1 Introduction

In this final project, with **FeedbackControl** function implemented, I was able to simulate the pick and place task by KUKA youBot in the following scenarios: *overshoot*, *best*, and *newTask*. Moreover, I also added joint limit to avoid self-collision and and tolerance for singularities, which is implemented in *improved*. The scripts that one can run to get the results sit in the main directory.

2 Code Structure

There are 3 main sections in my codebase. First, a configuration file called `milestone_config.py` which contains the global default values used for initialization of the tasks across different scenarios. Meanwhille, there is another file called `milestone_func.py` which has the functions that I implemented in the milestones. With these two files, the variables and the functions can be easily called from different scripts. After you download and unzip the folder, you can run the main scripts `overshoot.py`, `best.py`, `newTask.py`, `improved.py` by

```
cd Ruan_Aria_Final_Proj
conda env create -f environment.yaml
conda activate MR_env
python {script_name}.py
```

As a result each main script will generate a log.txt (where the program progress is documented with timestamps), a README.txt (which includes parameter values used for this specific scenario), an error plot in png format that can be viewed directly, one result.csv for simulation, and an error.csv to record error. All these file mentioned above can be find in the `result/script_name` directory, along with a video recording of the simulation.

3 Implementation

Next, I will discuss the implementation of the functionalities described above in each main script.

3.1 Initialization

Given the scenario name, the initialization is done by reading from the `task_config` dictionary where it stores all the controller parameters. Noticeably, I split the angular and the linear gain in the feedback controller, since these two values are in different units and would make more sense to have separate k_p and k_i.

3.2 Trajectory Generation

The trajectory is generated by calling the **TrajectoryGenerator** function implemented in Milestone 2.

3.3 Processing Trajectory

Here I loop through the entire generated trajectory and use **FeedbackControl** function to apply controllers that is characterized by

$$\mathcal{V} = [Ad_{X^{-1}X_d}]\mathcal{V}_d(t) + K_p X_{err}(t) + K_i \int_0^t X_{err}(t)dt$$

Given the twist $\mathcal{V}$ and the end-effector Jacobion J_e, I then calculate the control input with

$$[u, \dot{\theta}]^T = J_e^{\dagger}(\theta)\mathcal{V}$$

where I added tolerance of 0.01 during the calculation of pseudoinversing $J_e^{\dagger}$.
And with the control input, I plug these values into function **NextState** in Milestone 1. However, I made some changes to the original function where I added a function called **testJointLimits** to limit joint range of motion and thus avoid self-collision. I will discuss more details in Section 4 Result.
As a result, I was able to get a new set of trajectories that are the controller response and simulate the pick and place tasks with those.

4 Result

In this section, I will demonstrate and discuss the result that I got from Section 3 Implementation. Overall, the errors in each scenario managed to converge to 0. Of all the scenarios, singularities prevention was enabled and self-collision avoidance was used in *improved*. There was also a constraint for the wheel speed of 10m/s.

4.1 Overshoot

In order to get overshoot, I set both angular and linear k_p to be relatively large, where $k_{p,lin} = 1$ and $k_{p,lin} = 1$. Despite the overshoot, the controller was able converge to 0 at around 5 seconds and the simulation video also were aligned with the result shown in Figrue 1.

4.2 Best

The overshoot system as a starting point, I tuned the controller (decreasing the gains) via trial-and-error and found the optimal controller parameters, which had a fast response without any overshooting. The errors were plotted in Figure 2.

4.3 NewTask

I specified a new cube configuration (x, y, ϕ) for *newTask*. The initial configuration became $(0, 0.5, 0)$ and the end pose of the cube was $(1, 1.5, 0)$. The errors still converged to 0 quickly without any significant overshoot shown in Figure 3.

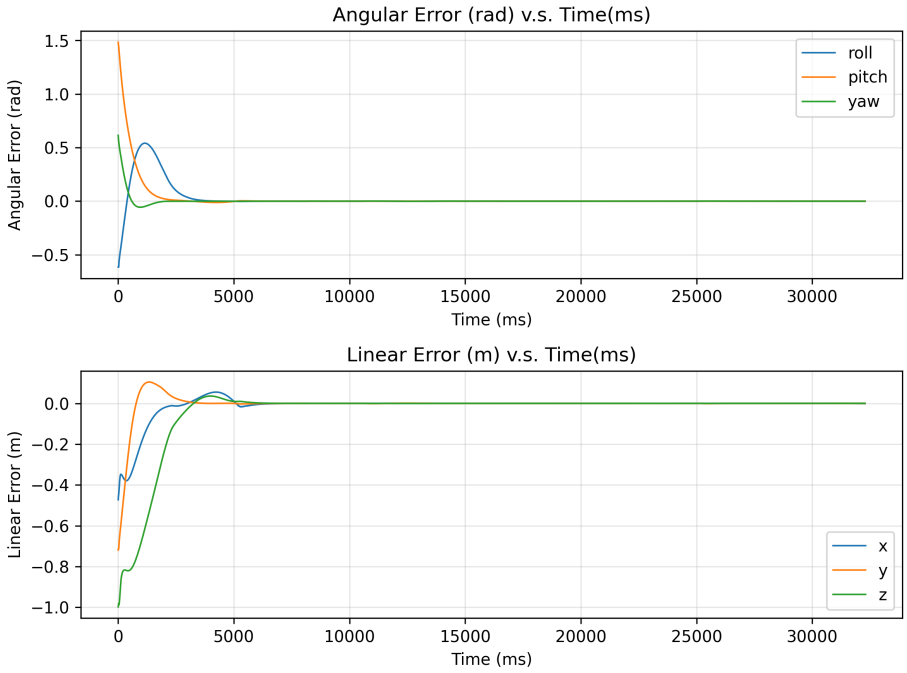

Figure 1: Error plot of *overshoot*

4.4 Improved

While recording *overshoot* and *best*, I noticed some potential self-collision configurations in the first 2 screenshots in Figure 4. And that was when I decided to apply range of motion limit on each arm joint. Even though there were some fluctuations and overshooting at first, it eventually converged to 0 and, most importantly, it was able to maintain a safe pose at the time step when it was previously prone to self-collision. The comparison is shown in Figure 4.

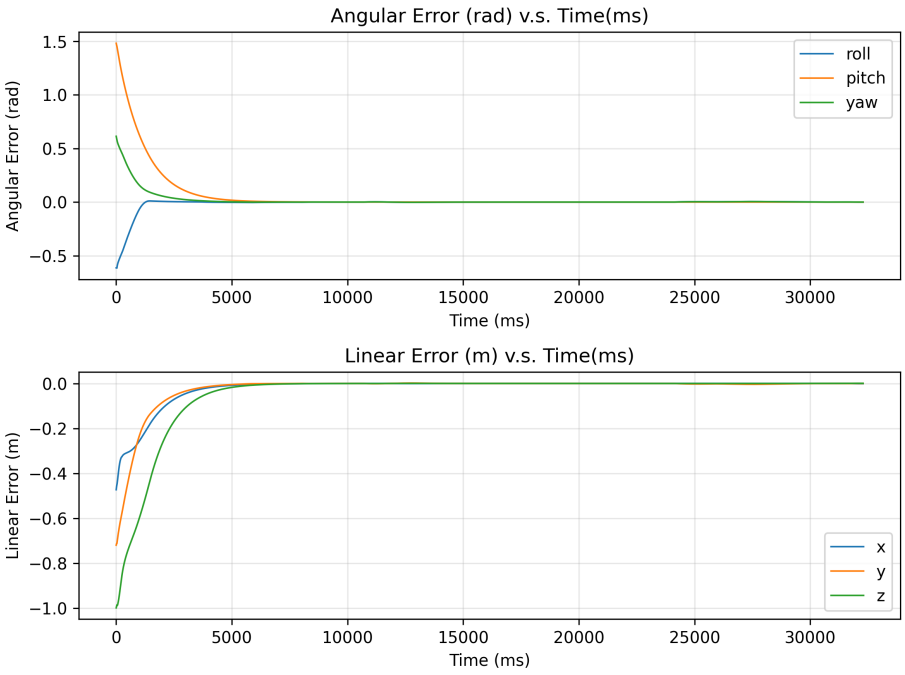

Figure 2: Error plot of *best*

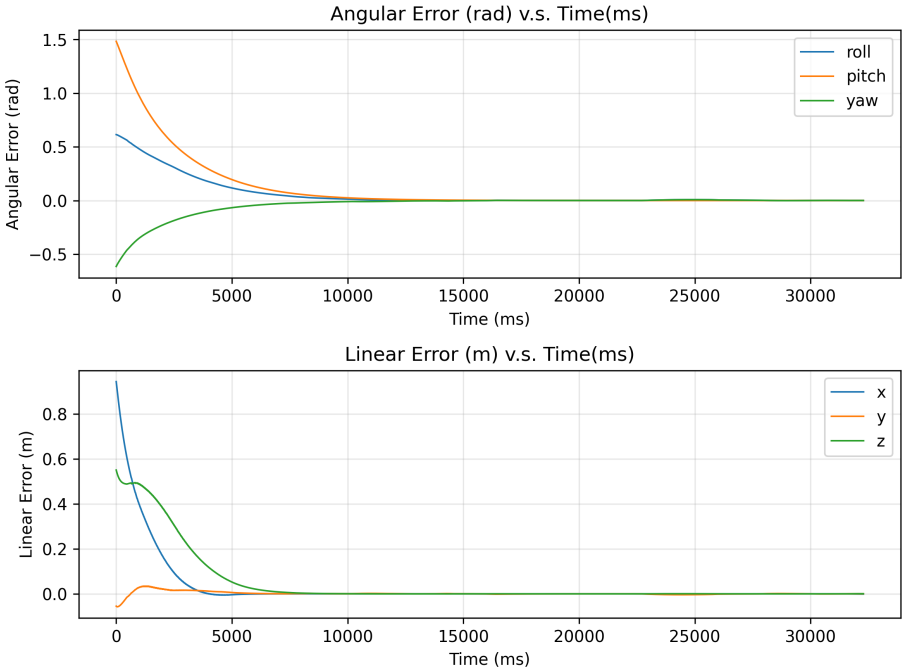

Figure 3: Error plot of *newTask*

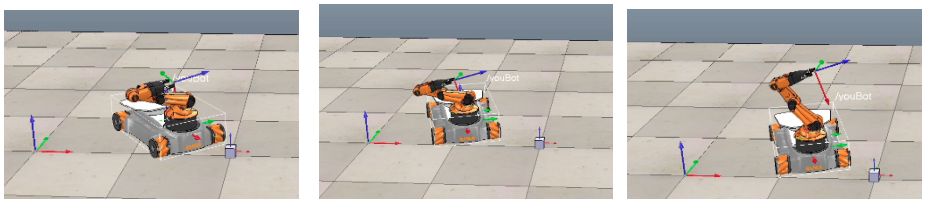

Figure 4: Robot configuration (left: *overshoot*, middle: *best*, right: *improved*)

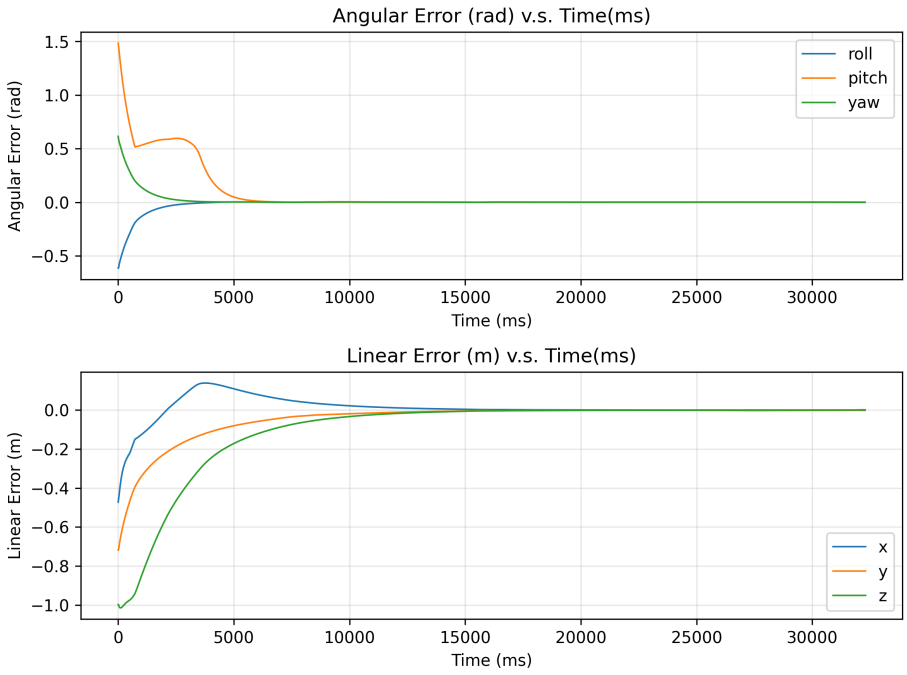

Figure 5: Error plot of *improved*