Luis Kligman Computer Assignment 6

Problem 1:

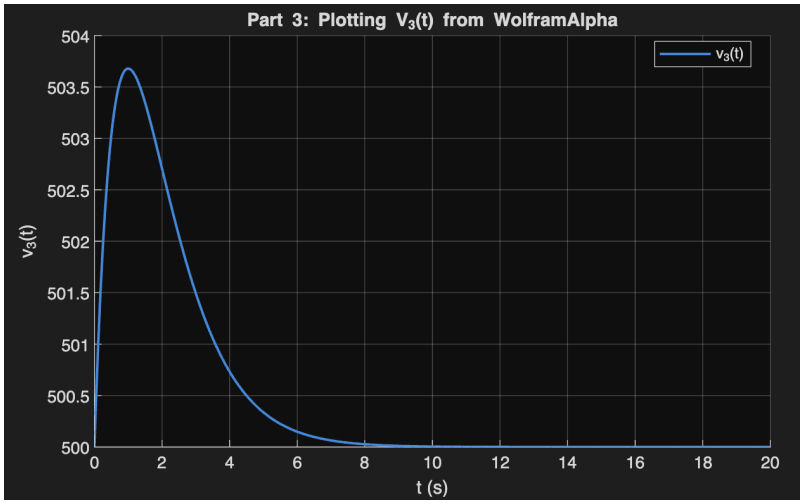
In question one, we are asked to determine $v_i(t)$ as t approaches infinity for all i (meaning all of our jets). Our code solves this by creating a column vector of one by five with all values being 500. With relative-only coupling, the average velocity is invariant as time goes to infinity. The damping term beta * L drives the velocity difference to zero. Therefore, as time goes to infinity, all velocities converge to the common value equal to the initial velocity, $v_i(0) = 500$.

Problem 2:

```
V1 =

(10*(100*s^6 + 602*s^5 + 1308*s^4 + 1416*s^3 + 862*s^2 + 303*s + 50))/(s*(s + 1)^2*(2*s^4 + 8*s^3 + 8*s^2 + 4*s + 1))

V2 =

(20*(50*s^4 + 200*s^3 + 202*s^2 + 101*s + 25))/(s*(2*s^4 + 8*s^3 + 8*s^2 + 4*s + 1))

V3 =

(500*s^2 + 1010*s + 500)/(s*(s + 1)^2)

V4 =

(20*(50*s^4 + 200*s^3 + 198*s^2 + 99*s + 25))/(s*(2*s^4 + 8*s^3 + 8*s^2 + 4*s + 1))

V5 =

(10*(100*s^6 + 594*s^5 + 1276*s^4 + 1368*s^3 + 830*s^2 + 295*s + 50))/(s*(s + 1)^2*(2*s^4 + 8*s^3 + 8*s^2 + 4*s + 1))
```

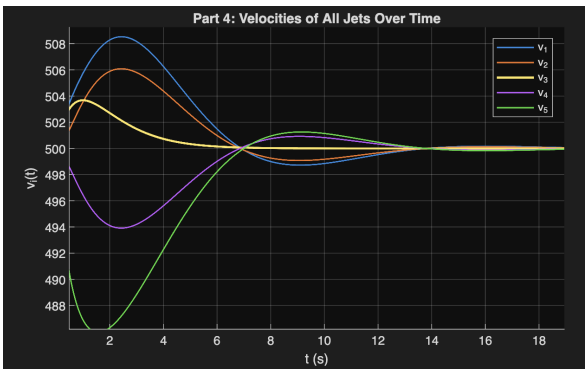
In question 2, we must determine the velocity of each jet in the s domain by taking the Laplace transform of the given differential equation in the assignment description. From the Laplace transform, we get, $sV(s) - v(0) = -D^{-1} [\alpha(LY(s) - b/s * \beta LV(s)]$. Where L is the degree matrix (number of neighbors the ith jet has) minus the adjacency matrix (which jets are the ith jet's neighbors) to get the Laplacian matrix L. In this equation, b is the collection of the delta terms, which comes out to be [-d; 0; 2d; 0; -d], alpha and beta are given by the problem's instructions being one and two, respectively, and d is 10, the desired spacing between neighboring jets. We also need to find $Y_i(s)$, which we can easily get from taking the Laplace transform of $y_i(t)$, which is given by the problem instructions. With this, we can now simplify our equation as all values have become known, leaving only V(s), which we can have MATLAB solve for. In my code, I have comments in the (“part 2”) section which demonstrate the processes of transforming the given functions into the forms we need and how to use them to solve for V(s); unfortunately, due to the limitations of Google Docs, I cannot explicitly copy some of the functions over. Although they are perfectly documented in my MATLAB code file. Since there are five jets in this case, I have already created matrices that correlate to our data. Using these matrices in the functions allows for better readability of the code I have produced, and also results in easier-to-work-with results. When we do simplify(M / RHS) in the MATLAB code, it assigns a one-by-five matrix to the variable V with all five jet velocities in it. Then I output each to the console, as shown in the image, which correlates to problem two.

Problem 3:



Problem three asks us to take the output function from problem two corresponding to $V_3(s)$ and use Wolfram Alpha to find the inverse Laplace transform. Resulting from this, I received the function $V_3(t) = 10te^{-t} + 500$. I further tested this using MATLAB by using ilaplace(V3, s, t) to check for any error between applications. Once I received my result, I used MATLAB's plotting functionality to make a plot containing the velocity value over time for jet 3.

Problem 4:



For problem four, we must plot $v_i(t)$ for all jets while firstly employing MATLAB to find $v_i(t)$ for all other jets over a time interval from 0 to 20, taking measurements every 0.001. Problem four also wants us to plot the position difference at every time step. MATLAB is able to solve this by creating arrays that we can use to store all of the velocities and positions of all jets for each time step. Each array will be used as our data for the plots. V and Y, which are the velocity and position matrices, are of length 5 by Nsteps+1, where Nsteps is the number of steps we need to take based on the given range of time and the step length (0.001); In this case, Nsteps equals 20,000. We initialize our first column vectors of each matrix (position and velocity) to be equal to the initial conditions given by the problem. We then use a for loop from 1:Nsteps, starting at 1 because these are our initial conditions, and our next jet movements must be based on current positions and velocities. In the code, v_k is the vector of all five jets' velocities at the current time step, and y_k is the vector of all five jets' positions at the current time step. We use these to

compute the derivative using the given ODE, $dv = -D^{-1} * (\alpha(Ly_k - b) + \beta Lv_k)$. From this, we add the value dv*dt to our current v_k value to increase or decrease our velocity based on our neighbors' positions. We update our velocities first because our new position is dependent on the current velocities. Once the new velocities are stored, we then integrate position, including our new velocity (v) value, using the forward Euler technique. Once this is done, we can now plot each column in our v matrix, as each column refers to a given time step stored monotonically. This is plotted in Figure two, and in the code I gave $v_3(t)$ a bit heavier linewidth so we can see how it looks compared to Figure one from Problem two, where it was the only one graphed, compared to how it looks when all five jets are graphed. In Figure three, we plot the spacing of each jet over the time interval by subtracting the matrices from each other to come up with the differences (example: y21 = y(2,:) - y(1,:)), which gives the spacing between jet 2 and jet 1. Then we can simply plot each matrix for each time step, resulting in a graph of the jets' positions over time.

Appendix:

```
%% --- PROGRAMMING ASSIGNMENT 6  LUIS KLIGMAN ---
% Ni : set if neighbors of jet i (adjacent jets)
% |Ni| : Number of those neighbors; cardinality of the set Ni
% alpha : "stiffness" (penalize position error to desired spacing)
% beta : "damping" (penalizes velocity disagreement with neighbors)
% deltaij : desired long-term spacing yi - yj (signs are important)
clear all;
close all;
clc;
syms s t
%% --- Problem Data (given in assignment) ---
% Initial positions y(0) and velocities v(0):
y0 = [0; 20; 40; 60; 80];  % Column Vector
v0 = [500; 500; 500; 500; 500];  % Column Vector; all 500 initially
% Control gains and desired spacing:
alpha = 1;  % Stiffness coefficient
beta = 2;  % Damping coefficient
d = 10;  % Desired neighbor spacing
% Adjacency Matrix (A) -- "Who talks to whom"
% Jet 1 --> connected to Jet 2
% Jet 2 --> connected to Jets 1 and 3
% Jet 3 --> connected to Jets 2 and 4
% Jet 4 --> connected to jets 3 and 5
% Jet 5 --> connected to Jet 4
A = [ 0, 1, 0, 0, 0;
     1, 0, 1, 0, 0;
     0, 1, 0, 1, 0;
     0, 0, 1, 0, 1;
     0, 0, 0, 1, 0 ];
% Each row represents one jet
% Each column shows which other jets it can sense/communicate with
```

```
% A '1' in row i, column j means that jet i and jet j can communicate
% A '0' means no direct communication
% The matrix is symmetric since communication is bidirectional
% y2 - y1 --> d
% y3 - y2 --> d
% y3 - y4 --> d
% y4 - y5 --> d
% That implies (by definition Deltaij = lim t->inf (yi - yj)
% Delta12 means the position of jet 1 minus the position of jet 2
% Delta21 means the position of jet 2 minus the position of jet 1
% Edge (1,2): delta21 = d :: delta12 = -d
% Edge (2,3): delta32 = d :: delta23 = -d
% Edge (3,4): delta34 = d :: delta43 = -d
% Edge (4,5): delta45 = d :: delta54 = -d
% Degree Matrix (D) -- "How many neighbors each jet has"
% Jet 1 --> 1 neighbor
% Jet 2 --> 2 neighbors
% Jet 3 --> 2 neighbors
% Jet 4 --> 2 neighbors
% Jet 5 --> 1 neighbor
% Degree matrix D (|Ni| on the diagonal): ends have 1 neighbor, middle
% have 2
D = [ 1, 0, 0, 0, 0;
     0, 2, 0, 0, 0;
     0, 0, 2, 0, 0;
     0, 0, 0, 2, 0;
     0, 0, 0, 0, 1 ];
% Dii = number of neighbors (the 'degree' of node i)
% Off-diagonal entries are zero since no jet's degree depends on others
% Multiplying by this matrix makes it so jets that have two neighbors (like
% 2, 3, 4) do not end up reacting with twice the corrective force of jets
% that have only one neighbor (like 1 and 5)
% Laplacian Matrix (L = D - A) -- "Relative difference operator"
L = D - A;
% Computing this gives:
% L =
%      1    -1     0     0     0
%     -1     2    -1     0     0
%      0    -1     2    -1     0
%      0     0    -1     2    -1
%      0     0     0    -1     1
% The diagonal element Lii = number of neighbors (from D)
% The off-diagonal element Lij = - 1 if jet i and jet j are connected
% Why this matters: When you multiply L by a position vector y, this
% produces the sum of all relative position errors for each jet with its
% neighbors
% It measures how far each jet is from being aligned with its neighbors
% Inside the differential equation for a single jet i, it tells us:
% Each jet looks at its position and velocity relative to its neighbors
```

```
% Deltaij is the desired spacing between jets i and j
% Alpha and Beta control how strongly the jets react to position or
% velocity errors
% Inside the sum, we have (yi - yj - deltaij)
% That term measures how far jet i is from where it should be relative to
% jet j
% If yi - yj > deltaij : jet i is too far -> it slows down
% If yi - yj < deltaij : j i is too close -> it speeds up
% We can use this to simplify the given differential equation
% Create column vectors y(t) and v(t) to collect all positions and
% velocities
% Using the Laplacian L = D - A matrix, it encodes all relative
% differences
% Then the sum of differences over all neighbors can be compactly written
% as Ly and Lv
% This condenses the differential equation into
%  dvi(t)     -1
%  -----  = -D  [ alpha * (Ly(t) - b) + beta*Lv(t) ]
%    dtpwd
% Where b is the collection of the delta terms
% Compute bi = Sigma   Deltaij
%               j element of Ni
%    Jet    Neighbors    bi                                result
%     1        {2}       delta12 = -d                        -d
%     2        {1,3}     delta21 + delta23 = d - d           0
%     3        {2,4}     delta32 + delta34 = d + d           2d
%     4        {3,5}     delta43 + delta45 = -d + d          0
%     5        {4}       delta 54 = -d                       -d
b = [-d; 0; 2*d; 0; -d];
% Therefore, b acts like a bias correction vector that tells the controller
% what steady-state geometry formation shape we want
%% --- Part 1:    lim   v_i(t)    ---
%               t -> inf
% With relative-only coupling, the average velocity is invariant, and the
% damping term beta*L drives velocity difference to zero. Therefore, all
% velocities converge to the common value equal to the initial average.
v_0 = 500;              % Initial average velocity
v_inf = v_0*ones(5,1);
disp('Part 1: Steady-state velocities:');
disp('lim_{t->inf} v_i(t) = ');
disp(v_inf);  % Expect all 500
% The formation reaches the right shape (relative distance) and then moves
% together with a common constant velocity c. In this case, c = 500, but c
% is not arbitrary; instead, it is based on the initial given velocities
%% --- Part 2: Determine V_i(s) with MATLAB for all i ---
%  dv_i(t)       1
%  ------   = - -----  SIGMA  α(y_i(t) - y_j(t) - Δij) + β(v_i(t) - v_j(t))
%   dt          |N_i| j ∈ N_i
%% FROM LAPLACE
```

```
%
% sV(s) - v(0) = -D^-1 [α(LY(S) - b/s) * βLV(s)]
%
%% NOW FIND LY(s) FROM y_i(t)
%           y(0)     V(s)
% Y(s) =   ------ + ------
%            s         s
% Build the left-hand operator M(s) and right-hand RHS(s):
D_inv = inv(D);  % D^{-1}
M    = s*eye(5) + D_inv*( beta*L + (alpha/s)*L );      % sI + D^{-1}(βL + α/s *
L)
RHS  = v0 - (alpha/s)*D_inv*( L*y0 - b );              % v0 - (α/s)D^{-1}(Ly0 -
b)
% Solve symbolically for all V_i(s)
V = simplify( M \ RHS ); % 5x1 symbolic vector: [V1(s); ...; V5(s)]
disp('Part 2: V_i(s) for all i using MATLAB')
V1 = simplify(V(1))
V2 = simplify(V(2))
V3 = simplify(V(3))
V4 = simplify(V(4))
V5 = simplify(V(5))
%% --- PART 3: ---
%% WITH WOLFRAMALPHA, CALCULATE THE INVERSE LAPLACE TRANSFORM OF V_3(s) AND
PLOT V_3(t) IN MATLAB
% Taking the output function of V_3(s) from part 2: (500*s^2 + 1010*s +
500)/(s*(s + 1)^2)
% Giving that to WolframAlpha results in: 10te^-t + 500
V3_t = ilaplace(V3, s, t);  % V3_t is equal to the result from WolframAlpha
% Plot V3_ as a function of time
V3_ = 10*t*exp(-t) + 500;
% Change V3_ which has syms t, so that t becomes a time variable
V3_func = matlabFunction(V3_, 'Vars', t);
t_vals = linspace(0, 20, 1000);
figure; hold on; grid on;
plot(t_vals, V3_func(t_vals), 'LineWidth', 1.5);
xlabel('t (s)'); ylabel('v_3(t)')
title('Part 3: Plotting V_3(t) from WolframAlpha');
legend({'v_3(t)'}, 'Location','best');
%% --- PART 4: NUMERICAL SIMULATION OF VELOCITIES AND POSITIONS ---
dt = 1e-3;  % Time step
Sec = 20;  % End time (seconds)
Nsteps = round(Sec/dt);  % Number of steps
% Pre-allocate arrays for speed efficiency
t_vals = (0:Nsteps)*dt;  % Time vector (length Nsteps+1)
v = zeros(5, Nsteps+1);  % Initializes a 5×(Nsteps+1) matrix of zeros to store
velocity values.
y = zeros(5, Nsteps+1);  % Initializes a 5×(Nsteps+1) matrix of zeros to store
position values.
% Set initial conditions at index 1 (time t=0)
```

```
v(:,1) = v0;
y(:,1) = y0;
% Time stepping:
for k = 1:Nsteps
   % Current values
   vk = v(:,k);  % vk is the vector of all five jets' velocities at the current
time step t_k
   yk = y(:,k);  % yk is the vector of all five jets' positions at the current
time step t_k
   % Compute derivatives using the ODE
   % dv/dt = -D^-1 * [ alpha*(L*yk - b) + beta*(L*vk) ]
   % dv is the vector of computed derivatives from the equation
   dv = -D_inv * ( alpha*(L*yk - b) + beta*(L*vk) );
   % Forward Euler
   v(:,k+1) = vk + dt*dv;  % update velocity first
   y(:,k+1) = yk + dt*v(:,k+1);  % then integrate position with new v
end
% Plots for velocities
% Each row of v(i,:) stores the velocity history of jet i across all time
% steps.
figure; hold on;
plot(t_vals, v(1,:), 'LineWidth', 1.2);  % Jet 1
plot(t_vals, v(2,:), 'LineWidth', 1.2);  % Jet 2
plot(t_vals, v(3,:), 'LineWidth', 1.8);  % Jet 3   highlight v3
plot(t_vals, v(4,:), 'LineWidth', 1.2);  % Jet 4
plot(t_vals, v(5,:), 'LineWidth', 1.2);  % Jet 5
grid on;
xlabel('t (s)');
ylabel('v_i(t)');
title('Part 4: Velocities of All Jets Over Time');
legend({'v_1', 'v_2', 'v_3', 'v_4', 'v_5'}, 'Location', 'best');
% Plot the four required position differences (neighbor spacings)
% These represent the distances between adjacent jets over time
% Each curve shows how fast the jets reach the desired steady-state spacing
% d = 10
% Compute position differences at every time step
y21 = y(2,:) - y(1,:);  % Spacing between Jet 2 and Jet 1
y32 = y(3,:) - y(2,:);  % Spacing between Jet 3 and Jet 2
y34 = y(3,:) - y(4,:);  % Spacing between Jet 3 and Jet 4
y45 = y(4,:) - y(5,:);  % Spacing between Jet 4 and Jet 5
figure;
hold on;
% Plot each spacing vs time
plot(t_vals, y21, 'LineWidth', 1.6);
plot(t_vals, y32, 'LineWidth', 1.6);
plot(t_vals, y34, 'LineWidth', 1.6);
plot(t_vals, y45, 'LineWidth', 1.6);
yline(d, 'k--', 'LineWidth', 1.2);  % Reference Line for the desired spacing to
converge to
```

```
grid on;
xlabel('t (s)');
ylabel('y-differences (m)');
title('Part 4: Position Differences of Jets Over Time');
legend({'y_2 - y_1','y_3 - y_2','y_3 - y_4','y_4 - y_5'}, 'Location', 'best');
```