Homework 3 Report

John Meshinsky

***Abstract*— This homework covers the robotic control of an elastic beam. The robot controls the end of the rod with the other end fixed to the origin. The purpose is to control the end point to have the middle of the beam follow a given trajectory under the effect of gravity.**

I. INTRODUCTION

This homework covers the robot control of an elastic beam to follow a given trajectory. The beam has the given properties listed in Table 1.

The values of beam are as followed:

Measurement	Value
Beam Length [L]	1 (m)
Outer Radius [R]	0.013 (m)
Inner Radius [r]	0.011 (m)
Young's Modulus [E]	70 (GPa)
Density [ρ]	2700 ($\frac{kg}{m^3}$)

Table 1: Given Values of the Beam

The beam also has the following boundary conditions for the node at the origin

$$x_1(t_{k+1}) = 0 \quad [1]$$

$$y_1(t_{k+1}) = 0 \quad [2]$$

And Dirichlet constraints for the last node,

$$x_N(t_{k+1}) = x_c(t_{k+1}) \quad [3]$$

$$y_N(t_{k+1}) = y_c(t_{k+1}) \quad [4]$$

and the second to last node.

$$x_{N-1}(t_{k+1}) = x_c(t_{k+1}) - \Delta L cos(\theta_c(t_{k+1})) \quad [5]$$

$$y_{N-1}(t_{k+1}) = y_c(t_{k+1}) - \Delta L cos(\theta_c(t_{k+1})) \quad [6]$$

The beam must be directed by the end node pivoting from the node at origin to direct the movement of the middle of the beam to the given trajectory in 1000 seconds under gravity. The trajectory given is:

$$x^*(t) = \frac{L}{2} cos(\frac{\pi}{2}\frac{t}{1000}) \quad [7]$$

$$y^*(t) = -\frac{L}{2} sin(\frac{\pi}{2}\frac{t}{1000}) \quad [8]$$

This trajectory is a clockwise circle with a radius half the length of the beam around the origin. The path it would travel would be a quarter of the circle in 1000 seconds.

$$x_c(t) = Lcos(\frac{\pi}{2}\frac{t}{1000}) \quad [9]$$

$$y_c(t) = -Lsin(\frac{\pi}{2}\frac{t}{1000}) \quad [10]$$

$$\theta_c(t) = -\frac{\pi}{2}\frac{t}{1000} \quad [11]$$

Considering this, I figured that it would be best for the end point of the beam to travel a similar clockwise circle with a radius of the full length of the beam to cause the middle to follow a similar trajectory. This plan alone, however, does not consider the deviation the middle of the beam might experience under the forces of bending, stretching, and gravity.

To correct the path for the deviation, I would want to set up a PID control feedback loop to correct the robot path to minimize the deviation. The loop would break within a threshold of error (it was 1 micrometer for this code) or after a maximum amount of iterations. This would return a corrected position for the end point for the desired.

II. MATH

The following equations calculate the values needed for the mass at each node as well as the moment of Inertia for the cross-section of the beam:

$$m = \frac{\pi(R2 - r2)l\rho}{(N-1)} \quad [12]$$

$$I = \frac{4}{\pi}(R^4 - r^4) \quad [13]$$

The homework will use the implicit methods from the lecture as well as force balancing to account for the load:

$$F_{net} = F_{stretching} + F_{bending} + F_{Gravity} \quad [14]$$

$$J = J_{stretching} + J_{bending} \quad [15]$$

The Euclidean distance formula to measure the error between actual and desired middle node position.

$$d_{error}^2 = (x_{mid} - x^*)^2 + (y_{mid} - y^*)^2 \quad [16]$$

The equations for PID control:

$$P = Kp * d_{error}(t) \quad [17]$$

$$I = Ki \int d_{error}(t)\, d(t) \quad [18]$$

$$D = Kd \frac{d_{error}(t)}{dt} \quad [19]$$

$$x_c/y_c/\theta_c \mathrel{+}= P + I + D \quad [20]$$

III. PESUDOCODE

The code uses the modified implicit functions that we have used in the previous lectures, focusing on the iterations of the one for the falling beam simulations from Lecture 6. The modified use of the implicit function uses load force and the position the load is applied to simulate the force applied to the beam. The implicit function then returns the new positions of the arrays for calculation.

The main function will set up the values and measurements for the simulations. Functions will set up arrays for the desired middle node positions using equations [7] and [8] and the starting robot control positions code using equations [9], [10], and [11]. The first loop takes a time array from 0 seconds to 1000 seconds separated by steps of 0.1 seconds. The loop will measure the positions of the nodes at the timestep with the given forces and robot controls.

The code will then go into a nested while loop measuring error the simulated and desired middle node positions, using the Euclidean distance equation [16], and using the error to PID equations [17], [18], [19] , and [20] to correct the robot controls. This will go on until the error is less than 1 micrometer and go through 50 iterations. The node positions and new robot control at the step are then saved. It will then go on to plot the values.

The code uses functions crossMat(), gradEb(), hessEb(), gradEbs(), hessEs(), getFs() and getFb() from the lectures.

compute_control_inputs():

This function makes an array of the initial robot controls for the system.

compute_target_position():

This code makes an array of the desired middle node trajectory for the system.

objfun_with_control():

This function applies the Dirichlet constraints of the system. It also applies the energy and forces to compute the position of the nodes at the given time step.

Main:

The main function sets up the parameters of the system and measures the trajectory. It goes into a for loop for each time step to compute the positions of the nodes, then it goes into a nested time loop to correct the robot control positions to minimize the error of the middle node.

The measurements and snapshots are saved. Then the function plots the data.

IV. PLOTS

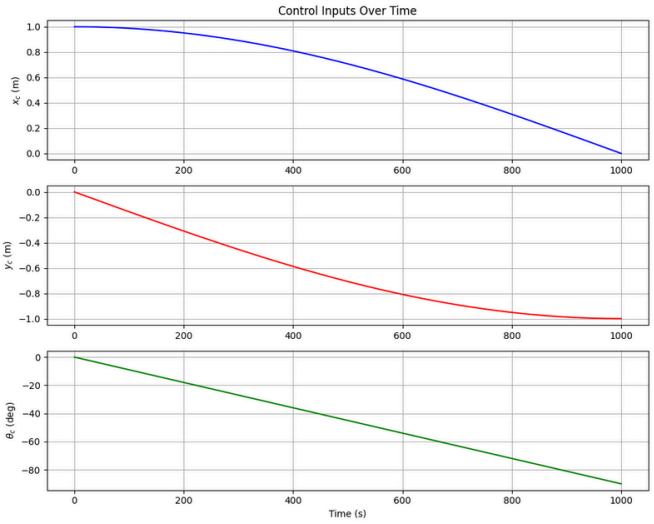
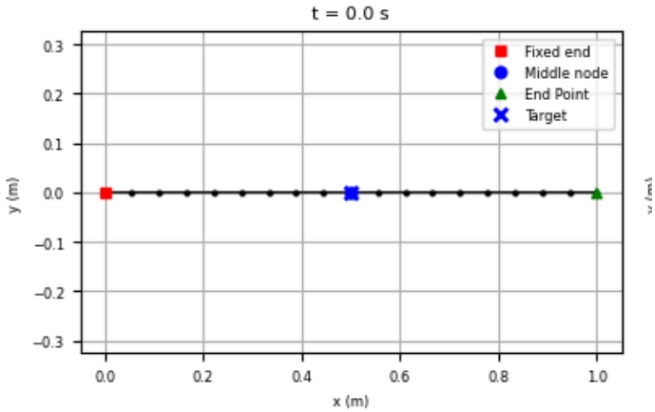
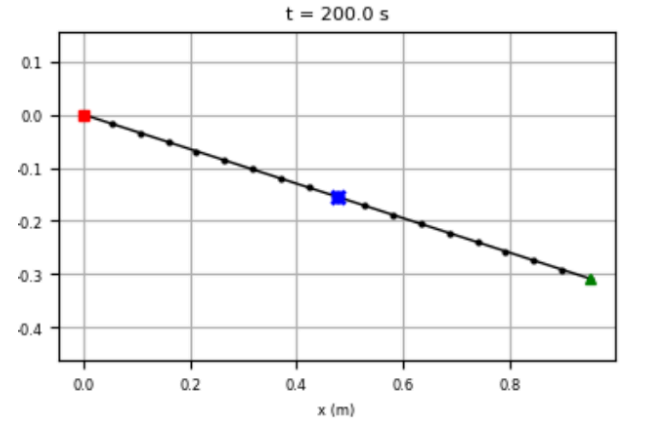
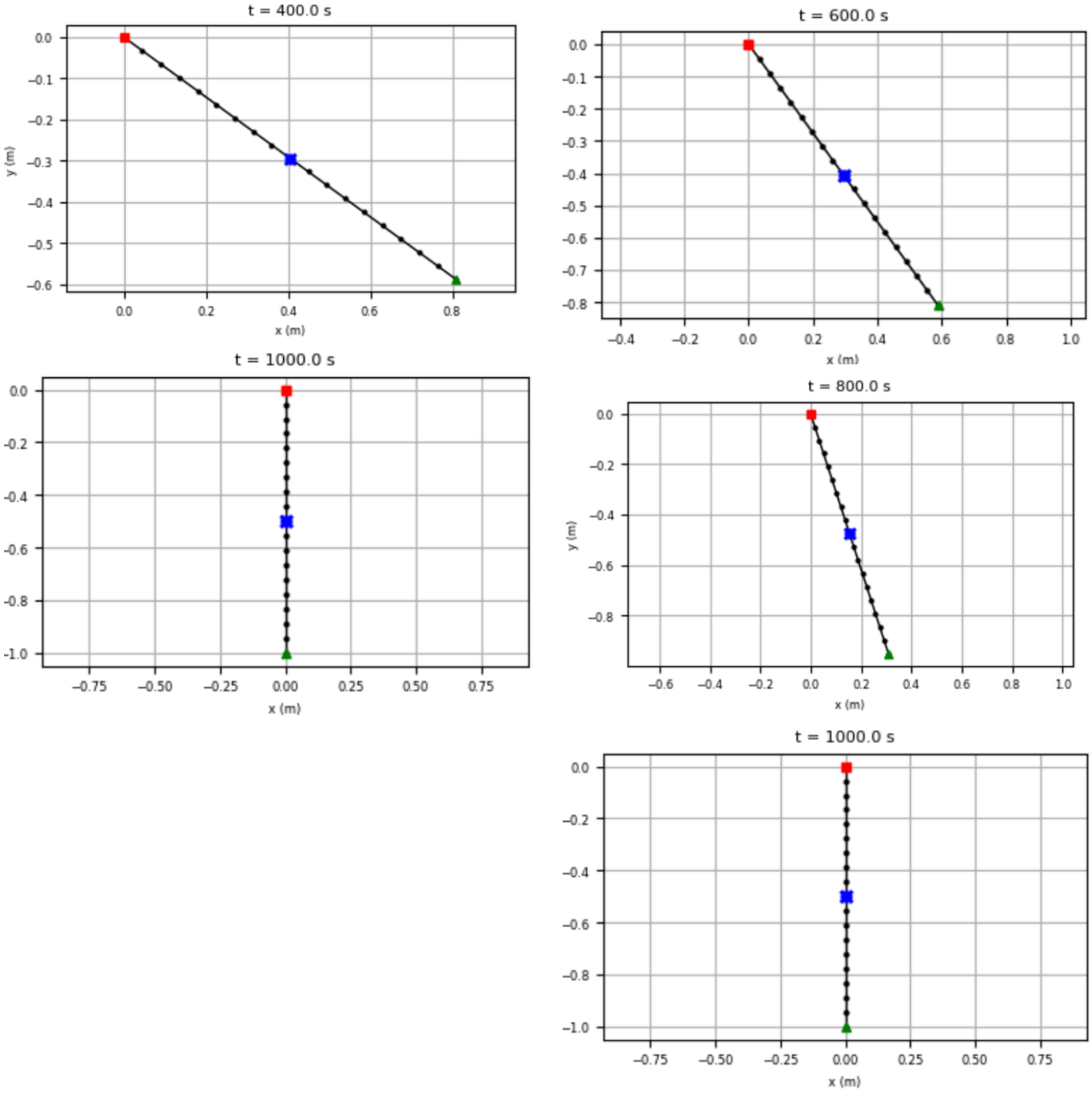


Figure 1-3: Plots of the Corrected Robot Controls over Time







Figures 4-9: Snapshots of the system over time

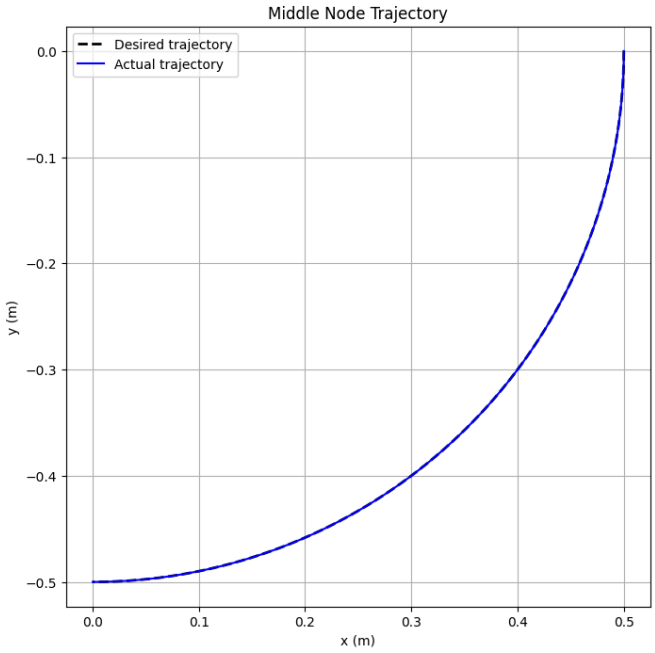


Figure 10: Trajectory of the middle node

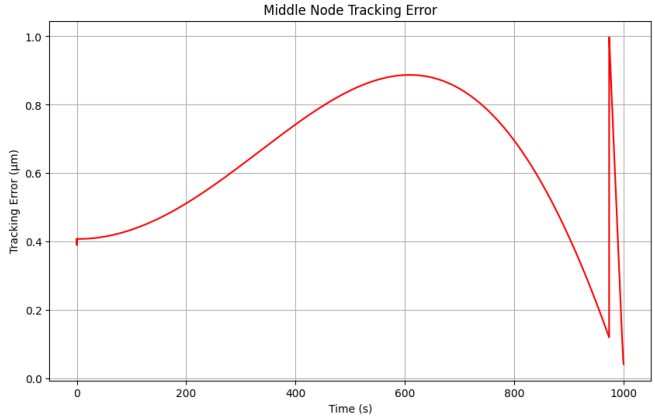


Figure 11: Distance of Error middle node over time

V. Discussion

The robot's workspace would limit the space which the robot could move and angle the end point. The Joints would also limit how the robot can move within the workspace. Depending on its positions and workspace, some things cannot handle specific commands and direction due to these limits. It is good to implement different methods to handle limitations.

Setting up saturation limits is a good way to handle commands it cannot handle. It sets up a contradiction which the system checks if each joint handles the suggested movement and prevents it from moving in those ways. This method can usually guarantee not pushing the robot past its limits. However, this alone is not usually enough as it often does not plan in advance for avoiding these situations and would need to pair methods to alternative ways to handle the commands.

There is feedback control where the system can sense it is approaching a limit of the system and adjust the controls to correct the commands to make it feasible. These methods often take up computational power spending on how it is implemented.

Then there is trajectory re-timing which slows down the processes to maintain the path given but adjusts the velocity and acceleration of the controls to maintain smooth motion. This often still achieves the desired motions, but done at a slower time frame.