Homework 3

Harrison Sandstrom

I. Deliverable 1

A. *force/energy evaluation*

Pseudocode located in Appendix 1:

B. *time stepping*

For the time steps I used a time step of 0.2 going from $t \in [0, 1000]s$. In each time step the error was minimized through the objfun function before moving on to the next time step where the control input was predefined. This time step was chosen as it balanced the amount of computation required due to reducing the movement per step while also keeping the general amount of steps small as we simulated over individual time steps.

C. *enforcement of Dirichlet constraints*

Enforcement of the Dirichlet constaints nodes N-2 and N-1 (assuming you start counting at 0) were removed from the free nodes in the calculations and then I directly set the X, Y, and theta values of the nodes based on the desired location and angle. This was done directly through the set of 4 equations given and the chosen values will be further explained in the path planning section.

D. *path planning*

Because the central node had to track a path that mapped a semicircle around the origin at $\frac{L}{2}$ from the origin, there were not very many ways that made sense to track this path as it lay on a point where the simplest path relied on the bar being straight. As such if the desired path of the center node was based around the radius $\frac{L}{2}$ the control end of the bar was swung around the origin at radius L with θ_c being equivalent to $\arctan 2(x_c, y_c)$ about the origin. This kept the bar straight at all angles and led to only a very small error between the rod and the desired location as the gravitational deflection of the rod was very small.

The equations that modeled this were as follows:

$x_c = L\cos(\frac{\pi}{2}\frac{t}{1000})$

$y_c = -L\cos(\frac{\pi}{2}\frac{t}{1000})$

$\theta_c = \arctan 2(x_c, y_c)$

II. Deliverable 2

In this section we can see the map of the control inputs over time as x_c, y_c and θ_c evolve over time. As can be seen in figures 1 and 2 the progression of the control points is quite smooth as there is no error tracking and it is just following a fixed path through the course of its trip. We can see the visualization of this trip

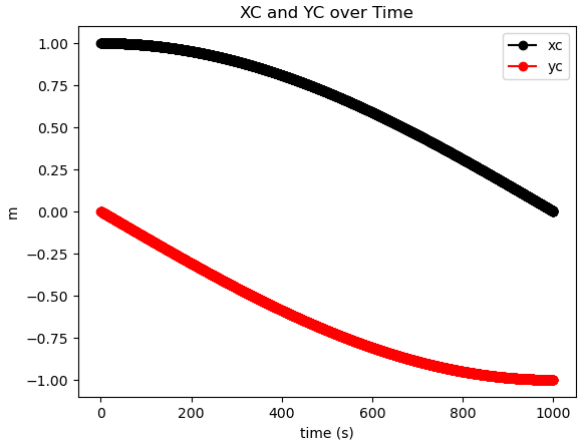

Fig. 1. Evolution of x_c and y_c over time..

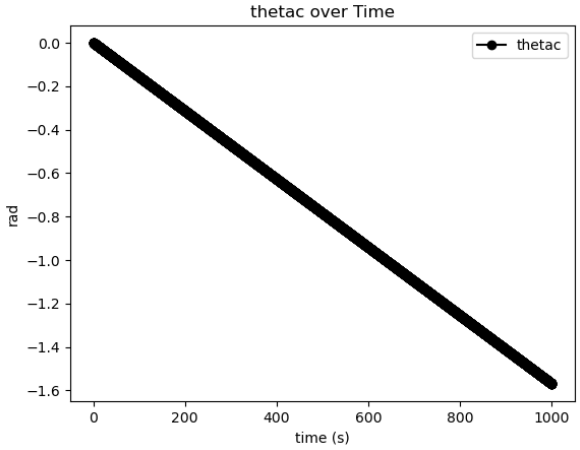

Fig. 2. Evolution of θ_c over time.

III. Deliverable 3

In this section we will see the evoluation of the nodes over the course of the trajectory. It will be seen that the rod remains mostly straight over the course of its trajectory with very minor positional error as well over the trip.

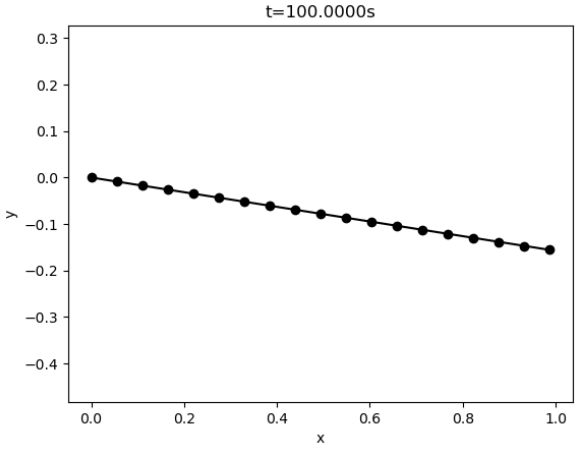

Fig. 3. Beam nodes at $t = 100s$.

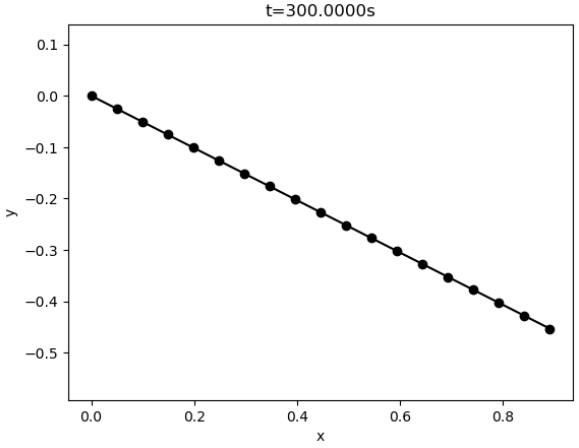

Fig. 4. Beam nodes at $t = 300s$.

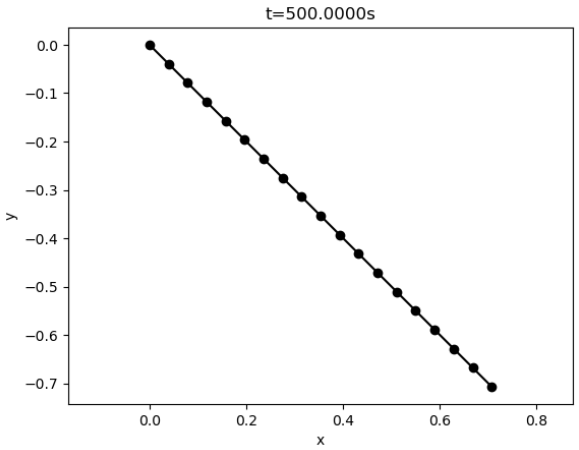

Fig. 5. Beam nodes at $t = 500s$.

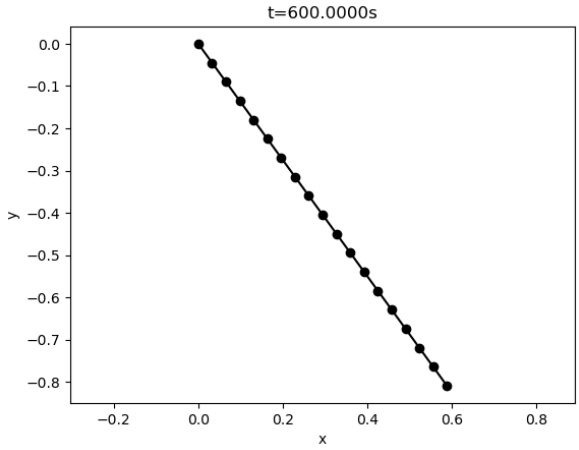

Fig. 6. Beam nodes at $t = 600s$.

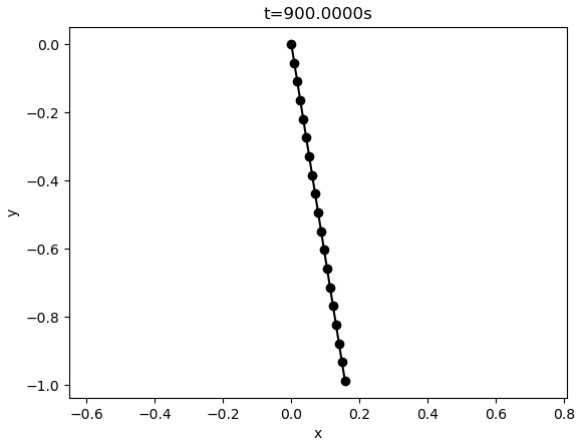

Fig. 7. Beam nodes at $t = 900s$.

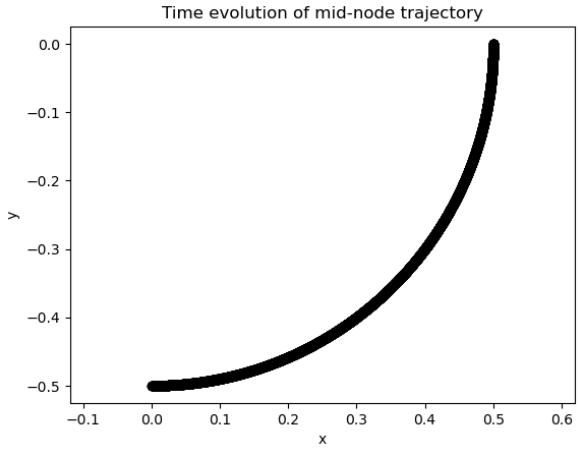

Fig. 8. Trajectory of the mid node over the course of 1000s.

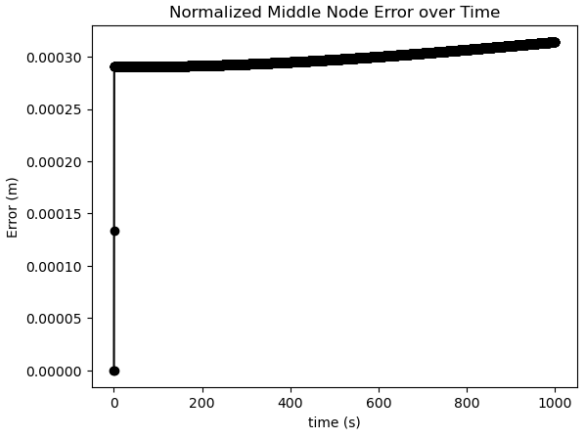

Fig. 9. Error of the midnode over time.

IV. Deliverable 4

The limitation of this method is that the maximum reach from the starting position that the robot arm has to move is $\sqrt{2}$m away which may be further than intended. However with the chosen node path it is pretty hard to avoid this path without requiring something like a kink as the rod pretty much has to be straight for the mid-node to reach its desired path as it is on the edge of the mid nodes reachable workspace without requiring a large amount of stretching of the nodes which is unrealistic in this model and would require a significant force input by the robot arm. Therefore it seems much more reasonable to expect the robot to have a reaching radius of $\sqrt{2}$ m than to reach further. There isn't really an issue with infeasible commands in this implementation as there is no feedback control that would cause a saturation of inputs, only direct movement of the arm to the desired location.

APPENDIX

```
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
```

REFERENCES

[1] M. K. Jawed, Lecture_06_Main_Falling_Beam_N_nodes_Fixed_DOFs.ipynb, UCLA, 2025