Motion Control of a Mobile Manipulator Robot

Shrey Kansal
Mechanical and Aerospace Engineering
University of California San Diego
La Jolla, US
skansal@ucsd.edu

Rachit Chhabra
Mechanical and Aerospace Engineering
University of California San Deigo
La Jolla, US
rachhabra@ucsd.edu

I. Introduction

There has been rapid development in the field of self-driving cars, automated logistics, drones, and manufacturing techniques. All these areas can be justifiably grouped into what we call Robotics. Regardless of the area of application, motion control has been at the core of research and development for robotics. As the name suggests, motion and control are essential for a robot to function with as much dexterity as humans. As part of the capstone project, we implement feedback control law for motion control of a mobile manipulator robot in Python.

II. Problem Formulation

Our problem statement consists of designing a motion control algorithm for the manipulator mobile robot. For this, we use a Feedback control law algorithm in which we try to minimize the error between the desired pose and the predicted pose across all timesteps.

The feedback gain matrices (K_p, K_i) are set such that the robot can achieve the designed task and the error converges to 0 by the end of time loop.

In the last part of this report, we compare the error results with respect to different feedback gain matrices and achieve a different task as well.

III. Technical Approach

The project for motion control of mobile manipulator robot has been divided into 3 components (Kinematic Simulator, Reference Trajectory Generator and Feedforward Plus Feedback Control), and has been tied together in end using a wrapper script.

A. Kinematic Simulator

To simulate the kinematics of the robot, we build a python function by the name of NextState. The NextState function uses the kinematics of the youBot to predict how the robot will move in a small timestep given its current configuration and velocity.

We provide 4 inputs to the function as follows:

1. The current state of the robot (12 variables, 3 for chassis (ψ, x, y), 5 for arm (θ_i), 4 for wheel angles (ϕ_i))
2. The joint and wheel velocities (9 variables, 5 for arm ($\dot{\theta}_i$), 4 for wheels ($\dot{\phi}_i = u$))
3. The timestep size (1 parameter, τ)
4. The maximum joint and wheel velocity magnitude (1 parameter)

First, we sort the input data into desired form, i.e., current pose of the end-effector (T_{se}), the arm joint angles, wheel angles and their respective velocities. Next, we limit the joint velocities to 10 units so that the robot's state does not overshoot from the desired state. Next, the body twist ($\mathcal{V}_b$) is computed using the following formula:

$$\mathcal{V}_b = F\Delta\theta = \frac{r}{4}\begin{bmatrix} -\frac{1}{l+w} & \frac{1}{l+w} & \frac{1}{l+w} & -\frac{1}{l+w} \\ 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix}\Delta\theta$$

The new arm joint angles and wheel angles are determined in the following manner:

$$\theta_{i,new} = \theta_{i,old} + \dot{\theta}_i\tau$$
$$\phi_{i,new} = \phi_{i,old} + \dot{\phi}_i\tau$$

The new pose, i.e., the next state is computed by transforming through matrix exponential formulation as follows:

$$T_{se,new} = T_{se}\, e^{[\mathcal{V}_{b6}]}$$

The parameters (ψ, x, y) from the new predicted pose, i.e., $T_{se,new}$, new arm joint angles and wheel angles (12 state variables) are returned as output from the NextState function.

B. Reference Trajectory Generator

To generate a desired or reference trajectory for the robot's end effector to follow, we build a python function by the name of TrajectoryGenerator.py. The TrajectoryGenerator function uses eight poses at which the end effector should reach to build a path around these waypoints.

We provide the following inputs to our function:

1. The initial configuration of the end-effector ($T_{se,initial}$).
2. The initial configuration of the cube ($T_{sc,initial}$).
3. The desired final configuration of the cube ($T_{sc,final}$).
4. The configuration of the end-effector relative to the cube while grasping ($T_{ce,grasp}$).
5. The standoff configuration of the end-effector above the cube, before and after grasping, relative to the cube ($T_{ce,standoff}$).
6. The number of trajectory reference configurations per 0.01 seconds: k. The value k is an integer with a value of 1 or greater.

We use the ScrewTrajectory function in Modern Robotics code library to construct a path in the form of series of poses. In total, we get 8 trajectories as follows:

1. Moving the gripper from its final configuration to a 'standoff' configuration a few cm above the block.
2. Moving the gripper down to the grasp position.
3. Closing the gripper.
4. Moving the gripper back up to the 'standoff' configuration.
5. Moving the gripper to a 'standoff' configuration above the final configuration.
6. Moving the gripper to the final configuration of the object.
7. Opening the gripper.
8. Moving the gripper back to the 'standoff' configuration.

In total, we get 1760 poses for the complete path from initial to final configuration.

We concatenate the poses along with gripper state (0: open, 1: close) into an array of 13 columns as follows:

$$Trajectory = r_{11}, r_{12}, r_{13}, r_{21}, r_{22}, r_{23}, r_{31}, r_{32}, r_{33}, p_x, p_y, p_z, gripper\_state$$

Finally, we save the reference trajectory in a csv file for it to be read later.

C. Feedforward Plus Feedback Control

In the third component, we write a FeedbackControl function that calculates the task-space feedforward plus feedback control law.

We pass the following inputs into the function:

1. The current *actual* end-effector configuration $X\ (aka\ T_{se})$.
2. The current *reference* end-effector configuration $X_d\ (aka\ T_{se,d})$.
3. The reference end-effector configuration at the next timestep $X_{d,next}\ (aka\ T_{se,d,next})$.
4. The PI gain matrices K_p and K_i.
5. The timestep τ between reference trajectory configurations.
6. The current and next desired poses from trajectory function ($X_d, X_{d,next}$).

First, we determine the adjoint of robot base, arm and the end-effector as follows:

$$J_{base}(\theta) = \left[Ad_{T_{0e}^{-1}(\theta)\, T_{b0}^{-1}(\theta)}\right] F_6$$

Simultaneously, we get the X_{error} and $X_{error,integral}$ as follows:

$$X_{error} = \log(X^{-1} X_d) \in \mathbb{R}^6$$

$$X_{error,integral} = \int_0^t X_{error}(t)\, dt$$

Next, we get the desired twist as follows:

$$\mathcal{V}_d = \frac{\log\left(X_d^{-1} X_{d,next}\right)}{\tau}$$

and the adjoint between actual and desired state as:

$$Adj = [Ad_{X^{-1} X_d}]$$

Next, the feedforward is determined as:

$$feedforward = Adj\ \mathcal{V}_d$$

The twist is then computed as:

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

and,

$$\begin{bmatrix} u \\ \dot{\theta} \end{bmatrix} = J_e^{\dagger}(\theta)\mathcal{V}$$

where,

$$J_e(\theta) = [J_{base}(\theta) \quad J_{arm}(\theta)]$$

Finally, we return the end-effector twist $\mathcal{V}$, and joint and wheel velocities $(u, \dot{\theta})$.

D. *Wrapper Script*

The goal of the wrapper script is to tie the 3 functions together. First, we initialize the desired and actual initial end effector pose as:

$$T_{se,desired} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{se,actual} = \begin{bmatrix} 0.5 & 0.866 & 0 & 0.1 \\ -0.866 & 0.5 & 0 & -0.2 \\ 0 & 0 & 1 & 0.0963 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This ($T_{se,actual}$) corresponds to a 30 degrees of orientation error, 0.1m position error in X direction, 0.2m in Y direction and 0.4m in Z direction.

And for the robot chassis, the initial configuration is initialized as:

$$config = [0,0,0,0,0,0.2,-1.6,0,0,0,0,0,0]$$

Now, we build a loop over the number of poses we get from our GenerateTrajectory Function. Within this loop, we first pass our reference trajectories in the feedback control function and get the error at that point in time. Now, we get the next state based on the velocities returned by the feedback control function and pass it again into the feedback control function again.

Post this, we accumulate the total error into an array so that it can be plotted and analyzed.

IV. Results

A. *Best Case*

For the best case, Feedforward-plus-PI controller was implemented with the following feedback gains

$$K_p = 5$$
$$K_i = 0.5$$

The results for the error convergence can be visualized in Fig.1 and Fig.2.

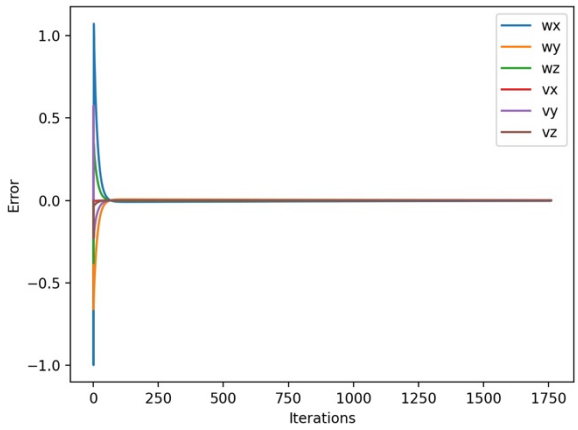

Figure 1 Best Case error

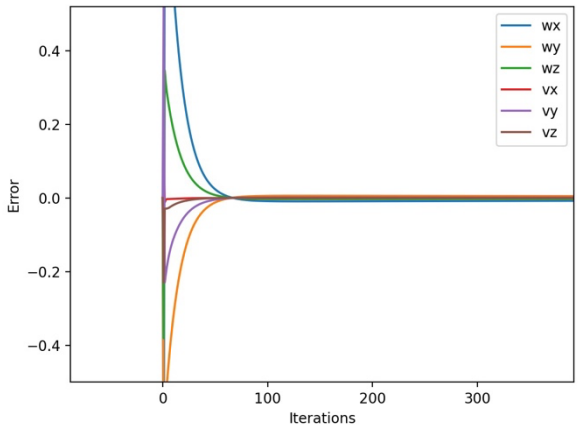

Figure 2 Best Case error (zoomed)

The video for the same can be found below:

Best Case Video

It is seen that the error converges before the 400 timesteps, i.e., when the first part of trajectory completes. This seems correct because the error should

converge before reaching the first destination, otherwise the following tasks will be erroneous.

B. *New State Case*

For the new state case, Feedforward-plus-PI controller was implemented with the following feedback gains

$$K_p = 5$$
$$K_i = 0.5$$

The cube was positioned as follows:

$$T_{cube,initial} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0.025 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{cube,final} = \begin{bmatrix} 1 & 0 & 0 & 1.5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.025 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The results for the error convergence can be visualized in Fig.3 and Fig.4.

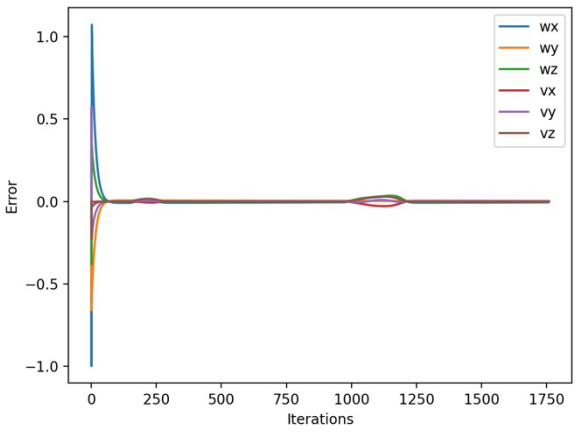

Fig. 3 New State Case error

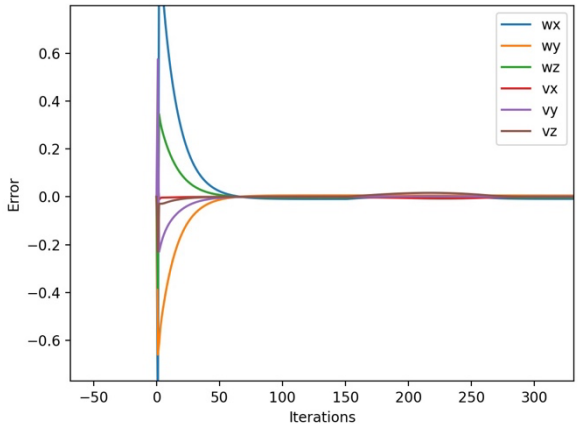

Fig. 4 New State Case error (Zoomed)

The video for the same can be found below:

New State Case

It is seen that the error converges before the 400 timesteps, i.e., when the first part of trajectory completes. This seems correct because the error should converge before reaching the first destination, otherwise the following tasks will be erroneous.

C. *Overshoot Case*

For the Overshoot case, Feedforward-plus-PI controller was implemented with the following feedback gains

$$K_p = 3$$
$$K_i = 7$$

The results for the error convergence can be visualized in Fig.5 and Fig.6.

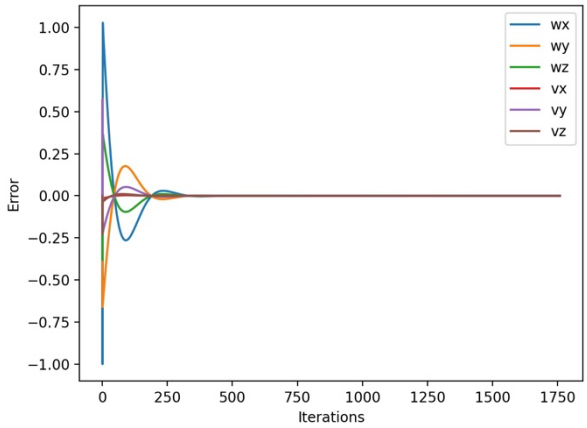

Fig. 5 Overshoot Case error

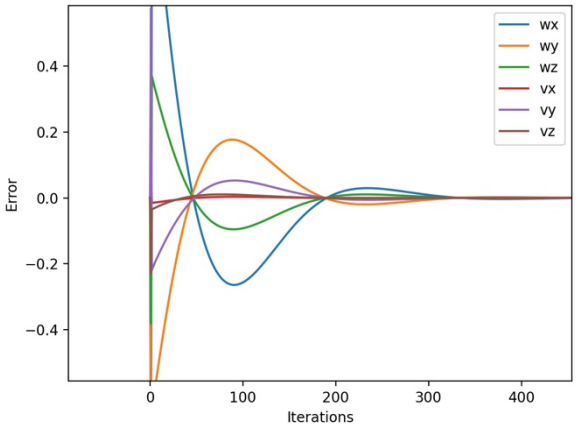

Fig. 6 Overshoot Case error (zoomed)

The video for the same can be found below:

Overshoot Case

It is seen that the error converges before the 400 timesteps, i.e., when the first part of trajectory completes. However, the error is seen to be oscillating, which results in slightly erratic trajectory for the robot until the first timestep.

The error is oscillating because we increase the feedback gain values, which are not optimal, and still manages to stabilize before the first destination is reached, which seems correct because otherwise the following tasks will be erroneous.