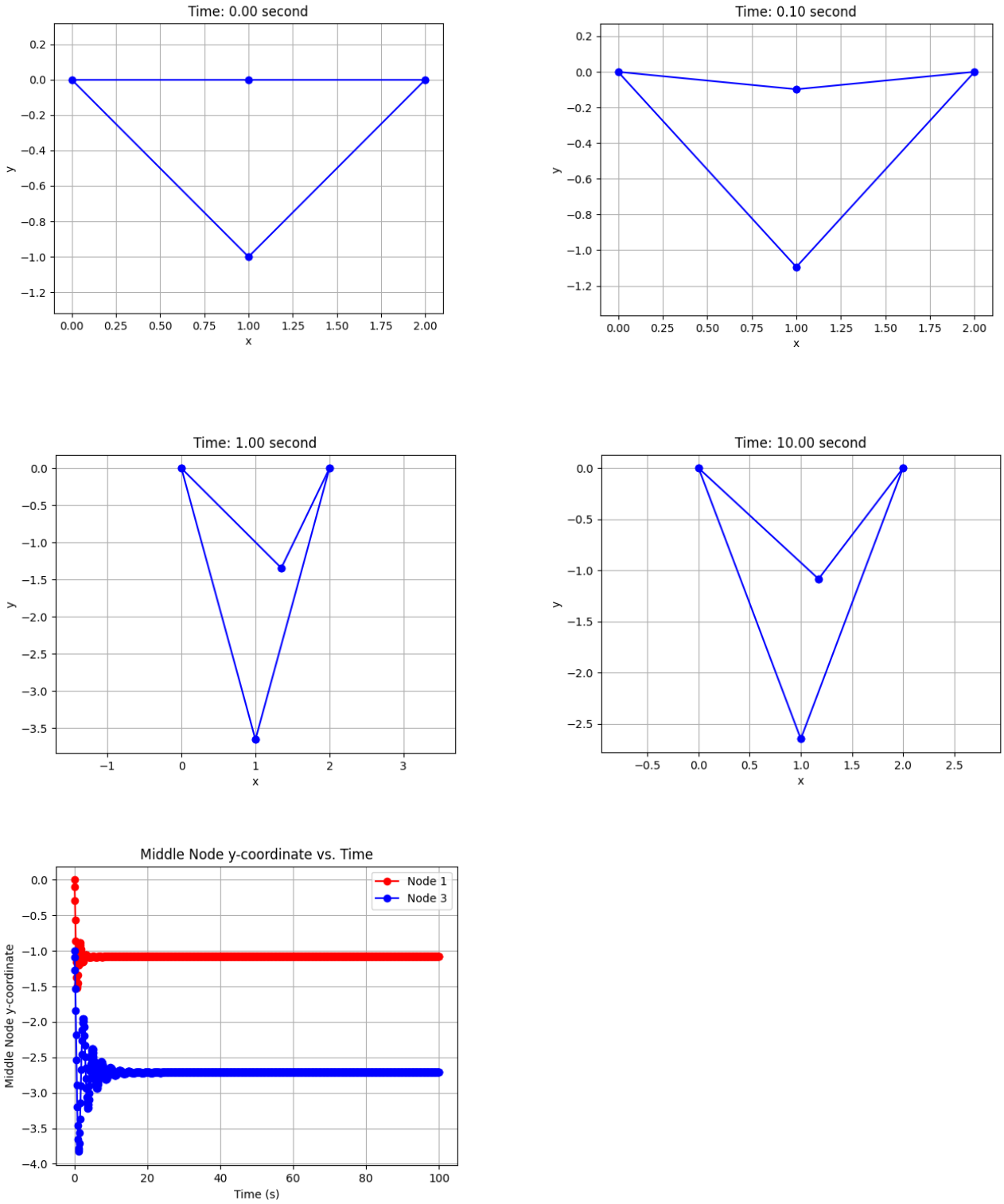
HW1 MAE 263F

Harrison Sandstrom

I. INTRODUCTION

Figure 1: Spring system at 0, 0.1, 1, and 10s with y position of free nodes over time



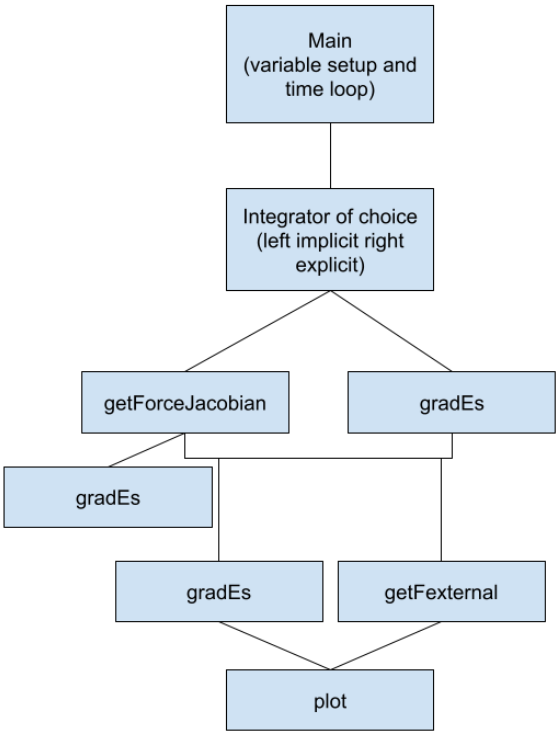
II. REPORT DELIVERABLES

1. Pseudocode and Program Structure

The following pseudocode describes the programs structure with a diagram showing the programmatic flow from calling the function to output.

```
main()
Load the nodes and springs from nodes.txt and springs.txt
Calculate # of nodes
Calculate # DoF (2*# of nodes)
Initialize x, u and v (pos vel and acc)
Initialize the rest length of the springs
Initialize node masses (1kg)
Fill x_old from node vectors in nodes.txt
Set the mass vector w

For every timestep:
 X_new, u_new = myInt(t_new, x_old, …)
   Plot desired timesteps t=0, 0.1, 1, 10, 100
 Plot middle vs time

def gradEs(xk,yk,xp1, ykp1, l_k, k):
Computes the internal force contributed by a spring connecting two nodes also
known as the gradient of the stretching energy. Returns a vector containing
the Force on each node of its inputs (node k and node k+1 in the two
directions x and y).

def hessEs(xk,yk,xp1, ykp1, l_k, k):
 Gives a 4x4 symmetric matrix which is the hessian of the stretching energy
E_k^s. This is effectively the jacobian of the springs between two nodes.

def getFexternal(m):
 Takes in the input nodes and calculates based on the masses the applied
gravitational force on them.

def getForceJacobian(x_new, x_old, u_old, stiffness_matrix, index_matrix, m,
dt, l_k):
 Calculates the jacobian and force so that the residuals can be minimized in
the newton-raphson method. It loops through the various indices and
calculates the stiffness for each node as well as the f_spring and J_spring
to get the net force and net Jacobian.

# Integrator
def myInt(t_new, x_old, u_old, free_DOF, stiffness_matrix, index_matrix, m,
dt):
 Solves for the next x_new by using the Newton-raphson method to reduce error
by constraint of setting f=0, using getForceJacobian. It only solves for the
free_DOF minimizing compute and removing the possibility of error being added
through the boundaries, updating x_new until the residual is sufficiently
small. Finally returns x_new and u_new.

# Explicit Euler
def myIntExplicit(x_old, u_old, free_DOF, m, dt, index_matrix,
stiffness_matrix, l_k):
 Calculate the future position based on the current state values. Doesn't
require any error minimization or anything to run. Base equations are as
follows:
 a_old = f/m
 x_new = x_old + dt*u_old
 u_new = u_old + dt*a_old
# plotting helper
def plot(x, index_matrix, t):
 Plots the nodes at a given time t based on the current state
```

Figure 2: Control Flow Diagram



2. Choosing an appropriate time step Δt?

   When choosing an appropriate time step Δt we want to ensure that there is no unexpected behavior while running the code. In the case of our implicit euler, a Δt that is too large will lead to the amount of iterations that each step of the implicit euler requires is higher, so choosing a step that keeps it from stalling is important so that the implicit euler step can progress. Additionally the larger the time step the more "lost energy" there is in the system, so a smaller time step can lead to the system behaving more realistically over the course of its run. However if we make the time steps too small the simulation can be inefficient and take too long to run which is similarly bad as we cannot get our results in a timely manner while wasting resources or even just running out of compute before reaching our solution. This is a balance where we have to choose a dt that neither causes too much error minimization in the newton-ralphson step nor wastes time calculating too many time steps.

3. Explicit Euler

   When simulating using Explicit Euler a dt of 0.1 didn't cause there to be any crashing of the system

Figure 3: dt = 0.1 Implict Euler 1s

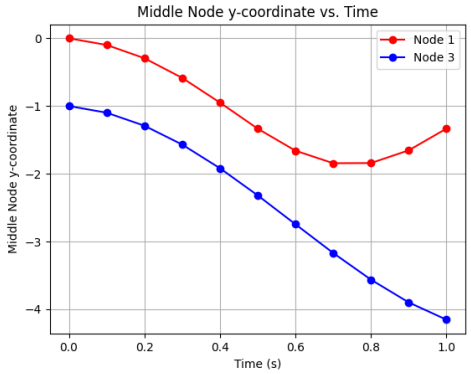
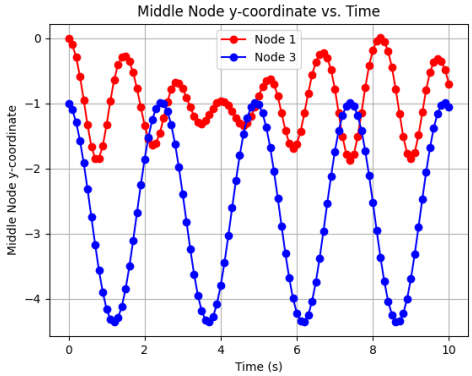


Figure 4: dt = 0.1 Implict Euler 1s



In fact, when we run this system using implicit euler we see that the y positions using a dt =0.1s are relatively stable, however we see that node 1 varies and increases in amplitude at times while decreasing at others. I cannot say for sure where this extra energy enters the system and leaves to give this increasing amplitude.

4. Implicit euler reaches a static state before the end of the term because of what is called neumerical damping. This is because the method is only what is called “first order accurate” which then causes unconditional stability and eventual damping. This happens specifically through damping of energy from the system. Effictively this comes from the fact that only the first order derivatives are directly present in the system, so acceleration is seen as a stair step between each step and the system therefore has to lose energy between these steps to mitigate the error in the systems residuals. What it comes down to is that the additional step causes the system to be able to conserve energy through the use of the acceleration term.

5. Could not get the newmark beta working in the meantime

Citations:

1. Jawed, K., Flexible Robotics, MAE 263F, Fall 2025, UCLA

2. "Newmark-β method – Modelling and Simulation in Structural Mechanics," BayernCollab, 28 October 2020, by Sebastian Resch-Schopper. URL: https://collab.dvb.bayern/spaces/TUMmodsim/pages/71122788/Newmark-%CE%B2+method