ELEC518 Homework 4

Ender Erkaya

January 2022

Truth ... is much too complicated to allow anything but approximations.
John Von Neumann

Problem 1

The differential equation $\frac{d}{dt}x(t) = \lambda x(t)$ is solved using the approximation of the derivative at $t_m = mh$ by:

$$\frac{d}{dt}x(t) = \lambda x(t_m) = \frac{x(t_{m+1}) - x(t_{m-1})}{2h}$$

The approximation used is visualised below as Figure 1:

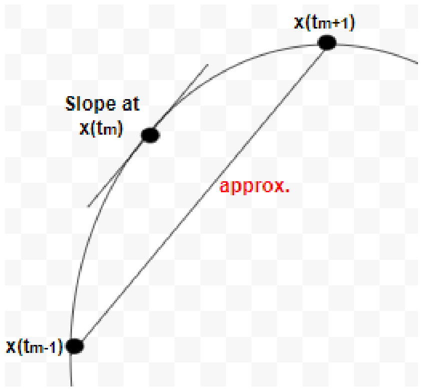

Figure 1: Used Approximation Scheme, P1a

This leap-frog technique is a 2-step technique. Above equation equals to the below integral equation:

$$x(t_{m+1}) = x(t_{m-1}) + \int_{x(t_{m-1})}^{x(t_{m+1})} \lambda x(\tau) d\tau$$

where integral part is approximated by:

$$\int_{x(t_{m-1})}^{x(t_{m+1})} \lambda x(\tau) d\tau = 2\Delta t \lambda x(t_m)$$

The below figure 2 demonstrates the approximation scheme: This approxima-

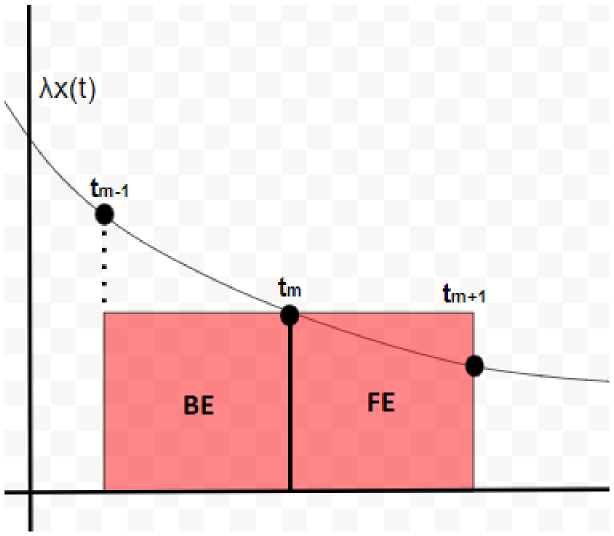

Figure 2: Integral Visualization of Approximation, P1a

tion of integration corresponds Backward Euler step, followed by a forward euler step. One iteration of the 2-step algorithm may be regarded as a 1-step backward euler(btw t_{m-1}, t_m) followed by 1-step forward euler(btw t_m, t_{m+1})). Intuitively, I would expect this scheme perform better than forward euler and backward euler, more similar to trapezoid method. Because, forward euler and backward euler for this problem have opposite error characteristics and the combined error tends to be smoothed.

1a

Above 2-step method can be written as:

$$\hat{x}_l - \hat{x}_{l-2} = 2h\lambda\hat{x}_{l-1}$$

The multistep parameters(nonzero) are:

$$\alpha_0 = 1$$

$$\alpha_2 = -1$$

$$\beta_1 = 2$$

The multi-step technique is an explicit technique since $\beta_0 = 0$ shows computing next point does require only previous and current information. We have following exactness linear system for a p-monomial solution t^p of our 2-step method, ie causality for LTI system.

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & -1 & -1 \\ 4 & 1 & 0 & -4 & -2 \\ 8 & 1 & 0 & -12 & -3 \\ 16 & 1 & 0 & -32 & -4 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 6 \\ 8 \end{bmatrix}$$

Our method satisfies the exactness for monomials with order of $p \leq 2$. Hence,

$$\mathbf{LTE} = C(\Delta t)^3 = \frac{\lambda^3}{6} h^3 = O(h^3)$$

This 2-step leap frog method results truncation error similar to trapezoid method as expected.

1b

For the method, we have the following stability difference equation:

$$E^l - 2\lambda h E^{l-1} - E^{l-2} = u^l$$

Rearranging as a linear system of equations:

$$E = \begin{bmatrix} E^l \\ E^{l-1} \end{bmatrix} = \begin{bmatrix} 2\lambda h & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} E^{l-1} \\ E^{l-2} \end{bmatrix} + \begin{bmatrix} u^l \\ 0 \end{bmatrix}$$

Then, eigenvalues of the system is the solution of

$$\xi^2 - 2z\xi - 1 = 0$$

where

$$z = \lambda h$$

Small time step stability($z = 0$): $\{-1, 1\}$
Since, the roots are equal to 1 in magnitude and distinct, the method is **stable.**(Small time step Stability Theorem)
Local truncation error $\to 0$ $(p = 2)$: **Consistency**
Accumulation not grows : **Stability**
Both consistency(local condition) and stability(global condition) results that the method is **Convergent**. (Dahlquist Equivalence Theorem)
Dahlquist First Stability Barrier states that an accuracy p of s-step linear multistep explicit method must satisfy:

$$p \leq s$$

The leapfrog method has accuracy $p = 2$ as $O(h^3)$. Hence, it remains in Dahlquist Stability Barrier.
Large time step stability analysis results roots are at: $\{z+\sqrt{1+z^2}, z-\sqrt{1+z^2}\}$ where

$$z = \lambda h$$

Absolute stability requires:

$$|\xi_{1,2}| \leq 1$$

This forces:

$$Re\{z\} = Re\{\lambda h\} = 0$$

and

$$Im\{z\} = |Im\{\lambda h\}| < 1$$

The absolute stability domain is a line segment on the imaginary axis, ie if $Re\{\lambda\} \neq 0$, we do not have convergence guarantee for larger discretization steps: It corresponds below absolute stability region(Figure 3):

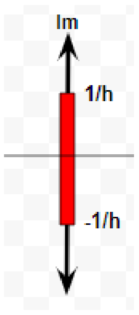

Figure 3: Absolute Stability Domain, P1b

This means for larger discretization steps h, the leapfrog algorithm does not have convergence guarantee, ie not A-stable. Consequently, the explicit midpoint algorithm is stable, ie global error goes to zero by $h \to 0$, but not absolutely stable, ie unstable for larger time steps.

1c

For different step sizes, $h \in \{2, 0.5, 0.1\}$, the numerical solution together with exact solution is plotted(Figure 4)

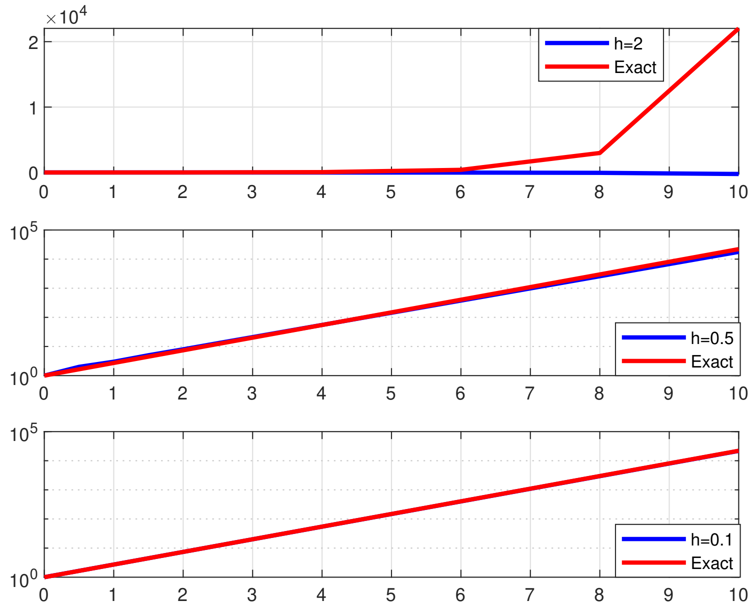

Figure 4: Numerical Solutions vs Exact Solution, P1c

1d

The error accumulation is very apparent for step size $h = 2$. As step size increases, the global error increases. The method does not have stability and convergence as expected for large values. If we use larger step sizes, ie $h > 1$, the method diverges and it became useless. To utilize the explicit leapfrog method, we must use smaller step sizes.

Problem 2

The simulated circuit has the following ode system:

$$\mathbf{C}\ \dot{\mathbf{v}} = -\mathbf{G}\ \mathbf{v} + \mathbf{i}_s$$

2a

The loadMatrix is modified to create capacitance matrix after capacitor values are read by readckt script. The modifications for creating capacitance matrix is given below snippet:

Listing 1: loadNewton.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
...
% Load the capacitors.
for i = 1:size(Capacitors,1)
    n1 = Capacitors(i,2);
    n2 = Capacitors(i,3);
    cap = Capacitors(i,1);
    if (n1 > 0)
        C(n1, n1) = C(n1,n1) + cap;
    if (n2 > 0)
        C(n2, n2) = C(n2,n2) + cap;
    end
end
...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The loadmatrix further modified to create initial values vector z after reading of initial values by readckt script.

Listing 2: loadNewton.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
...
% Load the initial values
for i = 1:size(Initials,1)
    n1   = Initials(i,2);
    zval = Initials(i,1);
    if (n1 > 0)
        z(n1)=zval;
    end
end
...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

2b

On contrary to that is in hw1, for the given Poisson equation we do not have any heat sinks that correspond to voltage sources or heat sources that correspond to current sources or heat air dissipation component that had represented as air resistors. In the left hand side we have time derivative of $T(x)$ as:

$$\frac{\partial(T(x))}{\partial t}$$

The time derivative component can be simulated using capacitors. The Poisson equation simplifies to:

$$\mathbf{C}\ \dot{\mathbf{v}} = \mathbf{G}\ \mathbf{v}$$

For the purpose, assigning capacitor and resistor values together with initial node voltages we can use the circuit simulator.

2c

The solutions at time steps $t = \{0, 2, 4, 6, 8, 10\}$ obtained by Forward Euler algorithm is given below in figure 5.

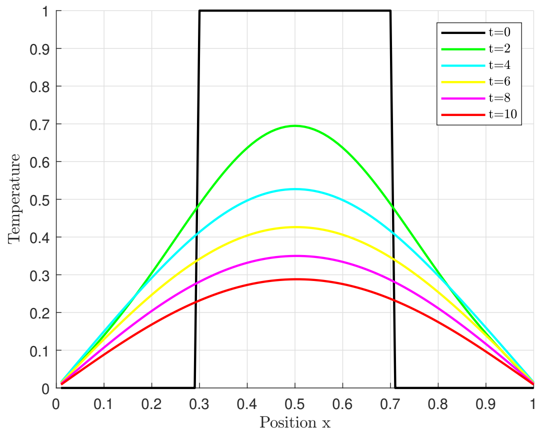

Figure 5: Forward Euler, N=100, h=1e-3, P2c

When we increase the step size to $h = 5e-3$, the obtained solutions are given below in the figure 6.

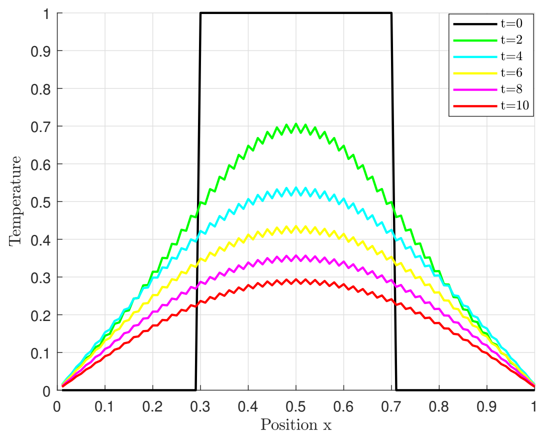

Figure 6: Forward Euler, N=100, h=0.5e-2, P2c

The obtained solution by Backward Euler Algorithm at $h = 0.1$ is given as figure 7

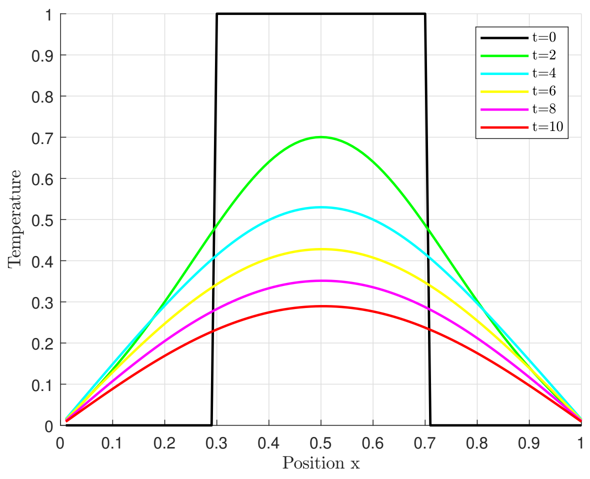

Figure 7: Backward Euler, N=100, h=0.1, P2c

For BDF2 solution, we set even $h = 1$ that results corresponding plot figure 9:

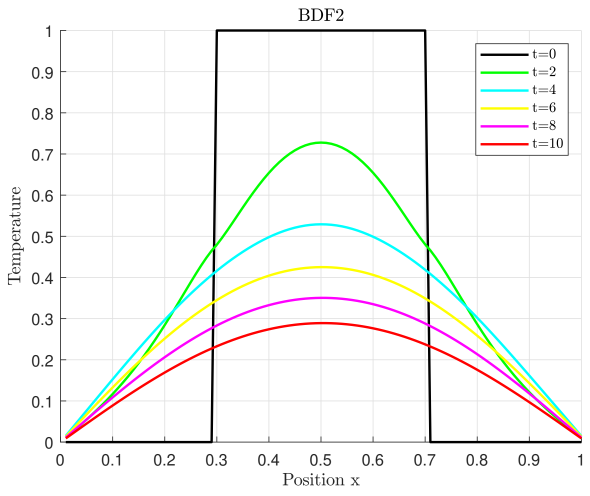

Figure 8: BDF2, N=100, h=1, P2c

For $N = 10$, the obtained solution is given below:

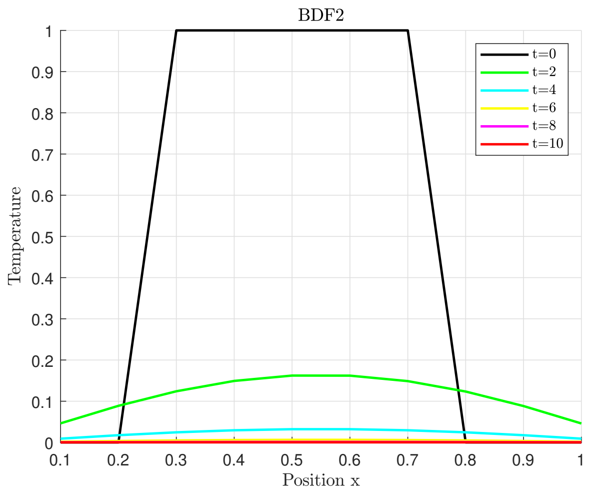

Figure 9: BDF2, N=10,h=1, P2c

We plot norm error values compared to the obtained solutions by each algorithm at $h = 1e - 4$, according to different step sizes. The obtained error plot is given by figure 10 We observe that forward euler algorithm diverges, unstable after $5e - 3$ immediately. Our stability analysis in Problem 5 reveals that algorithm is unstable for $\lambda < \frac{-2}{h}$. We have maximum negative eigenvalue

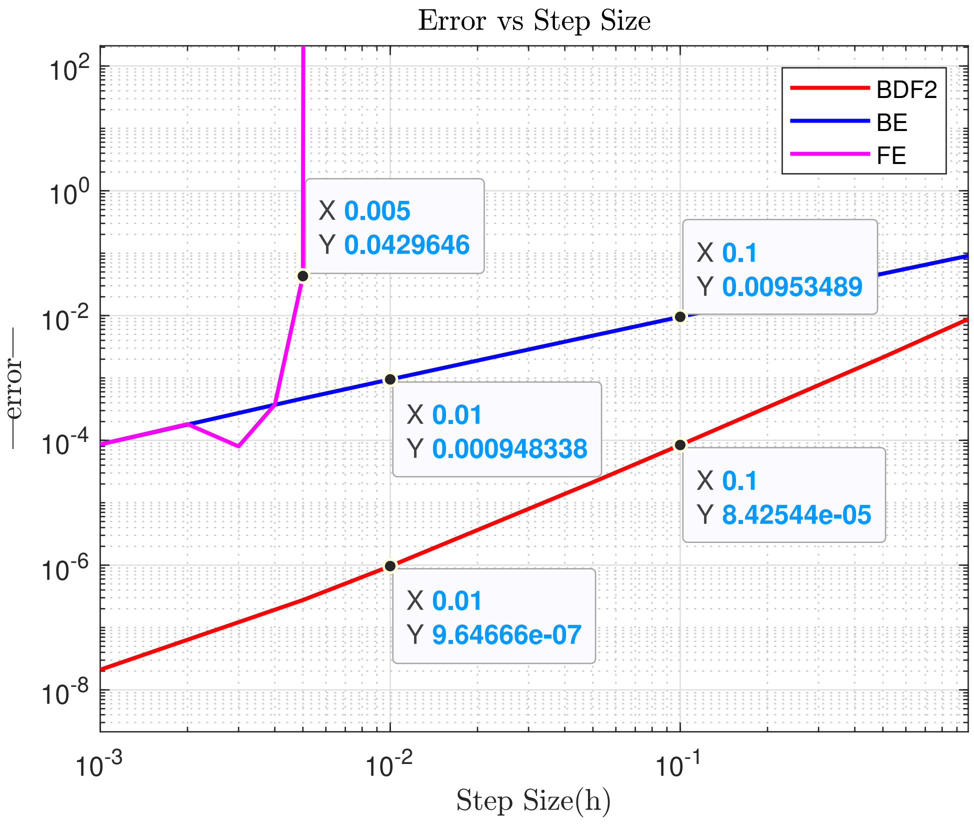

Figure 10: —Error— vs Step Size(N=100), P2c

of $-G$ matrix as $\lambda_{min} = -399$ that corresponds to unstability for

$$h > 0.005$$

. The obtained numerical result matches to our theoretical findings. On the contrary to Forward Euler Algorithm, Backward Euler is very stable for higher step sizes. According to plot, error experienced at Backward Euler is order 1 of step size:

$$G_{BE}(error) = Ch$$

that is also theoretically found in Problem 5. Even further, BDF2 algorithm converges with order 2 of step size h, which is the fastest convergence ratio among the three algorithms analyzed:

$$G_{BDF2}(error) = Ch^2$$

Since, the original problem is stable, ie $\forall i : \lambda_i < 0$, we cannot observe unstability for Backward Euler and BDF2 algorithms. Because, they are A-stable, ie absolutely stable for any left hand side λ(stable system). Aside from not creating a numerical unstability, both Backward Euler and BDF2 schemes may highly probably acquire stable solutions even unstable problems. That is due to their stability region coincides the right hand side.

We also plot the absolute errors for $N = 10$ and we get the same convergence ratios as observed in $N = 100$. However, the main difference is where Forward Euler Algorithm becomes unstable. According to plot, Forward Euler

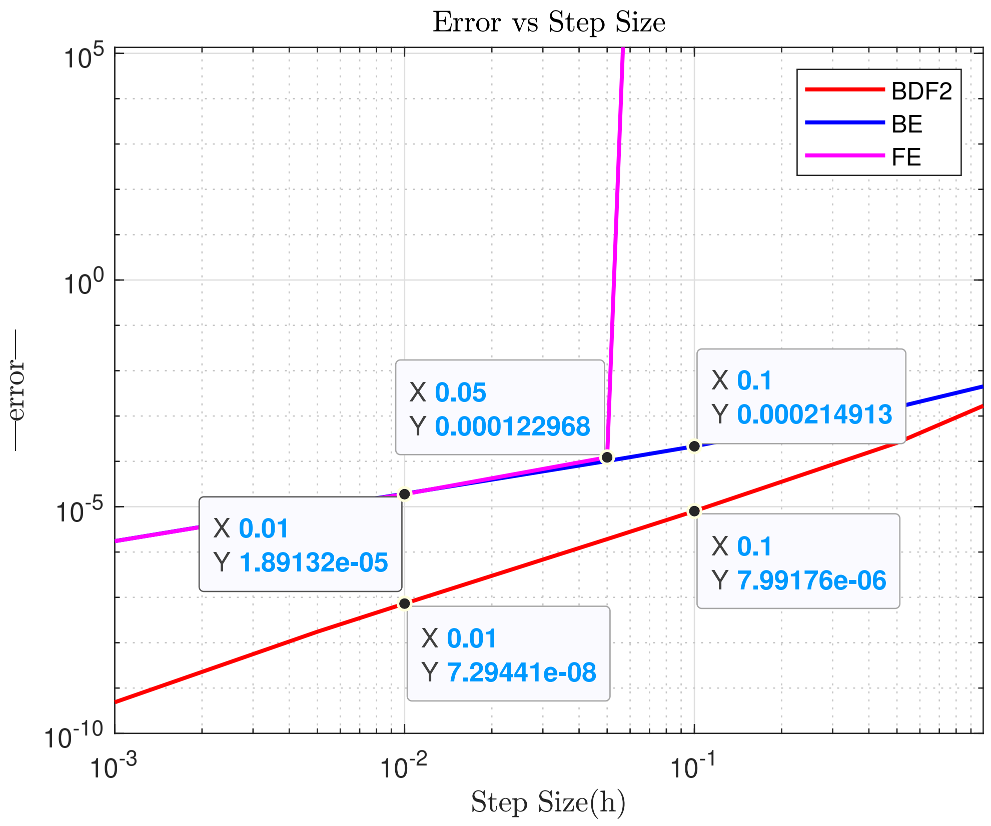

Figure 11: —Error— vs Step Size(N=10), P2c

algorithm becomes unstable after $h > 0.05$. This numerical observation become meaningful when we check the maximum eigenvalue of the RHS matrix.

$$\lambda_{min} = -39$$

and putting the boundary of absolute stability reveals the above numerical result. Whereas Forward Euler Algorithm at $N = 10$ diverges after $h > 0.05$, it diverges at $N = 100$ after $h > 0.005$. As λ converges toward the origin, the algorithms stability region improves. Due to absolute stability of BDF1 and BDF2 algorithms, we cannot observe unstability for any stable problem.

Problem 3

3a

Defining an auxiliary 2-dimensional vector as:

$$\eta = [\ x \ \ \dot{x}\ ]^T$$

we form first-order ODE as:

$$\dot{\eta} = \mathbf{A}(\eta)$$

where

$$\mathbf{A}(\eta) = \begin{bmatrix} \eta(2) \\ \\ -\eta(1) + \mu(1 - \eta(1)^2)\eta(2) \end{bmatrix}$$

3b

The three algorithms are implemented using one multi-step solver function.(Appendix) We use different parameters for each one of them. Since, Forward Euler is an explicit method, it does not use Newton solver. Whereas Backward Euler and Trapezoid methods use Newton solver. Both of them use same newton solver function. The written matlab codes are given in Appendix. An example convergence plot drawn from a trapezoid iteration step, demonstrating the quadratic convergence behaviour, is given in the figure 12 below:

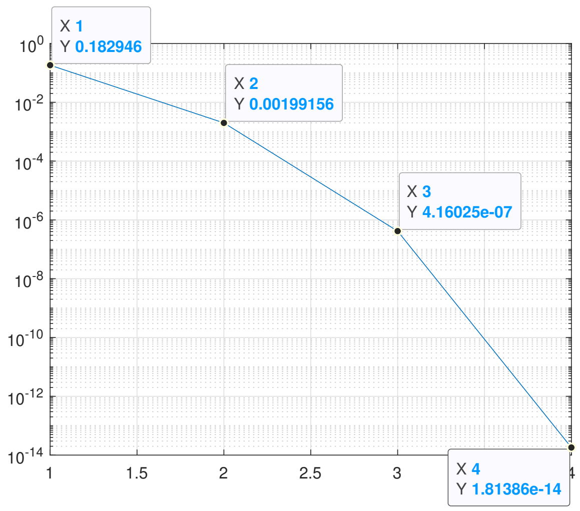

Figure 12: Quadratic Convergence Demonstration, P3b

3c

Three algorithms are plotted using $h = 1e-2$ for $\mu = 1$(Figure 13) and $\mu = 10$(Figure 14):

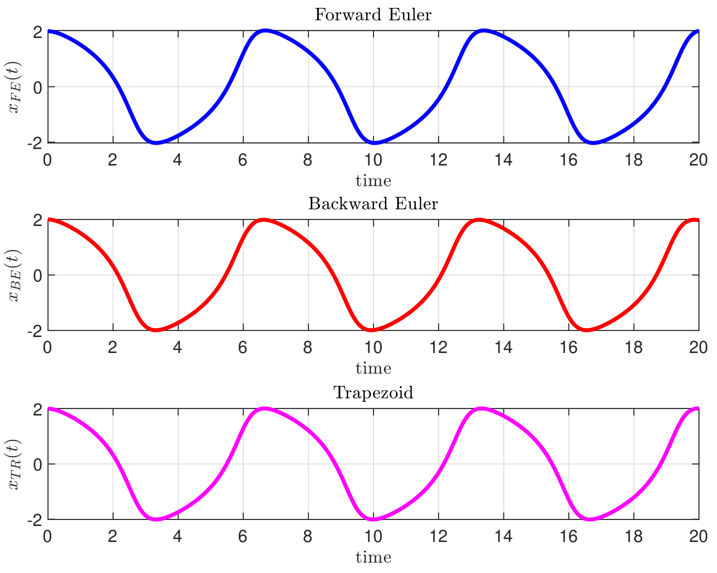

Figure 13: $\mu = 1, \ \ h = 0.01$, P3c

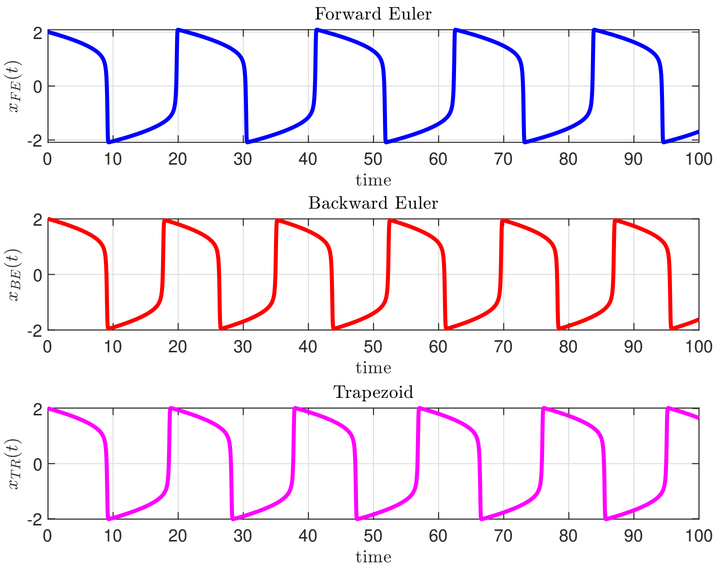

Figure 14: $\mu = 10, \ \ h = 0.01$, P3c

The numerical solution of Van-der-Pol oscillator in an unforced setting, reveals relaxation oscillation behavior. According to the obtained plots, all methods are stable. But, each creates different characteristics. Whereas forward euler method introduces a leading phase, the backward euler method introduces a lagging phase to the solution according to each other and trapezoidal method. The trapezoidal method results in-between solution and is known as an optimum scheme.(Dahlquist Second Barrier states that an A-stable multi step method cannot exceed order $p > 2$. And trapezoidal method has the smallest error constant.) The results indicate similar error accumulation characteristics with the test problem.

To investigate the effect of step size h on the convergence, we plot each solution where $h = 0.3$(Figure 15 and $h = 0.4$(Figure 16) together with optimum solution at $\mu = 2$ (as trapezoidal method solution $h = 0.001$). Experimenting

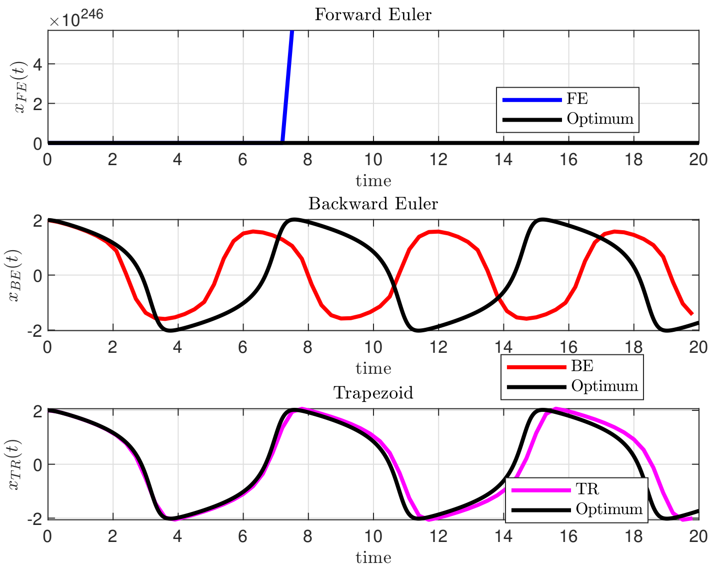

Figure 15: $\mu = 2,\ \ h = 0.3$, P3c

with different step sizes yiels $h = 0.3$ where forward euler method is divergent and useless, backward euler method is around marginal stability, but trapezoid rule preserves stability despite its error. As we further increase the step size, we observe that backward euler becomes unstable. Further investigate this behavior, we increase μ and so T_{max}, as $\mu = 100$ $T_{max} = 100$ with $h = 0.01$ Setting nonlinear damping ratio parameter $\mu = 100$ results slower oscillations, increases oscillation period, decreases frequency. This reveals that μ directly controls the oscillation behavior in relaxation oscillation. Given this setting, Forward Euler

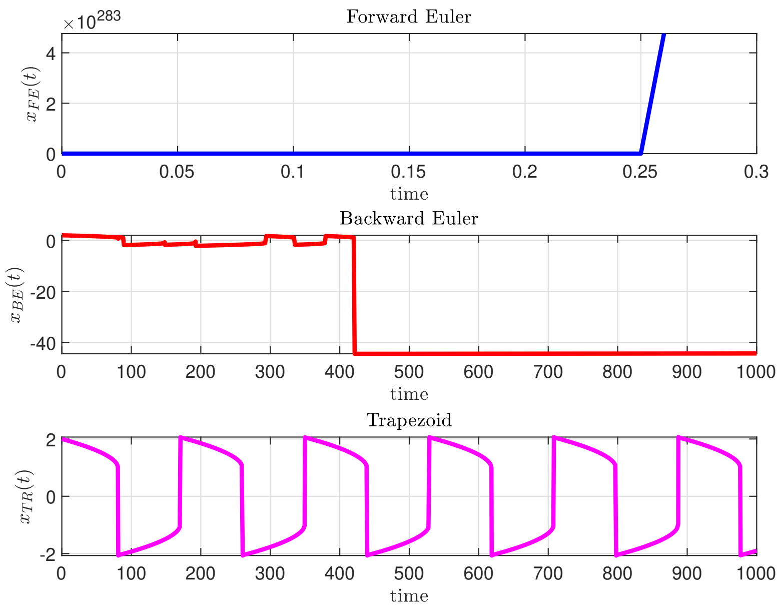

Figure 16: $\mu = 2, \;\; h = 0.01$, P3c

method immediately diverges. As expected, backward euler method is more stable and resists up to $t = 420$ and diverges later. Since, Trapezoidal method is A-stable, it preserves its stability. Among all, the Trapezoidal method is the only one useful. We can also infer from these experiments that μ is also effective on convergence behavior and stability. With same step size, Forward Euler method cannot track the system response.

Problem 4

4a

For each order k, we need to construct exactness linear equality system. Then, we try to solve maximum number of equations starting from $p = 0$. The optimum solution needs to solve all equations for $p \leq p^*$, ie at RHS, we have zeros for $p = 0, ...p^*$. First, an example linear system of equations for exactness is given for $k = 1$ and $k = 2$. Then, we present the script implemented for automatic computation of optimum coefficients and orders.

$k = 1$

For $k = 1$ first order method, we have $2k + 2$ variables as $\alpha_0 = 1$,$\alpha_1 = -1$ and due to explicitness $\beta_0 = 0$ and β_1(free parameter) and we need to solve below linear equation since we have 1 free parameter.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & -2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

$\beta_1 = 1$ satisfies $p = 0$ and $p = 1$. Hence, our solution is:

$$\{1,\ -1,\ 0,\ 1\} \rightarrow \textbf{Forward Euler Method}$$

$$\textbf{LTE} = C(\Delta t)^2$$

$k = 2$

Putting $\alpha_0 = 1$, $\alpha_1 = -1$, $\beta_0 = 0$(explicitness) reveals below linear system of equations to solve for 3 free parameters α_2, β_1, β_2:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & -2 & 0 \\ 0 & -3 & 0 \\ 0 & -4 & 0 \end{bmatrix} \begin{bmatrix} \alpha_2 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ -3 \\ -7 \\ -15 \end{bmatrix}$$

Since below zeros row, 3×3 matrix is full-rank, nonsingular. We have unique solution. Solving 3×3 results:

$$[\alpha_2\ \ \beta_1\ \ \beta_2] = [0\ \ 1.5\ \ -0.5]$$

We have coefficients:

$$\{1,\ -1,\ 0,\ 0,\ 1.5,\ -0.5\} \rightarrow \textbf{Adams-Bashforth, k=2}$$

Implemented script in order to find the coefficients and error order is presented below:

Listing 3: Problem 4a

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 HW4 Problem 4
%
% Ender Erkaya
%
% January 2022
%
clearvars
%% Take Input
k = input('Please enter k values 1,2,3,4  \n');
non_free_index = [1; 2];
coeffs = zeros(2*k+2,1);

M0   = zeros(2*k+1,2*k+2);
RHS0 = zeros(2*k+1,1);
for p = 0:2*k
    r = p+1;
    for c = 1:(k+1)
        M0(r,c)=(k-(c-1))^p;
        if (p==0)&&(c==k+1)
            M0(r,k+1+c)=0;
        else
            M0(r,k+1+c)=-p*(k-(c-1))^(p-1);
        end
    end
end

% Force \alpha_0 = 1;
RHS0 = RHS0 - M0(:,1);
coeffs(1)=1;

% Force \alpha_1 = -1
RHS0 = RHS0 + M0(:,2);
coeffs(2)=-1;

%%%%%% PART A %%%%%%%%%%%%%
%% Part a) force \beta0=0(explicit)
%% USE JUST FOR PART A
non_free_index(end+1)=k+2;
coeffs(k+2)=0;
%%%%%% PART A %%%%%%%%%%%%%
%% Find Coefficients
MA = M0;
```

```
if k>2
     non_free_index = vertcat(non_free_index,(4:k+1)');
     coeffs(4:k+1)  = 0;
end
MA(:,non_free_index)=[];

num_free = size(MA,2); % 2*k+2-length(non_free_index);

for it = num_free:size(MA,1)
     Matrix = MA(1:it,:);
     if rank(Matrix)==num_free
          break;
     end
end

solA    = Matrix \ RHS0(1:size(Matrix,1));

free_index = 1:2*k+2;
free_index(non_free_index)=[];

coeffs(free_index)=solA

% check and print LTE order of step size
% PART A
% order = find(abs(M0 * coeffs) > 1e-12, 1,'first')-1  %p*+1
% OR USE(PART B)
order = find(abs(M0 * coeffs) < 1e-12, 1,'last')  %p*+1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Outputs of the algorithm, coefficients and error orders are presented in the table below:

coeffs	α_4	α_3	α_2	α_1	α_0	β_0	β_1	β_2	β_3	β_4	LTE	Method
k=1	0	0	0	-1	1	0	1	0	0	0	**2**	**FE**
k=2	0	0	0	-1	1	0	1.5	-0.5	0	0	**3**	**A-B 2**
k=3	0	0	0	-1	1	0	1.9167	-1.3333	0.4167	0	**4**	**A-B 3**
k=4	0	0	0	-1	1	0	2.2917	-2.4583	1.5417	-0.3750	**5**	**A-B 4**

As summarized in the table above, we obtained an explicit algorithms with zeros $\alpha_{2,..} = 0$, optimized coefficients $\beta_{1,..}$s. The algorithm scheme is known as Adams Bashforth in the literature. Consequently, having set α_0 and α_1, searching optimum β values among explicit methods leads to Adams-Bashforth algorithms.

If we observe error values, for $LTE = C(\Delta t)^{p+1}$, we have global convergence as:

$$CT(\Delta t)^p$$

The obtained result is already stated by Dahlquist First Stability Barrier as: An explicit linear multi-step method with order of k cannot exceed convergence of order k.

4b

For part b, we just remove the explicitness condition forcing $\beta_0 = 0$ that results the coefficients summarized in below table.

coeffs	α_4	α_3	α_2	α_1	α_0	β_0	β_1	β_2	β_3	β_4	LTE	Method
k=1	0	0	0	-1	1	0.5	0.5	0	0	0	**3**	**TRAP**
k=2	0	0	0	-1	1	0.4167	0.6667	-0.0833	0	0	**4**	**A-M 2**
k=3	0	0	0	-1	1	0.3750	0.7917	-0.2083	0.0417	0	**5**	**A-M 3**
k=4	0	0	0	-1	1	0.3486	0.8972	-0.3667	0.1472	-0.0264	**6**	**A-M 4**

Having set α_0 and α_1, not constrained to explicitness, searching among optimum β values brings us to the algorithm scheme, known as Adams-Moulton in the literature. When $k = 1$, the result is well known Trapezoid method, as Adams-Moulton-1.

The resulting convergence orders as $p = k + 1$ is appropriate to the findings known as Dahlquist First Stability Barrier. If we would be searching among implicit methods without any constraints(removing $\alpha_1 = -1$), we would achieve an order of convergence $k+1$ for odd k, $k+2$ for even k, according to Dahlquist First Stability Barrier.

4c

For part b, we further remove α_1 constraint from the code and set $\beta_{1,\ldots} = 0$

coeffs	α_4	α_3	α_2	α_1	α_0	β_0	β_1	β_2	β_3	β_4	LTE	Method
k=1	0	0	0	-1	1	1	0	0	0	0	**2**	**BE**
k=2	0	0	0.3333	-1.3333	1	0.6667	0	0	0	0	**3**	**BDF-2**
k=3	0	-0.1818	0.8182	-1.6364	1	0.5455	0	0	0	0	**4**	**BDF-3**
k=4	0.1200	-0.6400	1.4400	-1.9200	1	0.4800	0	0	0	0	**5**	**BDF-4**

As we fix $\beta_{1,..} = 0$, searching optimum filter yields us to the filter schemes in the literature known as Backward Differentiation Formula. For $k = 1$, BDF is Backward Euler method. We have LTE with order of $k + 1$ of the step size. The global convergence is order of k. The written codes together with the modifications are given in the Appendix.

Problem 5

5a

The system equation corresponds to a function consisting of exponents of $\lambda\Delta t$ (ARMA process) that associates λ and step size h with absolute stability. Hence, we can analyze absolute stability with different λ and step sizes h. Applying z transform to the given difference equation corresponding to the multi-step scheme, gives us idea about eigenvalues, spectral coefficients of the system. The multistep formula for scalar test problem:

$$\sum_{j=0}^{k} \alpha_j v_{n-j} = \Delta t\lambda \sum_{j=0}^{k} \beta_j v_{n-j}$$

$$\sum_{j=0}^{k} (\alpha_j - \Delta t\lambda\beta_j) v_{n-j} = 0$$

Applying z transform brings:

$$\sum_{j=0}^{k} (\alpha_j - \Delta t\lambda\beta_j) z^{-j} = 0$$

Reconstructing as a polynomial:

$$\sum_{j=0}^{k} (\alpha_j - \Delta t\lambda\beta_j) z^{k-j} = 0$$

The roots of the above polynomial are the eigen-modes of the linear multistep process. In order to find the stability region, we need to find the λ values corresponding to the decision boundaries, z is on the unit circle. As described in the lecture, to do this z values are created on the unit circle, and we compute the corresponding λ values for the given multistep scheme. After boundary is found, using meshgrid each z value is evaluated and signed as stable or unstable for each mesh data point.

Listing 4: Problem 4a

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 HW4 Problem 5
%
% Ender Erkaya
%
% January 2022
%
clearvars
%% Take Input
k = input('Please enter k values 1,2,3,..  \n');
x = zeros(2*k+2,1);

a = input('Enter array elements [ ] around them \n');
x(1:length(a)) = a; %array coefficients
alpha = x(1:k+1);
beta  = x(k+2:2*k+2);

h = input('Step size?\n');

N = 1e3;
theta = linspace(0,2*pi,N);

zvals = exp(1i.*theta);

% Create Vandermonde Matrix
Zvan  = zeros(N,k+1);
for c = 1:k+1
    Zvan(:,c)=zvals.^(k-c+1);
end

% Form Equation
rhs = Zvan*alpha;
lhs = h*(Zvan*beta); % lambda coefficient
lambdas = rhs./lhs;

%% Create Visual
len = linspace(-2*max(abs(lambdas)),2*max(abs(lambdas)),500);
```

```
% r = 0:1e-2:2*max(abs(lambdas));
% theta = linspace(0,2*pi,1e3);
% [rr, tt] = meshgrid(r,theta);
% x = rr.*cos(tt);
% y = rr.*sin(tt);
[x, y] = meshgrid(len, len);

z = zeros(size(x));
for r = 1:size(x,1)
    for s = 1:size(y,2)
        temp = x(r,s) + y(r,s)*1i;

        % Determine if stable
        if max(abs(roots(alpha-temp*h*beta)))<1
            z(r,s)= -0.2;
        else
            z(r,s)= 1;
        end
    end
end

figure
contourf(x,y,z,'k','LineWidth',3);
hold on
line([min(len) max(len)],[0 0],'Color','black','LineStyle','--');
hold on
line([0 0],[min(len) max(len)],'Color','black','LineStyle','--')
grid on
caxis([-1, 1]);
colormap('hot');
title(strcat('Stability Region, h =',string(h)),'Interpreter','latex')
xlabel('Re($\lambda$)','Interpreter','latex');
ylabel('Im($\lambda$)','Interpreter','latex');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

5b

The obtained plots for each algorithm scheme described in 4a, 4b, 4c are given figures 17, 18, 19: The most common observation of all plot is that the stability region reduces as k increases. As we try to fasten the algorithm by increasing the order, the large time step stability is disturbed. Among Adams-Bashforth methods, there is no A-stable method. Among Adams-Moulton algorithms, the only A-stable method is Trapezoidal Rule, with order $k = 1$. Between three algorithmic schemes, Backward Differentiation rule is the most stable, ie largest

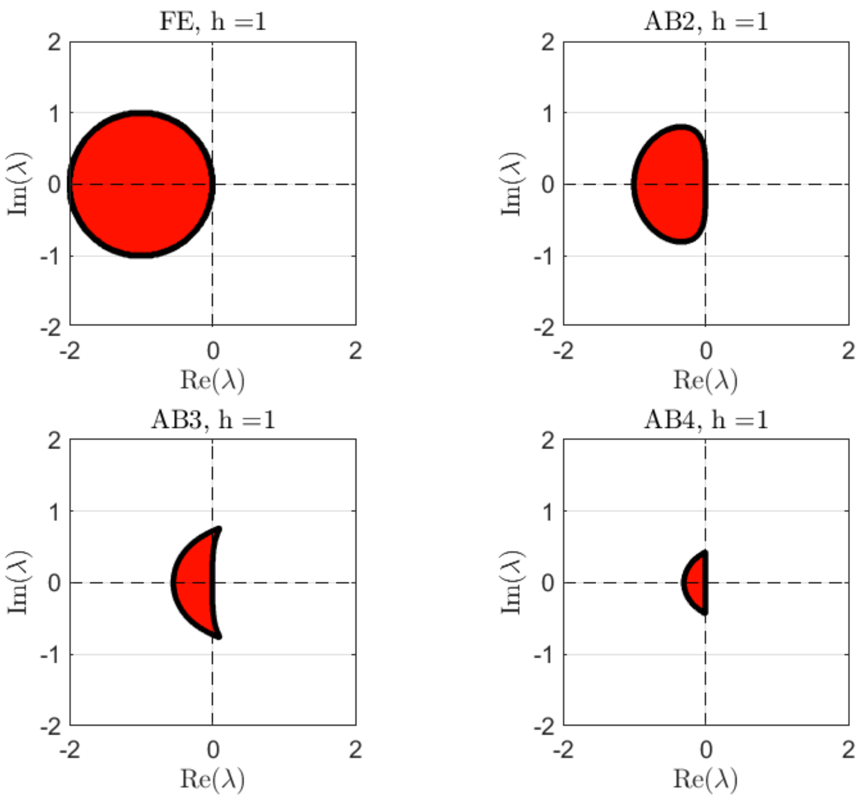

Figure 17: 4a AB Absolute Stability, $h = 1$, P5b

stability region. It is A-stable for both $k = 1$ and $k = 2$. After $k > 2$, absolute stability starts to be slightly corrupted. As stated by Dahlquist Second Stability Barrier, we do not observe an A-stable algorithm with $k > 2$.

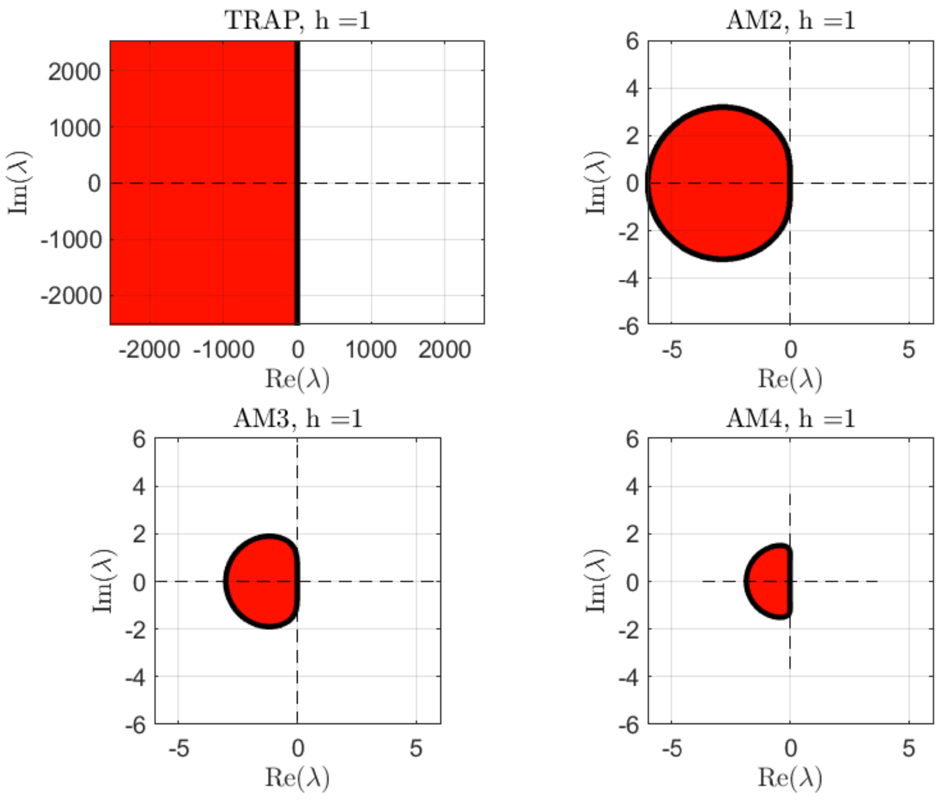

Figure 18: 4b AM Absolute Stability, $h = 1$, P5b

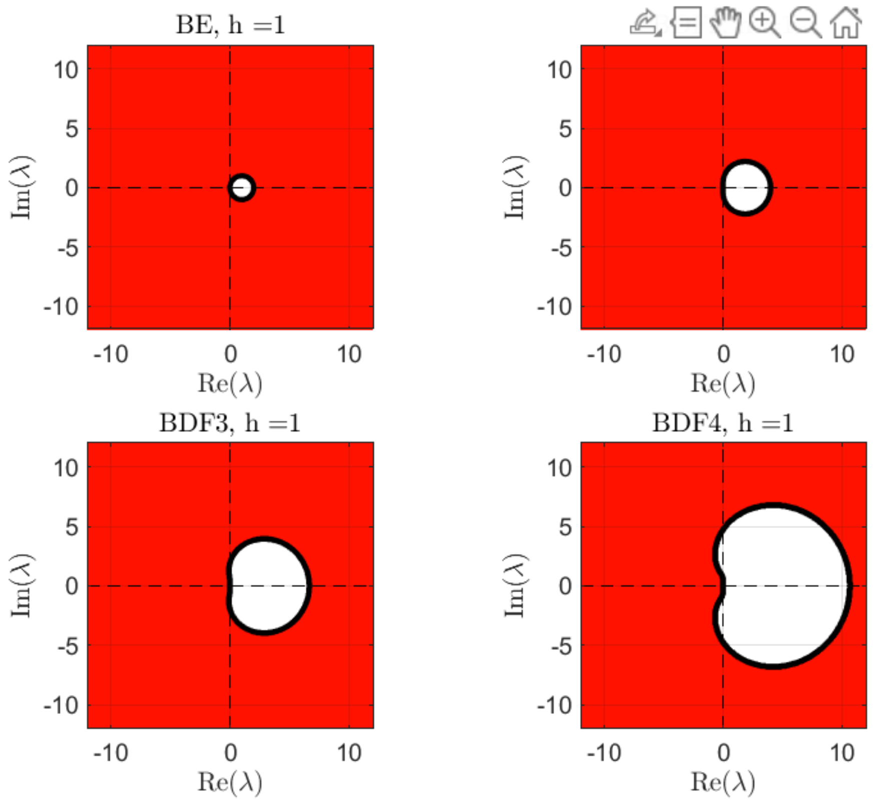

Figure 19: 4c BDF Absolute Stability, $h = 1$, P5b

Problem 6

6a

The solution of the above problem will in the space of $\{exp(-100t),\ cos(t),\ sin(t)\}$ as:

$$x^*(t) = a\ exp(-100t) + b\ cos(t) + c\ sin(t)$$

Putting into the equation yields:

$$b = 1,\ c = 0$$

: The generic form of the solution of the above problem is:

$$x^*(t) = a\ exp(-100t) + cos(t)$$

Incorporating the initial value information $x(0) = 1$:

$$x^*(t) = cos(t)$$

6b

The algorithm described in 4a, k=2, is implemented as below function:

Listing 5: Problem 6b

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xout,tout] = AB2(fun, Jfun, x0, h, Tmax)

%% Settings
alpha = [1 -1 0];
beta  = [0 1.5 -0.5];

%% Solve ODE
[xout,tout] = solve_ode(alpha,beta,x0,Tmax,h,fun,Jfun);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Trying to find the unstable boundary, $h = 1.07e-2$ and $h = 1e-2$ solutions are given as figure 20:

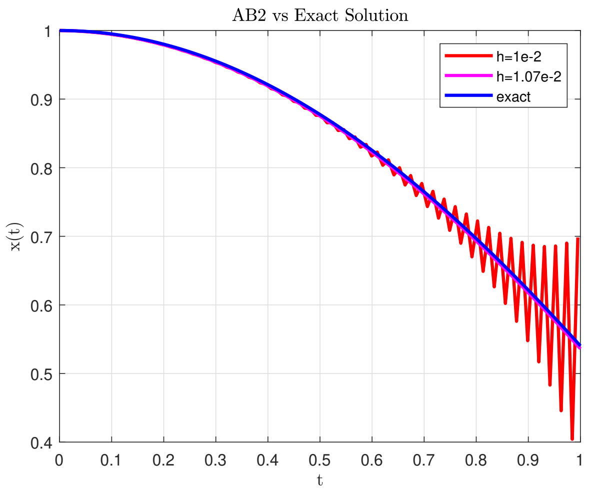

Figure 20: AB2 Solutions, P6b

6c

The BDF2 algorithm described in 4c, k=2, is implemented as below function:

Listing 6: Problem 6c

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xout,tout] = BDF2(fun, Jfun, x0, h, Tmax)

%% Settings
alpha = [1 -1.3333 0.3333];
beta  = [0.6667 0 0];

%% Solve ODE
[xout,tout] = solve_ode(alpha,beta,x0,Tmax,h,fun,Jfun)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

For $h = 5e-1$ and $h = 1$ solutions are given as figure 22: The algorithm is observed to be very stable. As we increase the step size h and T_{max}, its stability does not corrupt.

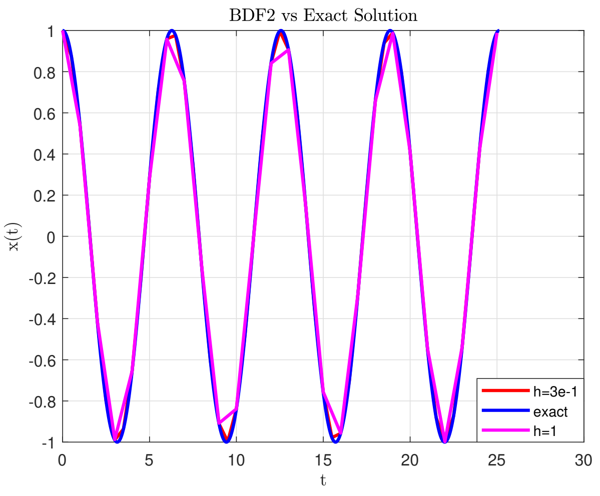

Figure 21: BDF2 Solutions, P6c

6d

For $h = \{0:2; 0:1; 0:05; 0:02; 0:01; 0:005; 0:001\}$, solutions are computed below:

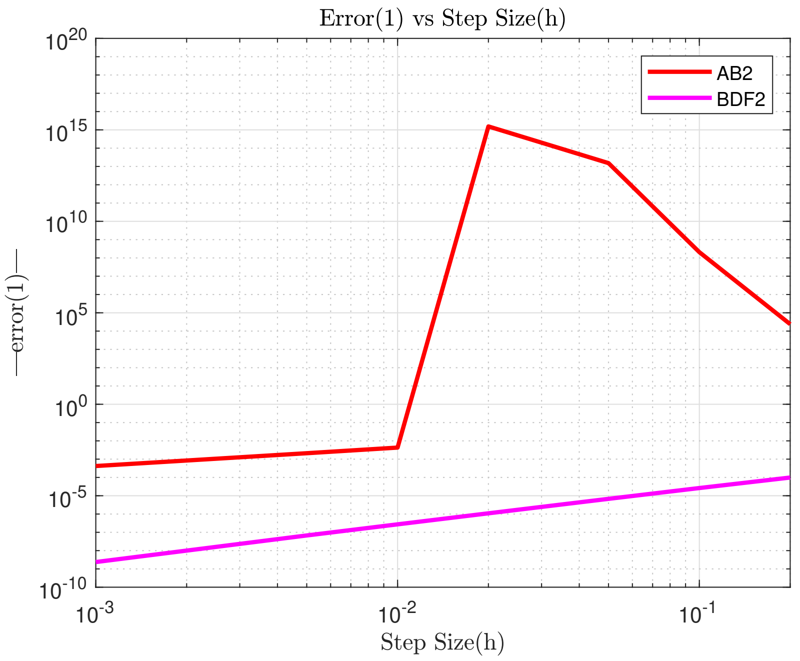

Figure 22: Solutions, P6d

6e

In the above plot, we observe that error at $t = 1$ of BDF2 algorithm depends on the step size with the order $\epsilon = Ch^2$. It is appropriate with our theoretical findings. Its stability is preserved at all values of h. However, the stability and usefullness of AB2 algorithm is highly dependent on the selected step size h. After h achieves 0.01, AB2 algorithm becomes unstable and useless. We observe that we have objective function consisting the component $\lambda x(t)$ with $\lambda = -100$. In stability analysis, we observe from our findings that stability of AB2 method corrupts when(figure 23):

$$\lambda < -\frac{1}{h}$$

that corresponds for $\lambda = -100$ to:

$$h > 0.01$$

. We observe this behavior of destruction of stability after $h > 0.01$ in the figure. Contrarily, since BDF2 algorithm is A-stable, we do not experience unstability for any $h > 0$ for the given λ.

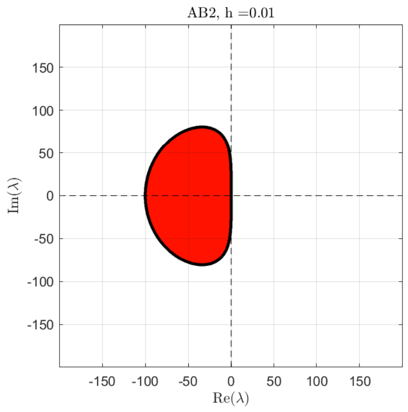

Figure 23: Solutions, P6e

References

https://en.wikipedia.org/wiki/Linear multistep method

https://cseweb.ucsd.edu/classes/wi13/cse245-b/papers/Dahlquist.pdf

Appendix

Problem 1

Listing 7: Problem1.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 - HW4 Problem 1
%
% Ender Erkaya
%
% January 2022
%
%% Settings
lambda = 1;
exact  = @(t) exp(lambda*t);
fun    = @(x) lambda*x;
alpha  = [1 0 -1];
beta   = [0 2 0];
x0     = 1;
Tmax   = 10;

%% Solve Differential Equation
[x1,t1] = solve_ode(alpha,beta,x0,Tmax,2,lambda,fun);
[x2,t2] = solve_ode(alpha,beta,x0,Tmax,0.5,lambda,fun);
[x3,t3] = solve_ode(alpha,beta,x0,Tmax,0.1,lambda,fun);

%% Figure
figure
subplot(3,1,1)
plot(t1,x1,'b','LineWidth',2)
hold on
plot(t1,exact(t1),'r','LineWidth',2);
grid on
legend('h=2','Exact')

subplot(3,1,2)
semilogy(t2,x2,'b','LineWidth',2)
hold on
semilogy(t2,exact(t2),'r','LineWidth',2);
grid on
legend('h=0.5','Exact')

subplot(3,1,3)
semilogy(t3,x3,'b','LineWidth',2)
hold on
```

```
semilogy(t3,exact(t3),'r','LineWidth',2);
grid on
legend('h=0.1','Exact')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 8: solve$_ode.m$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x,t] = solve_ode(alpha,beta,x0,Tmax,h,lambda,fun)
%
% Initializations
alpha = alpha./alpha(1); % normalize alpha
alpha = reshape(alpha,1,length(alpha));
beta  = reshape(beta,1,length(beta));
k     = length(alpha)-1;
t = 0:h:Tmax;

num_steps = length(t);
x = zeros(length(x0),num_steps);
x(1)= x0;
% Initialize with BE
x(2)= (1/(1-h*lambda))*x(1);

if beta(1)==0 % If Explicit Method
    for i = k+1:num_steps
        x(:,i) = -x(:,i-1:-1:i-k)*alpha(2:end)'+h*fun(x(:,i-1:-1:i-length(beta)+
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Problem 2

Listing 9: Problem2.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 HW4 Problem 2 Main
%
% Ender Erkaya
%
% January 2022
%
clearvars
clearvars -global
create_heat_data;
file = "heat_data.txt";
```

```
readckt;
loadMatrix;

M    = -sparse(diag(diag(C).^(-1)))*sparse(G);
fun  = @(v,u) M*v;
Jfun = @(v) M;
x0   = z;

%% Forward Euler Algorithm
Tmax = 10;
h    = 1e-4; %time step

[vFE,tFE] = forward_euler(fun, x0, h, Tmax);
vopt = vFE(:,end);
colors = {'k','g','c','y','m','r','b'};
tpart  = floor(size(tFE,2)/5);
figure
for n = 1:6
    hold on
    plot(delta_x:delta_x:1,vFE(:,tpart*(n-1)+1),colors{n},'LineWidth',1.5);
end
grid on
legend('t=0','t=2','t=4','t=6','t=8','t=10','Interpreter','latex');
xlabel('Position x','Interpreter','latex');
ylabel('Temperature','Interpreter','latex');

HFE = [0.001;0.002;0.003;0.004;0.005;0.006];
for n = 1:length(HFE)
    h = HFE(n);
    [vFE,tFE] = forward_euler(fun, x0, h, Tmax);
    hold on
    errorFE(n) = norm(vFE(:,end)-vopt);
end

figure
loglog(HFE,errorFE,colors{n},'LineWidth',1.5);
grid on
xlabel('Step Size(h)','Interpreter','latex');
ylabel('|error(-FE(1e-4))|','Interpreter','latex');
title('Forward Euler Error vs Step Size','Interpreter','latex');

%% Backward Euler Algorithm
Tmax = 10;
h    = 1e-4; %time step

[vBE,tBE] = backward_euler(fun,Jfun, x0, h, Tmax);
```

```
vopt = vBE(:,end);
tpart  = floor(size(tBE,2)/5);
figure
for n = 1:6
    hold on
    plot(delta_x:delta_x:1,vBE(:,tpart*(n-1)+1),colors{n},'LineWidth',1.5);
end
grid on
legend('t=0','t=2','t=4','t=6','t=8','t=10','Interpreter','latex');
xlabel('Position x','Interpreter','latex');
ylabel('Temperature','Interpreter','latex');
title('Backward Euler','Interpreter','latex');

HBE = [1e-3;5e-3;1e-2;5e-2;1e-1;5e-1;1];
for n = 1:length(HBE)
    h = HBE(n);
    [vBE,tBE] = backward_euler(fun,Jfun, x0, h, Tmax);
    hold on
    errorBE(n) = norm(vBE(:,end)-vopt);
end

%% BDF2
Tmax = 10;
h    = 1e-4; %time step

[vBDF2,tBDF2] = BDF2(fun,Jfun, x0, h, Tmax);
vopt = vBDF2(:,end);
tpart  = floor(size(tBDF2,2)/5);
figure
for n = 1:6
    hold on
    plot(delta_x:delta_x:1,vBDF2(:,tpart*(n-1)+1),colors{n},'LineWidth',1.5);
end
grid on
legend('t=0','t=2','t=4','t=6','t=8','t=10','Interpreter','latex');
xlabel('Position x','Interpreter','latex');
ylabel('Temperature','Interpreter','latex');
title('BDF2','Interpreter','latex');

HBDF2 = [1e-3;5e-3;1e-2;5e-2;1e-1;5e-1;1];
for n = 1:length(HBDF2)
    h = HBDF2(n);
    [vBDF2,tBDF2] = BDF2(fun,Jfun, x0, h, Tmax);
    hold on
    errorBDF2(n) = norm(vBDF2(:,end)-vopt);
end
```

```
figure
loglog(HBDF2,errorBDF2,colors{6},'LineWidth',1.5);
hold on
loglog(HBE,errorBE,colors{7},'LineWidth',1.5);
hold on
loglog(HFE,errorFE,colors{5},'LineWidth',1.5);
grid on
xlabel('Step Size(h)','Interpreter','latex');
ylabel('|error|','Interpreter','latex');
title('Error vs Step Size','Interpreter','latex');
legend('BDF2','BE','FE');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 10: readckt.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Global variables to store the elements
clear Resistors Isources Vsources Csources Capacitors Initials;
global Resistors Isources Vsources Csources Capacitors Initials;

%Global variables to store the circuit matrix, right-hand side
clear Matrix RHS x C z;
global Matrix RHS x C z;

%Global counters for number of nodes and number of nodes with sources
clear Nodes  sourcenodes;
global Nodes  sourcenodes;

%Global counters for number of branches
clear Branches;
global Branches;

fid = fopen(file, 'r');

% Global vector to store the resistors.
Resistors = [];

% Global vector to store current sources.
Isources = [];

% Global vector to store voltage sources.
Vsources = [];

% Global vector to store voltage-controlled current sources.
Csources = [];
```

```
% Global vector to store capacitors
Capacitors = [];

% Global vector to store initial values
Initials = [];

% Counter which keeps track of the maximum node number in the circuit
maxnode = 0;

% Read the file, line by line, and dispatch based on first character on line.
while 1
  line = fgetl(fid);
  if ~isstr(line), break, end
  if length(line) ~= 0
    if(strcmp(line(1),'r') > 0)
      [Resistors,maxnode] = readelement(line, Resistors, 2, maxnode);
    elseif(strcmp(line(1),'i') > 0)
      [Isources,maxnode] = readelement(line, Isources, 2, maxnode);
    elseif(strcmp(line(1),'v') > 0)
      [Vsources,maxnode] = readelement(line, Vsources, 2, maxnode);
    elseif(strcmp(line(1),'x') > 0)
      [Csources,maxnode] = readelement(line, Csources, 4, maxnode);
    elseif(strcmp(line(1),'c') > 0)
      [Capacitors,maxnode] = readelement(line, Capacitors, 2, maxnode);
    elseif(strcmp(line(1),'z') > 0)
      [Initials,maxnode] = readelement(line, Initials, 1, maxnode);
    end
  end
end

% Force allocate matrix of size NxN, where N is equal to the maximum node
% number.
Matrix(maxnode,maxnode) = 0;
RHS(maxnode) = 0;
x(maxnode) = 0;
sourcenodes(maxnode) = 0;
C(maxnode,maxnode) = 0;
z(maxnode) = 0;

% Create an array of indicating voltage source nodes and values (zero means the
% node is not a voltage source node).
for i = 1:size(Vsources,1)
 n1 = Vsources(i,2);
 n2 = Vsources(i,3);
 if ((n1 == 0) & (n2 ~= 0))
   sourcenodes(n2) = -Vsources(i,1);
  elseif (n1 ~= 0) & (n2 == 0)
    sourcenodes(n1) = Vsources(i,1);
  end
end

% Global Circuit Description

Branches = size(Capacitors,1) + size(Resistors,1) + size(Isources,1)+size(Vsourc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 11: loadMatrix.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Zero the matrix and the rhs.
Matrix = 0 * Matrix;
RHS = 0 * RHS;
C   = 0 * C;
z   = 0 * z;

% Load the resistors.
for i = 1:size(Resistors,1)
     n1 = Resistors(i,2);
     n2 = Resistors(i,3);
     res = Resistors(i,1);
     if n1 > 0
          Matrix(n1, n1) = Matrix(n1,n1) + 1/res;
     end
     if n2 > 0
          Matrix(n2, n2) = Matrix(n2,n2) + 1/res;
     end
     if ((n1>0) && (n2>0))
          Matrix(n1, n2) = Matrix(n1,n2) - 1/res;
          Matrix(n2, n1) = Matrix(n2,n1) - 1/res;
     end
end

% Load the capacitors.
for i = 1:size(Capacitors,1)
     n1 = Capacitors(i,2);
     n2 = Capacitors(i,3);
     cap = Capacitors(i,1);
     if ((n1>0) && (n2>0))
          C(n1, n2) = C(n1,n2) - cap;
          C(n2, n1) = C(n2,n1) - cap;
     elseif (n1 > 0)
          C(n1, n1) = C(n1,n1) + cap;
     elseif (n2 > 0)
```

```
            C(n2, n2) = C(n2,n2) + cap;
        end
    end

    % Load the initial values
    for i = 1:size(Initials,1)
        n1   = Initials(i,2);
        zval = Initials(i,1);
        if (n1 > 0)
            z(n1)=zval;
        end
    end

    % Load the VCCSs
    for i = 1:size(Csources,1)
        n1 = Csources(i,2);
        n2 = Csources(i,3);
        n3 = Csources(i,4);
        n4 = Csources(i,5);
        gm = Csources(i,1);
        if ((n1>0) && (n3 > 0))
            Matrix(n1, n3) = Matrix(n1,n3) + gm;
        end
        if ((n1>0) && (n4 > 0) )
            Matrix(n1, n4) = Matrix(n1,n4) - gm;
        end
        if ((n2>0) && (n3 > 0))
            Matrix(n2, n3) = Matrix(n2,n3) - gm;
        end
        if ((n2>0) && (n4 > 0))
            Matrix(n2, n4) = Matrix(n2,n4) + gm;
        end
    end

    % Load the current sources.
    for i = 1:size(Isources,1)
        n1 = Isources(i,2);
        n2 = Isources(i,3);
        is = Isources(i,1);
        if (n1 > 0)
            RHS(n1) = RHS(n1) - is;
        end
        if (n2 > 0)
            RHS(n2) = RHS(n2) + is;
        end
    end
```

```
original_RHS = RHS;
% Load the voltage sources
nonvs_node_index = find(sourcenodes==0);
for k = 1:length(sourcenodes) % for each nonvs node
    if sourcenodes(k)==0
        for i = 1:size(Vsources,1)
            n1 = Vsources(i,2);
            n2 = Vsources(i,3);
            vs = Vsources(i,1);
            if (n1*n2) == 0 %check if satisfies condition
                % force vn1 = vs add to right hand side
                if (n1 > 0) % force vn1 = vi
                    RHS(k) = RHS(k) - Matrix(k,n1)*(vs);
                end
                if (n2 > 0) % force vn2 = -vi
                    RHS(k) = RHS(k) - Matrix(k,n2)*(-vs);
                end
            end
        end
    end
end
original_Matrix = Matrix;
Matrix = Matrix(nonvs_node_index,nonvs_node_index);
RHS    = RHS(nonvs_node_index);
z      = z(nonvs_node_index);
C      = C(nonvs_node_index,nonvs_node_index);

% Eliminate any last rows with purely zeros.
realLength = length(RHS);
quit = 1;
while(quit > 0)
    if(norm(Matrix(realLength,:)) == 0)
        realLength = realLength - 1;
    else
        quit = 0;
    end
end

% Put a one on the diagonal of any purely zero row in Matrix.
for i = 1:length(RHS)
    if(norm(Matrix(i,:)) == 0)
        Matrix(i,i) = 1;
    end
end
```

```
% b and G contain the reduced matrices without the extra zero rows.
b = RHS(1:realLength)';
G = Matrix(1:realLength, 1:realLength);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 12: create$_heat_data.m$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% create heat data
%
% Settings
N         = 100;

delta_x     = 1/N;
res_vals    = delta_x;
cap_vals    = 1;
in_vals     = zeros(N,1);
in_vals(floor(N*0.3):ceil(N*0.7)) = 1;
num_vs      = 1;
vs          = 0;

fileID = fopen('heat_data.txt','w');
% Write Resistors
for it = 1:N-1
    fprintf(fileID,'r%i %i %i %.2f \n',[it it it+1 res_vals]);
end
fprintf(fileID,'r%i %i %i %.2f \n',[N+1 1 0 res_vals]);
fprintf(fileID,'r%i %i %i %.2f \n',[N+2 N 0 res_vals]);
fprintf(fileID,'\n');
% Write Capacitors
for it = 1:N
    fprintf(fileID,'c%i %i %i %.2f \n',[it it 0 cap_vals]);
end
fprintf(fileID,'\n');
% Write Initials
for it = 1:N
    fprintf(fileID,'z%i %i %.2f \n',[it it in_vals(it)]);
end
fprintf(fileID,'\n');
% fprintf(fileID,'v%i %i %i %.2f \n',[1 N 0 0]); % boundary condition T(1)=0;
fclose(fileID);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 13: solve$_ode.m$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [x,t,counter,dx] = solve_ode(alpha, beta, x0, Tmax, h, fun, varargin)
%
if nargin > 6
    Jf = varargin{1};
end

%% Settings
eps = 1e-8;
max_iter = 1e1;

% Initializations
alpha = alpha./alpha(1); % normalize alpha
alpha = reshape(alpha,1,length(alpha));
beta  = reshape(beta,1,length(beta));
k     = length(alpha)-1;
t = 0:h:Tmax;

num_steps = length(t);
n = length(x0);
x = zeros(n,num_steps);
x(:,1)= x0;

%% Initialize with Backward Euler
if k > 1
    x(:,1:k) = backward_euler(fun,Jf,x0,h, h*(k-1));
end

if beta(1)==0 % If Explicit Method
    for i = k+1:num_steps
        u      = t(i);
        x(:,i) = -x(:,i-1:-1:i-k)*alpha(2:end)' + h * fun(x(:,i-1:-1:i-length(be
    end
else
    % Implicit Method
    % Construct Jacobian and RHS
    for i = k+1:num_steps
        u = t(i);
        b = x(:,i-1:-1:i-k)*alpha(2:end)' - h * fun(x(:,i-1:-1:i-length(beta)+1)
        Jacobianfun = @(x) eye(n) - h * beta(1) * Jf(x);
        RHSfun      = @(x) (x - h * beta(1) * fun(x,u) + b);
        [x(:,i),counter,dx] = solvenewton(RHSfun, Jacobianfun, x(:,i-1), max_ite
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 14: backward$_euler.m$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xout,tout] = backward_euler(fun, Jfun, x0, h, Tmax)

%% Settings
alpha = [1 -1];
beta  = [1  0];

%% Solve ODE
[xout,tout] = solve_ode(alpha, beta, x0, Tmax, h, fun, Jfun);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 15: forward$_euler.m$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xout,tout] = forward_euler(fun, x0, h, Tmax)

%% Settings
alpha = [1 -1];
beta  = [0 1];

%% Solve ODE
[xout,tout] = solve_ode(alpha,beta,x0,Tmax,h,fun);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 16: BDF2.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xout,tout] = BDF2(fun, Jfun, x0, h, Tmax)

%% Settings
alpha = [1 -1.3333 0.3333];
beta  = [0.6667 0 0];

%% Solve ODE
[xout,tout] = solve_ode(alpha,beta,x0,Tmax,h,fun,Jfun);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 17: solvenewton.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xval,counter,dx] = solvenewton(fun, Jfun, xinit, max_iter, eps)

% Settings
% eps       = 1e-8;

% Initializations
```

```
counter  = 0;
xval     = xinit;
dx = [];
% Newton Algorithm
while (counter < max_iter)
    counter = counter + 1;

    % Newton Update
    funval  = fun(xval); % evaluate F(x)
    JFt     = Jfun(xval); % evaluate JacobianF(x)
    deltax  = -JFt \ funval;
    xval    = xval + deltax;

%      xxplot  = [xxplot xval];
    dx       = [dx deltax];
    % Convergence Check
    if (norm(deltax) < (eps * norm(xval))) && (norm(funval) < eps)
        break;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Problem 3

Listing 18: Problem3.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 Problem 3
%
% Ender Erkaya
%
% January 2022
%
% load xTR_mu2.mat;
% xopt = xTR;
%% Settings
h  = 1e-2;
mu = 100;
x0 = [2; 0];
Tmax  =  max(20, 10*mu);

%% Function Definitions
%
funx  = @(x) [x(2); -x(1)+ mu*(1-x(1)^2)*x(2)];
```

```
Jfunx = @(x) [0 1;-1-2*mu*x(2)*x(1) mu*(1-x(1)^2)];

% Forward Euler
[xFE,tFE] = forward_euler(funx, x0, h, Tmax);
[xBE,tBE] = backward_euler(funx, Jfunx, x0, h, Tmax);
[xTR,tTR,counter,dx] = trapezoid(funx, Jfunx, x0, h, Tmax);
% Backward Euler

%% Visualization
figure
subplot(3,1,1);
plot(tFE,xFE(1,:),'b', 'LineWidth',2);
% hold on
% plot(0:1e-3:20,xopt,'k', 'LineWidth',2);
grid on;
title('Forward Euler','Interpreter','latex');
xlabel('time','Interpreter','latex');
ylabel('${x_{FE}}(t)$','Interpreter','latex');
legend('FE','Optimum','Interpreter','latex');

subplot(3,1,2)
plot(tBE,xBE(1,:),'r', 'LineWidth',2);
% hold on
% plot(0:1e-3:20,xopt,'k', 'LineWidth',2);
grid on
title('Backward Euler','Interpreter','latex')
xlabel('time','Interpreter','latex');
ylabel('${x_{BE}}(t)$','Interpreter','latex');
legend('BE','Optimum','Interpreter','latex');

subplot(3,1,3)
plot(tTR,xTR(1,:),'m', 'LineWidth',2);
% hold on
% plot(0:1e-3:20,xopt,'k', 'LineWidth',2);
grid on
title('Trapezoid','Interpreter','latex')
xlabel('time','Interpreter','latex');
ylabel('${x_{TR}}(t)$','Interpreter','latex');
legend('TR','Optimum','Interpreter','latex');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 19: trapezoid.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xout,tout,counter,dx] = trapezoid(fun, Jfun, x0, h, Tmax)
```

```
%% Settings
alpha = [1  -1];
beta  = [0.5  0.5];

%% Solve ODE
[xout,tout,counter,dx] = solve_ode(alpha, beta, x0, Tmax, h, fun, Jfun);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Problem4

Listing 20: Problem4a.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 HW4 Problem 4
%
% Ender Erkaya
%
% January 2022
%
clearvars
%% Take Input
k = input('Please enter k values 1,2,3,4  \n');
non_free_index = [1; 2];
coeffs = zeros(2*k+2,1);

M0   = zeros(2*k+1,2*k+2);
RHS0 = zeros(2*k+1,1);
for p = 0:2*k
    r = p+1;
    for c = 1:(k+1)
        M0(r,c)=(k-(c-1))^p;
        if (p==0)&&(c==k+1)
            M0(r,k+1+c)=0;
        else
            M0(r,k+1+c)=-p*(k-(c-1))^(p-1);
        end
    end
end

% Force \alpha_0 = 1;
RHS0 = RHS0 - M0(:,1);
coeffs(1)=1;

% Force \alpha_1 = -1
```

```
RHS0 = RHS0 + M0(:,2);
coeffs(2)=-1;

%% Part a) force \beta0=0(explicit)
MA = M0;
non_free_index(end+1)=k+2;
coeffs(k+2)=0;

if k>2
    non_free_index = vertcat(non_free_index,(4:k+1)');
    coeffs(4:k+1)  = 0;
end
MA(:,non_free_index)=[];

num_free = size(MA,2); % 2*k+2-length(non_free_index);

for it = num_free:size(MA,1)
    Matrix = MA(1:it,:);
    if rank(Matrix)==num_free
        break;
    end
end

solA   = Matrix \ RHS0(1:size(Matrix,1));

free_index = 1:2*k+2;
free_index(non_free_index)=[];

coeffs(free_index)=solA

% check and print LTE order of step size
order = find(abs(M0 * coeffs)>1e-14,1,'first')-1  %p*+1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 21: Problem4b.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 HW4 Problem 4
%
% Ender Erkaya
%
% January 2022
%
clearvars
%% Take Input
k = input('Please enter k values 1,2,3,4  \n');
non_free_index = [1; 2];
```

```
coeffs = zeros(2*k+2,1);

M0    = zeros(2*k+1,2*k+2);
RHS0 = zeros(2*k+1,1);
for p = 0:2*k
    r = p+1;
    for c = 1:(k+1)
        M0(r,c)=(k-(c-1))^p;
        if (p==0)&&(c==k+1)
            M0(r,k+1+c)=0;
        else
            M0(r,k+1+c)=-p*(k-(c-1))^(p-1);
        end
    end
end

% Force \alpha_0 = 1;
RHS0 = RHS0 - M0(:,1);
coeffs(1)=1;

% Force \alpha_1 = -1
RHS0 = RHS0 + M0(:,2);
coeffs(2)=-1;
MA = M0;

if k>2
    non_free_index = vertcat(non_free_index,(4:k+1)');
    coeffs(4:k+1)   = 0;
end
MA(:,non_free_index)=[];

num_free = size(MA,2); % 2*k+2-length(non_free_index);

for it = num_free:size(MA,1)
    Matrix = MA(1:it,:);
    if rank(Matrix)==num_free
        break;
    end
end

solA    = Matrix \ RHS0(1:size(Matrix,1));

free_index = 1:2*k+2;
free_index(non_free_index)=[];

coeffs(free_index)=solA
```

```
% check and print LTE order of step size
order = find(abs(M0 * coeffs)<1e-12,1,'last')   %p*+1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 22: Problem4c.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 HW4 Problem 4
%
% Ender Erkaya
%
% January 2022
%
clearvars
%% Take Input
k = input('Please enter k values 1,2,3,4  \n');
non_free_index = 1;
coeffs = zeros(2*k+2,1);

M0    = zeros(2*k+1,2*k+2);
RHS0 = zeros(2*k+1,1);
for p = 0:2*k
    r = p+1;
    for c = 1:(k+1)
        M0(r,c)=(k-(c-1))^p;
        if (p==0)&&(c==k+1)
            M0(r,k+1+c)=0;
        else
            M0(r,k+1+c)=-p*(k-(c-1))^(p-1);
        end
    end
end

% Force \alpha_0 = 1;
RHS0 = RHS0 - M0(:,1);
coeffs(1)=1;
MA = M0;

if k>0
    non_free_index = vertcat(non_free_index,(k+3:2*k+2)');
    coeffs(k+3:2*k+2)   = 0;
end
MA(:,non_free_index)=[];

num_free = size(MA,2); % 2*k+2-length(non_free_index);
```

```
for it = num_free:size(MA,1)
    Matrix = MA(1:it ,:);
    if rank(Matrix)==num_free
        break;
    end
end

solA    = Matrix \ RHS0(1:size(Matrix,1));

free_index = 1:2*k+2;
free_index(non_free_index)=[];

coeffs(free_index)=solA

% check and print LTE order of step size
order = find(abs(M0 * coeffs)<1e-12,1,'last')  %p*+1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Problem5

Listing 23: Problem5.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 HW4 Problem 5
%
% Ender Erkaya
%
% January 2022
%
%% Take Input
% k = input('Please enter k values 1,2,3,..  \n');
x = zeros(2*k+2,1);

% a = input('Enter array elements [ ] around them \n');
x(1:length(a)) = a; %array coefficients
alpha = x(1:k+1);
beta  = x(k+2:2*k+2);

% h = input('Step size?\n');

N = 1e3;
theta = linspace(0,2*pi,N);

zvals = exp(1i.*theta);
```

```
% Create Vandermonde Matrix
Zvan   = zeros(N,k+1);
for c = 1:k+1
    Zvan(:,c)=zvals.^(k-c+1);
end

% Form Equation
rhs = Zvan*alpha;
lhs = h*(Zvan*beta); % lambda coefficient
lambdas = rhs./lhs;

%% Create Visual
len = linspace(-2*max(abs(lambdas)),2*max(abs(lambdas)),1000);
% r = 0:1e-2:2*max(abs(lambdas));
% theta = linspace(0,2*pi,1e3);
% [rr, tt] = meshgrid(r,theta);
% x = rr.*cos(tt);
% y = rr.*sin(tt);
[x, y] = meshgrid(len, len);

z = zeros(size(x));
for r = 1:size(x,1)
    r
    for s = 1:size(y,2)
        temp = x(r,s) + y(r,s)*1i;

        % Determine if stable
        if max(abs(roots(alpha-temp*h*beta)))<1
            z(r,s)= -0.2;
        else
            z(r,s)= 1;
        end
    end
end

contourf(x,y,z,'k','LineWidth',2);
axis equal
hold on
line([min(len) max(len)],[0 0],'Color','black','LineStyle','--');
hold on
line([0 0],[min(len) max(len)],'Color','black','LineStyle','--')
grid on
caxis([-1, 1]);
colormap('hot');
% title(strcat('Stability Region, h =',string(h)),'Interpreter','latex')
```

```
xlabel('Re($\lambda$)','Interpreter','latex');
ylabel('Im($\lambda$)','Interpreter','latex');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 24: Plotting.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
figure
subplot(2,2,1)
k = 1;
a = [1 -1 0.5 0.5];
h = 1;
Problem5;
title(strcat('TRAP, h =',string(h)),'Interpreter','latex')
hold off
subplot(2,2,2)
k = 2;
a = [1 -1 0 0 1.5 -0.5];
h = 1e-2;
Problem5
title(strcat('AB2, h =',string(h)),'Interpreter','latex')
hold off
subplot(2,2,3)
k = 3;
a = [1 -1 0 0 0.3750 0.7917 -0.2083 0.0417];
Problem5
title(strcat('AM3, h =',string(h)),'Interpreter','latex')
hold off
subplot(2,2,4)
k = 4;
a = [1 -1 0 0 0 0.3486 0.8972 -0.3667 0.1472 -0.0264];
Problem5
title(strcat('AM4, h =',string(h)),'Interpreter','latex')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Problem6

Listing 25: Problem6.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ELEC 518 Problem 6
%
% Ender Erkaya
%
% January 2022
```

```
%
clearvars
%% Settings
h     = 1e-2;
x0    = 1;

%% Function Handles
xorig = @(u) cos(u);
fun   = @(x,u) -100*(x-cos(u))-sin(u);
Jfun  = @(x) -100;

%% 6b
Tmax  = 1;
figure
[xAB2,tAB2] = AB2(fun, Jfun, x0, 1e-2, Tmax);
hold on
plot(tAB2,xAB2,'m','LineWidth',2);
hold on
plot(tAB2,xorig(tAB2),'b','LineWidth',2);
[xAB2,tAB2] = AB2(fun, Jfun, x0, 1.07e-2, Tmax);
hold on
plot(tAB2,xAB2,'r','LineWidth',2);
grid on
legend('h=1e-2','exact','h=1.07e-2')
xlabel('t','Interpreter','latex');
ylabel('x(t)','Interpreter','latex');
title('AB2 vs Exact Solution','Interpreter','latex');

%% 6c
Tmax = 3*2*pi;
figure
[xBDF2,tBDF2] = BDF2(fun, Jfun, x0, 5e-1, Tmax);
plot(tBDF2,xBDF2,'r','LineWidth',2);
hold on
plot(0:1e-2:Tmax,xorig(0:1e-2:Tmax),'b','LineWidth',2);
[xBDF2,tBDF2] = BDF2(fun, Jfun, x0, 1, Tmax);
hold on
plot(tBDF2,xBDF2,'m','LineWidth',2);
grid on
legend('h=3e-1','exact','h=1')
xlabel('t','Interpreter','latex');
ylabel('x(t)','Interpreter','latex');
title('BDF2 vs Exact Solution','Interpreter','latex');

%% 6d
H     = [0.2; 0.1; 0.05; 0.02; 0.01; 0.005; 0.001];
```

```
Tmax = 1;
for it = 1:length(H)
    h = H(it);
    [xAB2,tAB2] = AB2(fun, Jfun, x0, h, Tmax);
    errorAB2(it) = abs(xAB2(end)-xorig(1));
    [xBDF2,tBDF2] = BDF2(fun, Jfun, x0, h, Tmax);
    errorBDF2(it) = abs(xBDF2(end)-xorig(1));

end
figure
loglog(H,errorAB2,'r','LineWidth',2);
hold on
loglog(H,errorBDF2,'m','LineWidth',2);
grid on
legend('AB2','BDF2')
xlabel('Step␣Size(h)','Interpreter','latex');
ylabel('|error(1)|','Interpreter','latex');
title('Error(1)␣vs␣Step␣Size(h)','Interpreter','latex');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 26: AB2.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xout,tout] = AB2(fun, Jfun, x0, h, Tmax)

%% Settings
alpha = [1 -1 0];
beta  = [0 1.5 -0.5];

%% Solve ODE
[xout,tout] = solve_ode(alpha,beta,x0,Tmax,h,fun,Jfun);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 27: solve$_ode.m$

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x,t,counter,dx] = solve_ode(alpha, beta, x0, Tmax, h, fun, varargin)
%
if nargin > 6
    Jf = varargin{1};
end

%% Settings
eps = 1e-8;
max_iter = 1e1;

% Initializations
```

```
alpha = alpha./alpha(1); % normalize alpha
alpha = reshape(alpha,1,length(alpha));
beta  = reshape(beta,1,length(beta));
k     = length(alpha)-1;
t = 0:h:Tmax;

num_steps = length(t);
n = length(x0);
x = zeros(n,num_steps);
x(:,1)= x0;
% x(:,2)= cos(t(2));

%% Initialize with Backward Euler
if k > 1
    x(:,1:k) = backward_euler(fun,Jf,x0,h, h*(k-1));
end

if beta(1)==0 % If Explicit Method
    for i = k+1:num_steps
        u      = t(i);
        x(:,i) = -x(:,i-1:-1:i-k)*alpha(2:end)' + h * fun(x(:,i-1:-1:i-length(be
    end
else
    % Implicit Method
    % Construct Jacobian and RHS
    for i = k+1:num_steps
        u = t(i);
        b = x(:,i-1:-1:i-k)*alpha(2:end)' - h * fun(x(:,i-1:-1:i-length(beta)+1)
        Jacobianfun = @(x) eye(n) - h * beta(1) * Jf(x);
        RHSfun      = @(x) (x - h * beta(1) * fun(x,u) + b);
        [x(:,i),counter,dx] = solvenewton(RHSfun, Jacobianfun, x(:,i-1), max_ite
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

”forwardEuler.m”,”backwardEuler.m”,”BDF2.m” and ”solveode.m”, ”solvenewton.m” are common functions in multiple parts.