dx_dt,x_i,x_i1,u_1,h)
181         eul_dx_dt = x_i+h*dx_dt(u_1,t)-x_i1;
182     end
183     function eul_du_dt = eul_du_dt(du_dt,u_i,u_i1,x_1,h,t)
```

```
184             eul_du_dt = u_i+h*du_dt(u_i1,x_1,t)-u_i1;
185         end
186
187
188     end
189
190     %-------------------------------------------------------------------------
191     % FUNCTION FOR MEAN SQUARE ERROR
192     %-------------------------------------------------------------------------
193
194     function MSE = MSE(x_estimate,x_true)
195         MSE = (x_estimate-x_true)^2;
196     end
```

Working time: 24 hours

Week 9-10: 19-28 February

As the week 10 only included only three days it is included into week 9. To get the accuracy, the code used the summed MSE, but because of different step numbers in the same boundary the summed MSE was not suitable for measuring the accuracy. Therefore, the average MSE over all step sizes was used as a new measurement for the accuracy. The results where then included into the report. The results where than discussed and the discussion was updated. The flowcharts of the codes where illustrated and included to the report. The last few days were spent on the correction of the report and the description of the code in the code itself.

Working time: 40 hours

Bibliography

[1] Baird III, L. C. (1999). Reinforcement learning through gradient descent. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[2] Donges, N. (2022). Gradient descent in machine learning: A basic introduction. https://builtin.com/data-science/gradient-descent.

[3] engcourses uofa.ca (2023). Solution methods for ivps: Backward (implicit) euler method. https://engcourses-uofa.ca/books/numericalanalysis/ordinary-differential-equations/solution-methods-for-ivps/backward-implicit-euler-method/. 2022-12-05.

[4] Hairer, E. and Lubich, C. (2010). Numerical solution of ordinary differential equations.

[5] modellingsimulation.com (2020). Implicit euler method by matlab to solve an ode. https://www.modellingsimulation.com/2020/03/implicit-euler-method-by-matlab-to.html.

[6] The MathWorks Inc (2023). Help center: roots. https://de.mathworks.com/help/matlab/ref/roots.html. 2022-12-05.