Assignment-2

Report

Vedant Pandya-260907163

Part – A)

A. In creating the PID controller, I started by identifying the error, derivative, and integral terms, which would then be adjusted by their respective gains. The desired angle for the pendulum is 0, meaning the discrepancy is the actual angle. I saw that theta_dot was already available in the provided state, so I used it as the derivative. I also added a self.prev_integral attribute to the controller to compute the sum of past errors at each interval.

Part-B)

For the tuning process, I started with the derivative and integral gains set to zero (Kd and Ki) to determine an effective Kp value. A suitable Kp value was found to be around 2.0, which provided a rapid response without excessive oscillation. To manage the overshoot and reduce the oscillations, I used a Kd value of 0.41. I introduced the integral gain to address steady-state errors, settling on a Ki value of 0.005 mostly by trial and error. In summary, the final gain values for my PID controller were:

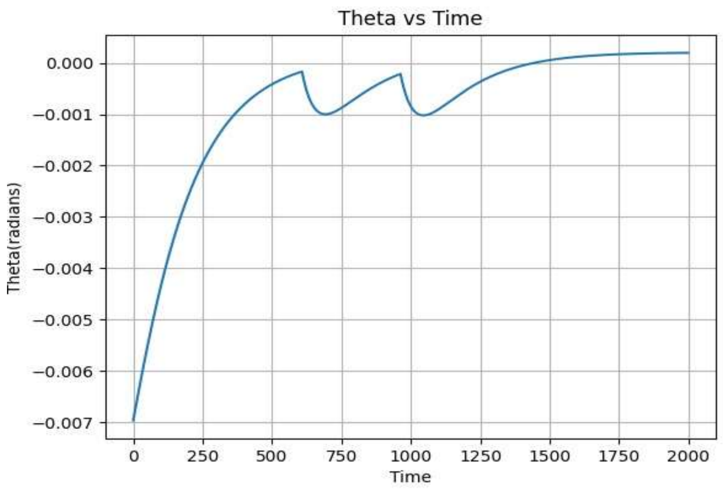
- Kp = 1.97

- Kd = 0.41

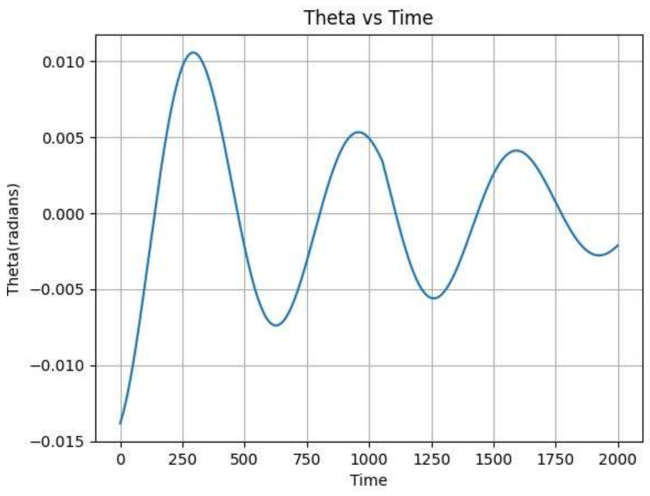
- Ki = 0.005.

This is the graph formed after final tuning:

Over here, we can see that it takes some time for it to adjust system and the theta and it produces oscillation but after 12.5 seconds, it nearly becomes a constant with value nearly zero (Our goal).

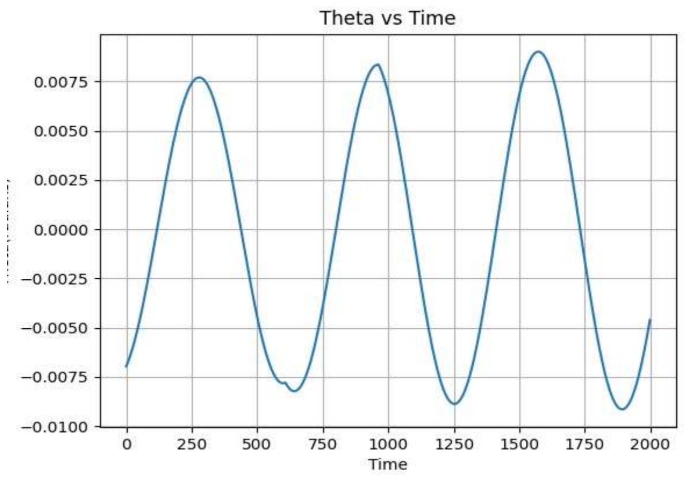


The graph before tuning was with oscillations was like this:-

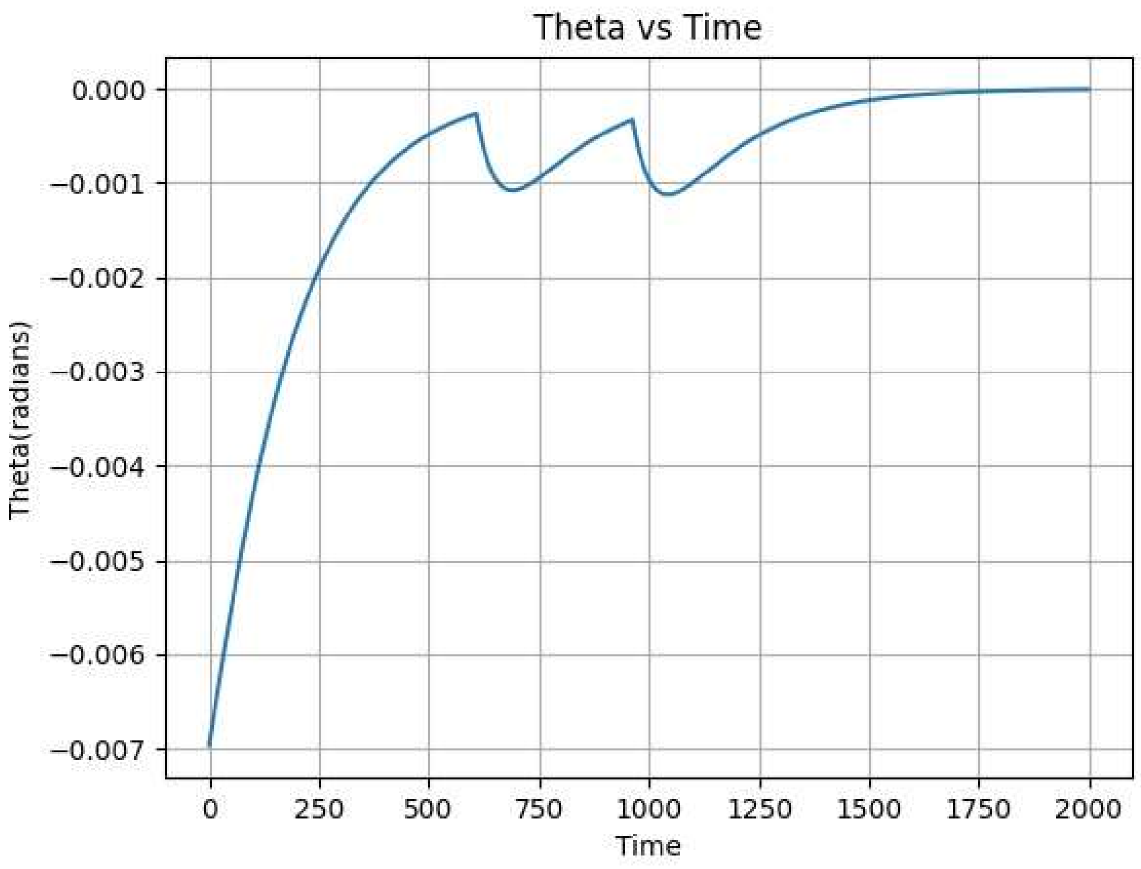


B. Here's an overview of my tuning process for various controllers, the outcomes I observed, and the obstacles I encountered. Accompanying each controller's summary is a graph showing the pendulum angle versus time over a 10-second span.

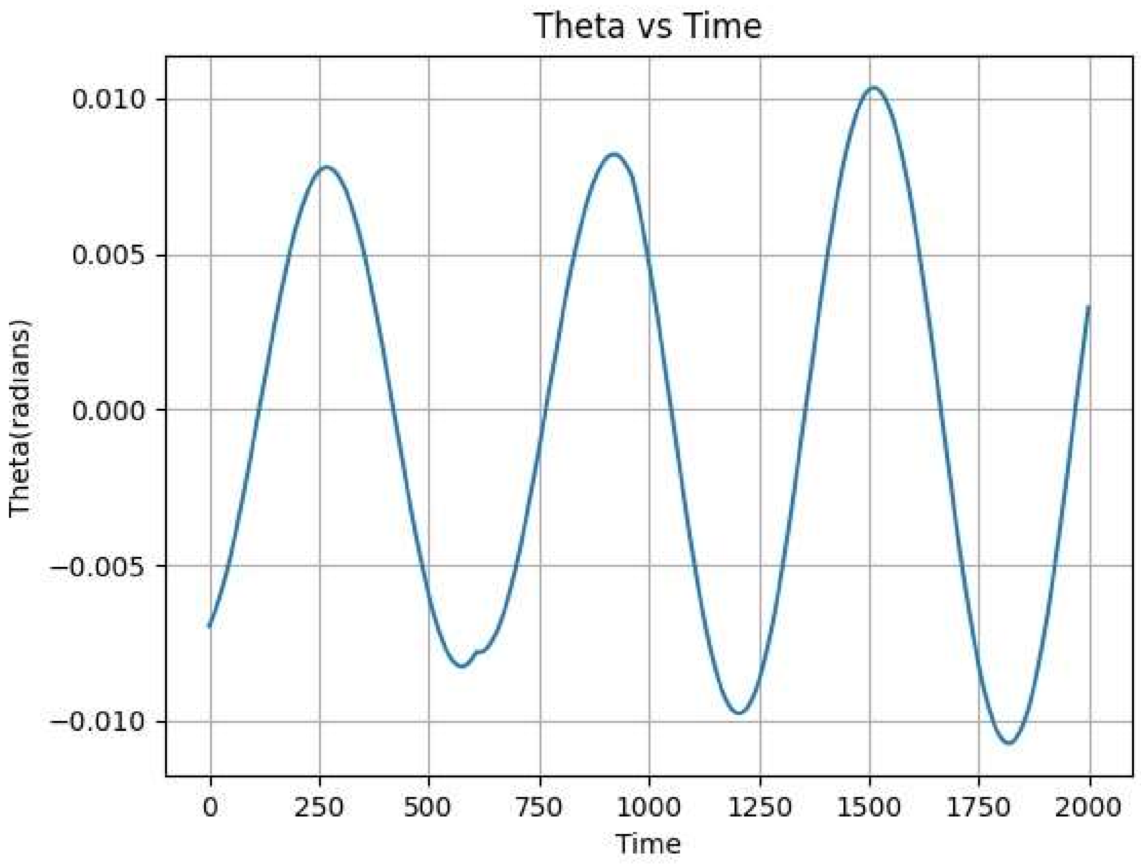
a. P-controller: I first experimented with this controller type. In my tuning efforts for Kp, I noticed that smaller values seemed more effective, but anything less than 1.5-1.6 was unsatisfactory, this is because, too low proportional gain made the system sluggish in error correction, while an excessively high one led to more swings in theta. After extensive adjustments, I settled on a kp = 1.9 as the optimal value. It was challenging to pinpoint this value because a better response time led to wider oscillations, pushing me to continuously search for an equilibrium.
Graph for P-controller:-



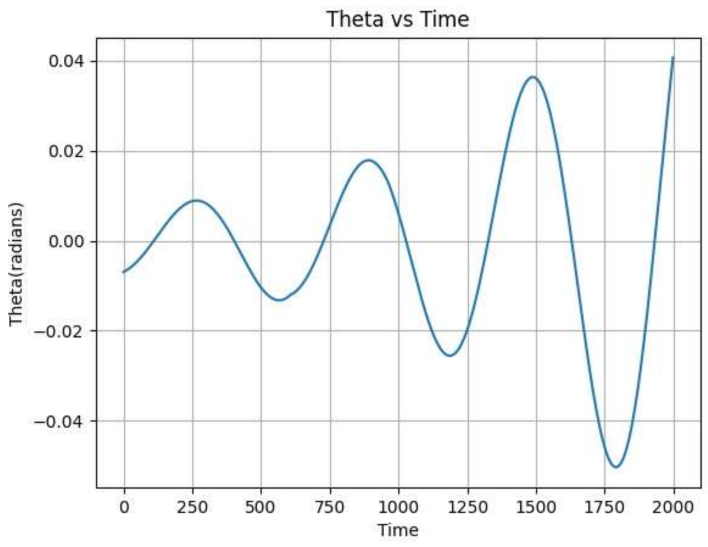
b. PD Controller:
This was easier to tune then P and PID controller and does give us better output then others, we can see that:-
Adjusting this controller was relatively straightforward. I started with the same Kp value from the P-controller. Setting Kd to 0.43 yielded a more stable response without notable oscillations and just a minor overshot. The negative values and higher positive did not balance the pendulum and Kd around 0.5 was more accurate.



c. PI Controller:
I had a lot of problems trying to tune this one, I knew from previous parts that Kp around of 2 is stable and because of that, I started using Kp as 2 and stared to adjust the values of Ki. I observed for small values of Ki around 0.009, it oscillates a lot but it does balance the pendulum, as visible in the graph below:-



When I tried to increase the Ki value significantly that is somewhere around 0.1, and it made the theta even bigger as we can see here:-

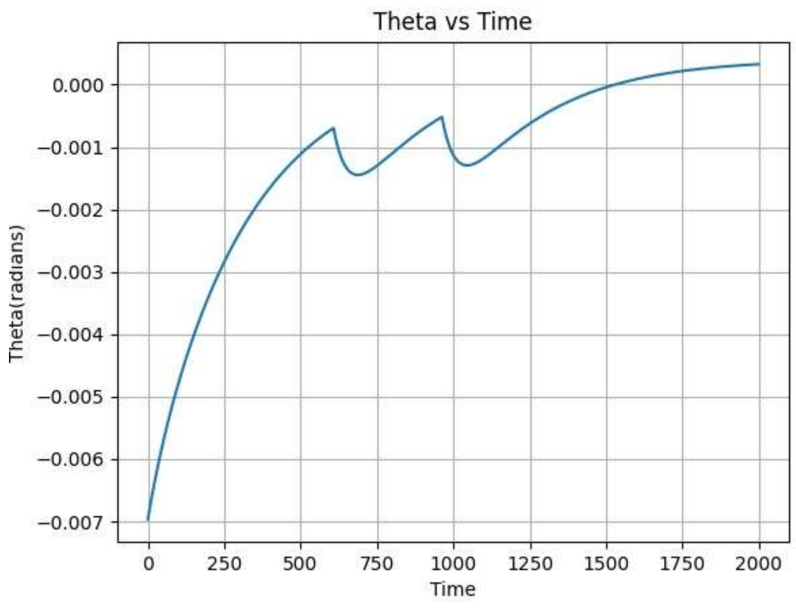


On further increasing the Ki, we can no longer balance the pendulum.

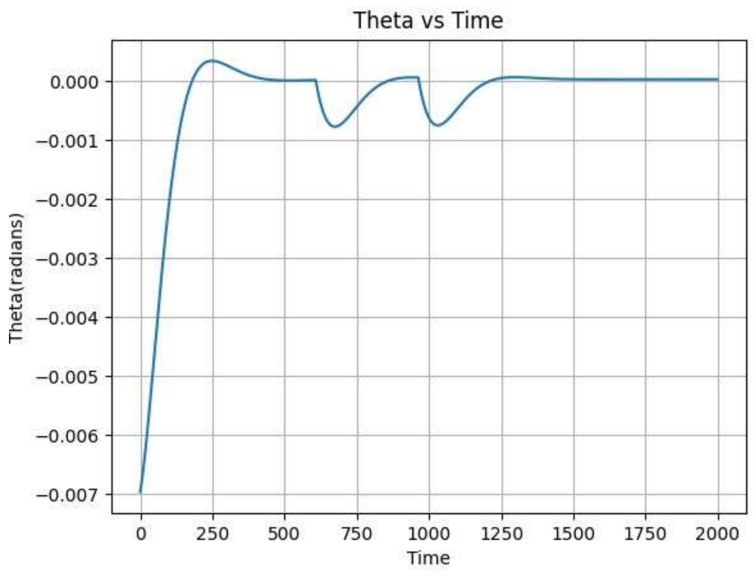
d. DI Controller:-

This controller struggles to stabilize the system effectively. Through trial and error and my understanding, I determined the optimal values to be around Kd = 25 and Ki = 43. Yet, the system remains unsteady with this setup. This instability stems from the absence of immediate error correction. The derivative adjustment focuses on error rate changes without considering the target value. Meanwhile, the integral accounts for the accumulated error over a period. However, without addressing the current error directly, the system doesn't always achieve stability in every situation.

Part C:-
Gravity 10:

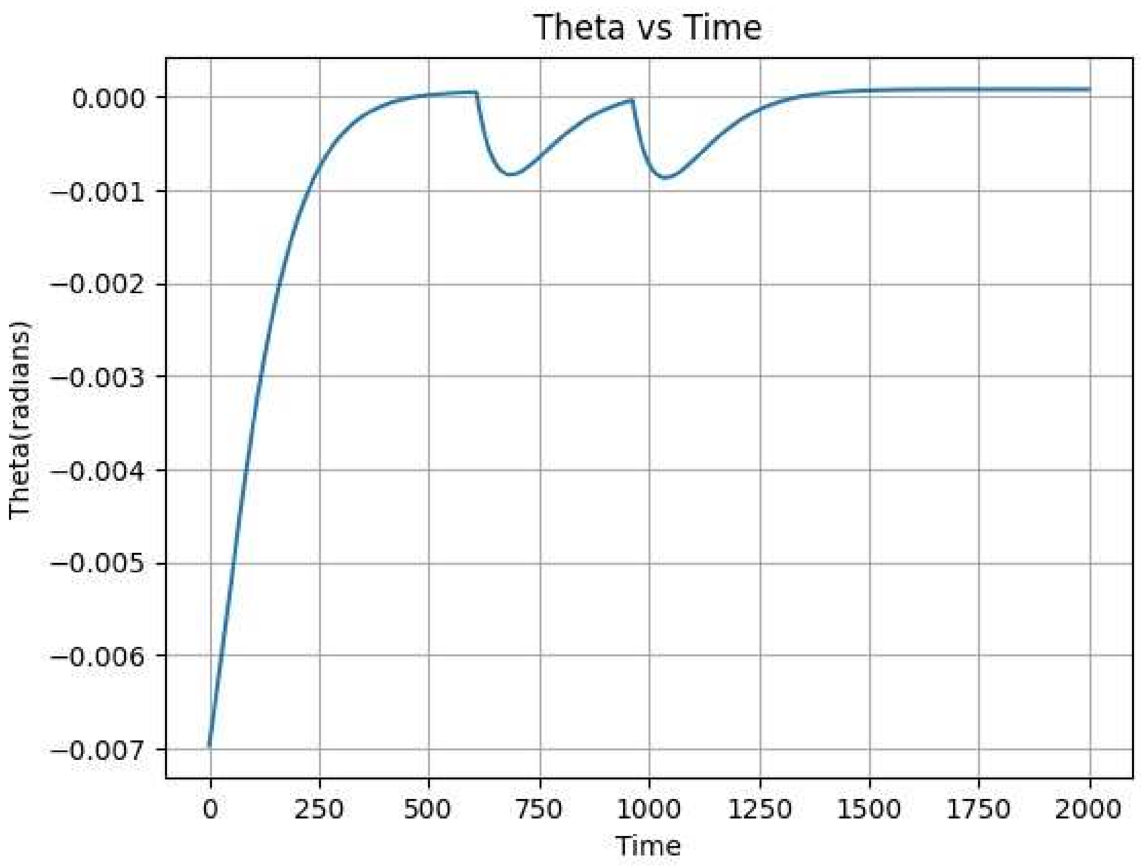


Gravity 6.0:



Over here, we can see that for higher gravity, it takes more time for theta to stabilize and with more gravity, the wavelength of oscillations is also higher than that of lower gravity.
But the pendulum is balanced by the system in both the case. When we increase the gravity by a lot, like 20, the system stops balancing the pendulum as well, meaning our system performs better at lower gravity values than the higher ones.
Now for Changing mass, the graph looks like:
For Mass of pendulum = 1.0,
As we increase the mass, the cart seems to not balance it for 10s and it falls off.

For Pendulum of mass =0.3:-

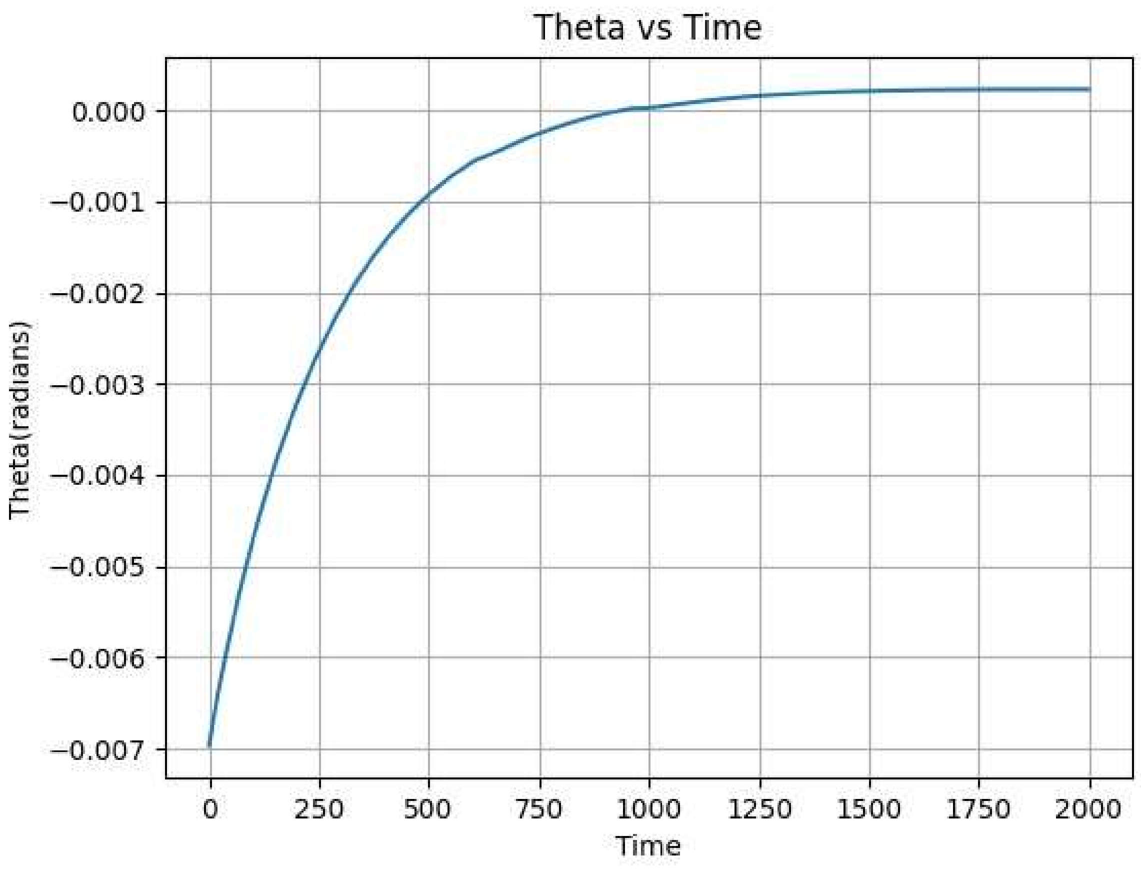


From the results, we can say that our system works better in low gravity and low mass of pole system, as the weight and gravity increases, our system does not perform that well.

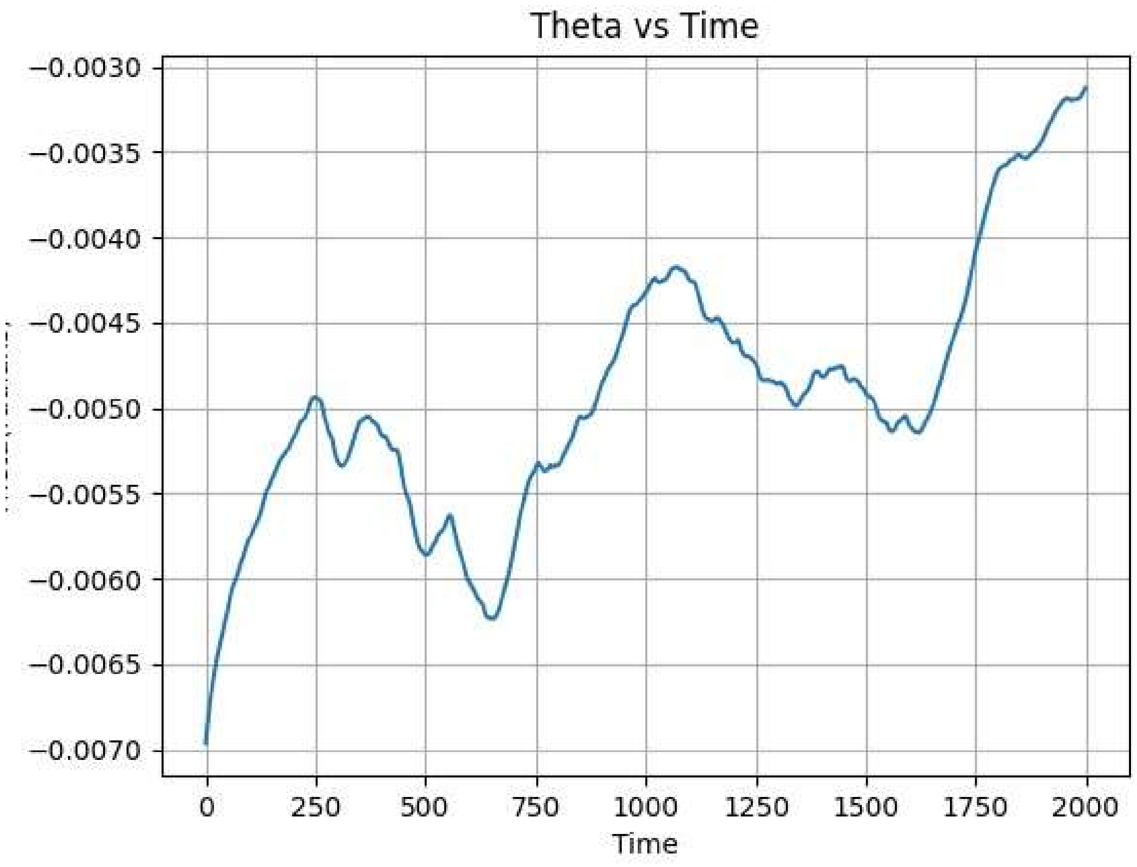
Part D:
I have added disturbances in PID :

```
np.random.rand() <=0.999,
action = (Kp * error) + (Kd * derivative) + (Ki * integral)
      else:
         action = 0.003
```

here, action is 0.003 because large action will make our system imbalanced, and our system will not work. The graph looks like:



When we, over here:- np.random.rand() <=0.999,
Decrease the number from 0.999 to 0.9, the system performance decreases, as we can see in the graph:-
This is because now, the random action is performed more times than previously, meaning the more times we perform the random action, the worse the performance gets, like if the action is 0.009 or something, it does not even end up balancing the pole till the end of the time.



Please note that all the graphs before this, were also with the random action added, the action step was 0.1 and therefore, we see the disturbances at the start of time. With no random action, the graph would look like the 2nd last graph.