HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

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

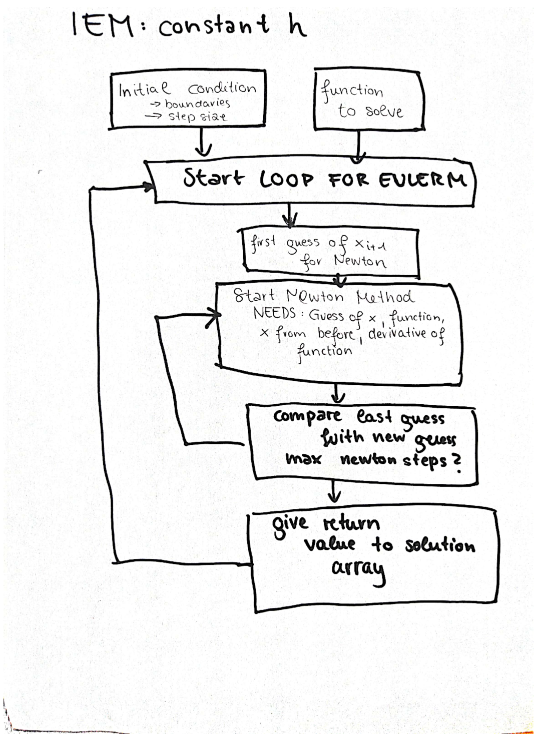
IEM: constant h
Initial condition → boundaries → step size
function to solve
Start LOOP FOR EULERM
first guess of x_{i+1} for Newton
Start Newton Method NEEDS: Guess of x, function, x from before, derivative of function
compare last guess with new guess max newton steps?
give return value to solution array

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.

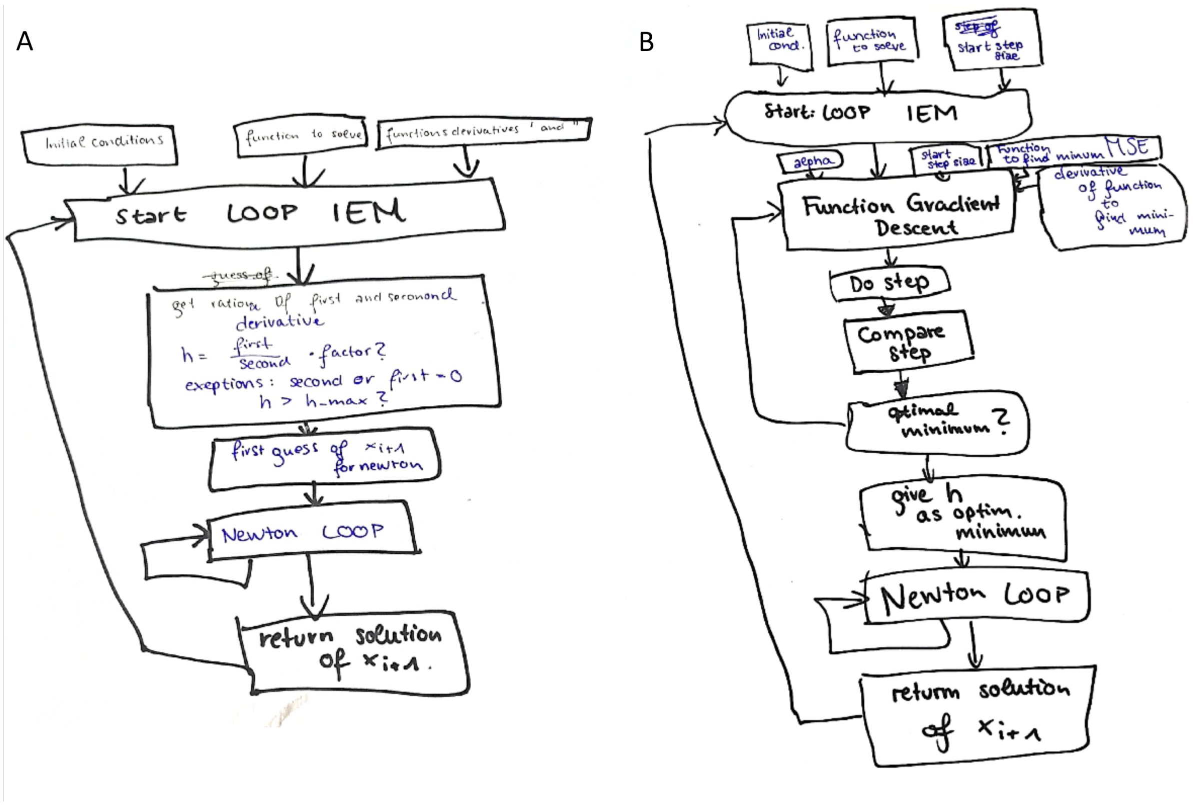
A
Initial conditions
function to solve
functions derivatives ' and ''
Start LOOP IEM
~~guess of~~
get ratio of first and second derivative
h = first/second · factor?
exceptions: second or first = 0
h > h_max ?
first guess of x_{i+1} for newton
Newton LOOP
return solution of x_{i+1}.
B
Initial cond.
function to solve
~~step of~~ start step size
Start: LOOP IEM
alpha
start step size
function to find minum MSE
derivative of function to find minimum
Function Gradient Descent
Do step
Compare step
optimal minimum?
give h as optim. minimum
Newton LOOP
return solution of x_{i+1}

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

```matlab

%==========================================================================
% 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
%--------------------------------------------------------------------------
clear all;
x_start = 1;
u_start = 0;
t_start = 0;
t_end = 2;
const_h = 0.001;
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

for i=1:n-1 %LOOP FOR EULER STEPS
    x_sol(1) = x_start; %solution of x with the newton method
    u_sol(1) = u_start;
    x_guess(1)= x_sol(1)+const_h;%guess of x for the newton method
    u_guess(1) = u_sol(1)+const_h;
    [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), ...
        t_array(i)+const_h,const_h,step_newt);%solution of x with the newton method
    x_guess(i+1) = x_sol(i+1)+(x_sol(i+1)-x_sol(i));
    u_guess(i+1) = u_sol(i+1)+(u_sol(i+1)-u_sol(i));%guess of x for the newton method
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; %solution 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_sum = sum(mse_value);
%--------------------------------------------------------------------------
% 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);





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

%--------------------------------------------------------------------------
% FUNCTION TO SOLVE AND FUNCTION FOR REAL SOLUTION (CHANGE THESE FOR YOUR
% PuRPOSE)
%--------------------------------------------------------------------------
function funcx = dx_dt(u,t)
  funcx = u;

end
function funcu = du_dt(u,x,t)
    funcu = -x^2
end
```

```matlab

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



%-------------------------------------------------------------------------
% FUNCTION FOR NEWTON METHOD
%For solving the Newton step for several functions depending on several
%variables a vector containing both function needs to be implemented of
%which you then try to f nd the root.
%The newton step (as a logical explaination) than is:
%next newton step (vector of the two variables) =
% = last newton step (vektor of the two variables)+ inverse of (the
% jacobian matrix * the vector of the functions)*the vector of the
% functions with last newton step variables
% The functions to get the roots for are implicit euler steps from both
% first order ordinary functions
% to get the jacobian the normal jacobian function provided by matlab is
% not use as all steps are tried to be solved numerically and nor
% symbolically (time consuming)
%
%
%-------------------------------------------------------------------------
function [x_newt u_newt] = newton(dx_dt,x_i, x_guess,du_dt,u_i,u_guess, ...
    t, h,step_newt)
%1. Implementation of variables
  prime_fact = 0.0001; %For getting the drivative
  error = 0.000000000001; %Tolerance for finding the root

 %2.First newton step

  search = zeros(2,step_newt);%search is the matrix containing each
  % vector step for finding the root
  search(:,1) = [x_guess;u_guess];%The first search step are the
  % first guesses of x and u
  t=[x_guess;u_guess]
  j=1;
  %Implementing the numerical jacobian, all four elements of the jacobian
  %times the function vector are aclculated separately
  f1_x=((eul_dx_dt(@dx_dt,x_i,search(1,j)+prime_fact,search(2,j),h))- ...
      (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
  f1_u=((eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j)+prime_fact,h))- ...
      (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
  f2_x=((eul_du_dt(@du_dt,u_i,search(2,j),search(1,j)+prime_fact,h,t))- ...
      (eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)))/prime_fact;
  f2_u=((eul_du_dt(@du_dt,u_i,search(2,j)+prime_fact,search(1,j),h,t))- ...
      (eul_du_dt(@du_dt,u_i,search(2,j),search(2,j),h,t)))/prime_fact;
  Df = [f1_x f1_u; f2_x f2_u]
  inv(Df)
  fer=[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h);
      eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]
  g=inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h);
      eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]
  t1=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
      eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)])
  search(:,1)
  search(:,j+1)=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
      eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]);
  j = 2;
 %3.newton steps
  while (abs(search(1,j)-search(1,j-1))>error) & (abs(search(2,j)- ...
          search(2,j-1))>error) & j < step_newt %LOOP FOR NEWTON STEPS
    f1_x=((eul_dx_dt(@dx_dt,x_i,search(1,j)+prime_fact,search(2,j),h))- ...
        (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
    f1_u=((eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j)+prime_fact,h))- ...
        (eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h)))/prime_fact
    f2_x=((eul_du_dt(@du_dt,u_i,search(2,j),search(1,j)+prime_fact,h,t))- ...
        (eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)))/prime_fact
    f2_u=((eul_du_dt(@du_dt,u_i,search(2,j)+prime_fact,search(1,j),h,t))- ...
        (eul_du_dt(@du_dt,u_i,search(2,j),search(2,j),h,t)))/prime_fact
    Df = [f1_x f1_u; f2_x f2_u];
    r=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
        eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)])
    search(:,j+1)=search(:,1)+(inv(Df)*[eul_dx_dt(@dx_dt,x_i,search(1,j),search(2,j),h); ...
        eul_du_dt(@du_dt,u_i,search(2,j),search(1,j),h,t)]);
    j=j+1;
  end%LOOP FOR NEWTON STEPS
  if (abs(search(1,j)-search(1,j-1))<error) &
      (abs(search(2,j)-search(2,j-1))<error) %IF STATEMENT FOR NEWTONSTEPS
      % (if not successful)
      x_newt  = search(1,j);
      u_newt = search(2,j);
  else
      x_newt  = 0;
      u_newt = 0;
  end%IF STATEMENT FOR NEWTONSTEPS

    function eul_dx_dt = eul_dx_dt(dx_dt,x_i,x_i1,u_1,h)
        eul_dx_dt = x_i+h*dx_dt(u_1,t)-x_i1;
    end
    function eul_du_dt = eul_du_dt(du_dt,u_i,u_i1,x_1,h,t)
```

```
            eul_du_dt = u_i+h*du_dt(u_i1,x_1,t)-u_i1;
        end


    end

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

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