MAE 204: Final Project

Chockalingam Senthil Rajen, Varun Vupparige
Mechanical and Aerospace Engineering
University of California San Diego
La Jolla, USA

I. Introduction

Robotics systems perceive their environments through sensors, and manipulate their environment through things that move. Examples of successful robotic systems include mobile robots for planetary exploration, industrial robotics arms in assembly lines, autonomous driving cars and manipulators that assist surgeons. Robotics systems are situated in the physical world, obtain information from their local environments through on-board sensors, and make reactions. In this project we write a control algorithm to control the motion of youBot, a mobile manipulator robot with four omnidirectional wheels and a 5R arm. The output of this algorithm are the configurations of chassis, arm and gripper state at each time step. Finally, we simulate the control action of the youBot by loading the configurations data into the CoppeliaSim simulator.

II. Technical Approach

In this section, we detail our approach to compute the control action of the youBot to pick and place a block from its start position to its end position.

A. Reference Trajectory (Milestone 2)

Trajectory is the specification of robot position as a function of time. In this part, we consider the trajectory to be a list of configurations describing the geometric description of each state of the gripper. Firstly, we compute the transformation matrices of each sequence. The sequence of trajectories are as follows:

- Move the gripper from its initial configuration to a "standoff" configuration a few cm above the block.
- Move the gripper down to the grasp position.
- Close the gripper
- Move the gripper back up to the "standoff" configuration.
- Move the gripper to a "standoff" configuration above the final configuration.
- Move the gripper to the final configuration of the object.
- Open the gripper.
- Move the gripper back to the "standoff" configuration.

We compute the transformation matrix for each sequence of trajectory as listed above. Using the transformation matrix for each of these 8 trajectory segments and $ScrewTrajectory$ function from the Modern Robotics package, we generate a list of configurations of the gripper at each timestep using quintic time scaling.

The output of the $ScrewTrajectory$ function is stored as a csv file where each line consists of 13 variables which are in the order: r_{11}, r_{12}, r_{13}, r_{21}, r_{22}, r_{23}, r_{31}, r_{32}, r_{33}, p_x, p_y, p_z, $gripperstate$. The .csv file is loaded into the scene 8: gripper to test the reference generator function and video link can be found in the appendix section (Milestone 2).

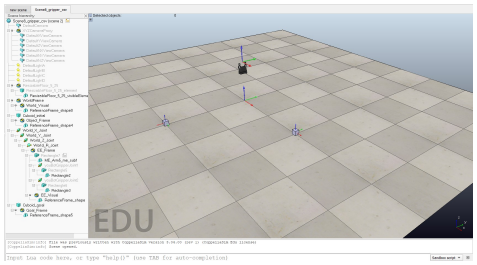

Fig. 1. Scene 8: Gripper (CoppeliaSim

B. Next State (Milestone 1)

In this section, we estimate the chassis configurations from the sensor data. We assume that each wheel of an omnidirectional robot, and each rear wheel of a drive or car, has an encoder that senses how far the wheel has rotated in its driving direction. Odometry data typically accumulates as the robot moves over time. Techniques such as Particle Filer, Extended Kalman Filter and neural networks have been researched extensively to accurately estimate the state of robot.

If the wheels are driven by stepper motors then we know the driving rotation of each wheel from the steps we have commanded to it. The goal of this part is to estimate the chassis configuration of the next time step given the current state. For a four-mecanum-wheel robot, the body twist V_b:

$$\mathcal{V}_b = F\Delta\theta \tag{1}$$

$$\mathcal{V}_b = \begin{bmatrix} -1/(\ell+w) & 1/(\ell+w) & 1/(\ell+w) & -1/(\ell+w) \\ 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \Delta\theta \tag{2}$$

The odometery estimate of robot's next configuration is updated as follows:

$$q_{k+1} = q_k + \Delta q \tag{3}$$

We test the $NextState$ function by initializing control signals for each state. The output of $NextState$ function is in the following format: $Chassis_x$, $Chassis_y$, $Chassis_\theta$, θ_1, θ_2, θ_3, θ_4, θ_5, $wheelangles_1$, $wheelangles_2$, $wheelangles_3$,

$wheelangles_4$. The first 3 variables describes the state of the chassis, next 5 variables detail about the joint angles of 5R arm and last variables explain about the wheel angles. The video link can be found in the appendix section (Milestone 1).

C. Feed-forward plus Feed-back control

For controlling the robot we typically assume to direct control using forces or torques at robot joints, and the robot's dynamic transforms those controls to joint accelerations. But for simulation, we take the leverage to control the robot directly using joint velocities.

We express the trajectory in task space and the controller is fed a steady stream of end-effector configurations $X_d(t)$, and the goal is to command joint velocities that cause the robot to track this trajectory.

1) Feed-forward Control: Given the desired joint trajectory $\theta_d(t)$, this is simplest type of control to command the velocity $\dot{\theta}(t)$ as

$$\dot{\theta}(t) = \dot{\theta}_d(t),$$

where $\dot{\theta}_d(t)$ comes from the desired trajectory. This is called a feed-forward or open-loop controller, since no feedback (sensor data) is needed to implement it.

2) Feed-back Control: We just start with P-control, by initially setting an positive-definite diagonal proportional gain matrix K_p.

$$\dot{\theta}(t) = K_p\left(\theta_d(t) - \theta(t)\right) = K_p\theta_e(t)$$

As we increase the gains, proportional control will reduce the error towards zero.

Next, we implement Proportional-Integral controller or PI controller, which adds a term that is proportional to the time-integral of the error:

$$\dot{\theta}_e(t) + K_p\theta_e(t) + K_i\int_0^t \theta_e(\mathrm{t})d\mathrm{t} = c$$

Taking the time derivative of this dynamics, we get

$$\ddot{\theta}_e(t) + K_p\dot{\theta}_e(t) + K_i\theta_e(t) = 0.$$

The drawback of feedback control is that error is required before the joint begins to move. It would be preferable to use our knowledge of the desired trajectory $\theta_d(t)$ to initiate motion before any error accumulates.Hence , we can combine the advantages of feed-forward control, which commands motion even when there is no error, with the advantages of feedback control,which limits the accumulation of error, as follows:

$$\dot{\theta}(t) = \dot{\theta}_d(t) + K_p\theta_e(t) + K_i\int_0^t \theta_e(\mathrm{t})d\mathrm{t} \tag{4}$$

Implementing the above equation in task space, the equation is as follows:

$$\mathcal{V}_b(t) = \left[\mathrm{Ad}_{X^{-1}X_d}\right]\mathcal{V}_d(t) + K_pX_e(t) + K_i\int_0^t X_e(\mathrm{t})d\mathrm{t} \tag{5}$$

After finding the Twist,we use this twist to find the velocities of the arm-joint angles and wheel velocities.

For a mobile manipulator with m wheels and n joints in the robot arm,the end-effector twist V_e in the end-effector frame e is written as

$$\mathcal{V}_e = J_e(\theta)\begin{bmatrix} u \\ \dot{\theta} \end{bmatrix} = \left[J_{\text{base}}\,(\theta)J_{\text{arm}}(\theta)\right]\begin{bmatrix} u \\ \dot{\theta} \end{bmatrix}. \tag{6}$$

The $6*$m Jacobian Jbase(θ) maps the wheel velocities u to a velocity at the end-effector, and the $6*$ n Jacobian Jarm(θ) is the body Jacobian derived using FkinBody function defined in Modern Robotics. The Jacobian Jbase(θ) is given by

$$J_{\text{base}}\,(\theta) = \left[\mathrm{Ad}_{T_{0e}^{-1}(\theta)T_{b0}^{-1}}\right]F_6 \tag{7}$$

Hence. final equation is given as,

$$\begin{bmatrix} u \\ \dot{\theta} \end{bmatrix} = J_e^{\dagger}(\theta)\mathcal{V}. \tag{8}$$

D. Wrapper Code

In this section we combine the functions defined in the previous three sections to generate the desired motion to generate the reference trajectory, simulate robot motion and compute control inputs. Firstly, we make the initiliasations of the these variables: time step (del_t), T matrix of 0 frame wrt b frame, T matrix of b frame wrt s frame, B list of the 5R arm, M matrix of e frame wrt 0 frame, maximum joint speed limits and initial thetalist.

We run the trajectory generation part to generate a list of T matrices for each time step. Now, in the main loop, we initialize the i^th configuration to be state X_d and the $i+1^th$ configuration to be state $X_{d,next}$. We run the feedback control function to compute the wheel and joint velocities, and the error evolution over time. Using these velocities, we update the next state of the robot using the $NextState$ function. For the next iteration, we update the thetalist and current state. Finally, we append gripper state based on the sequence of trajectory.

The output of the $ScrewTrajectory$ function is stored as a .csv file where each line consists of 13 variables which are in the order: r_{11}, r_{12}, r_{13}, r_{21}, r_{22}, r_{23}, r_{31}, r_{32}, r_{33}, p_x, p_y, p_z, $gripperstate$.

In generating the control algorithm for new state, we initialize the start position as (1,1,0) and end position as (1,-1,-pi/2). Based on this, we change the T matrices accordingly to generate the new reference trajectory. We repeat the same steps for generating control actions and next state.

III. Results and Discussion

In this section we present the error evolution for different cases over time. In Fig3, we start simulation using only feed-forward controller. In this case , we make the robot start at the an initial configuration of the robot that puts the end-effector exactly at the configuration at the beginning of the reference trajectory.We see the error accumulates over time and hence the robot does not follow the desired trajectory. Check appendix Case 1 for the Coppelia simulation.

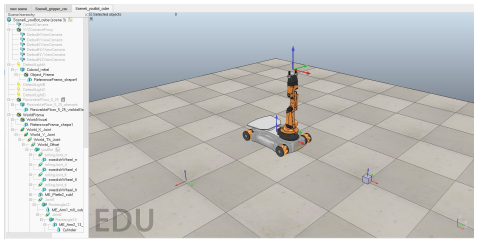

Fig. 2. Scene 6: youBot (CoppeliaSim)

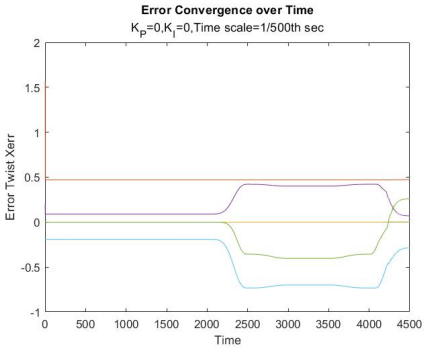

Fig. 3. Error plot for only feed-forward controller

We start setting the gain using small proportional value i.e K_P=1 and K_I=0. In Fig 4, we see that the error settles much better than just feed-forward controller. Check appendix Case 2 for the Coppelia simulation.

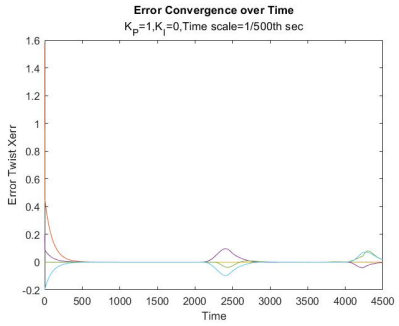

Fig. 4. Error plot for feed-forward plus feed-back controller

After iterating through values of proportional and integral value , we arrive at the best case. Fig 5 shows the error plot with the best controller tuning parameters. We see very less overshoot and achieves error of almost zero.We set the initial configuration of the youBot with 45 degrees of orientation error and 0.2 m of position error. Check appendix Best Case for the Coppelia simulation.

We increase the proportional controller value and in Fig 6 we see there is a huge overshoot.This results in jerky nature while the robot manipulates from the initial to final configuration. Check appendix Overshoot Case for the Coppelia simulation.

In this case,we change the start and final configuration i.e

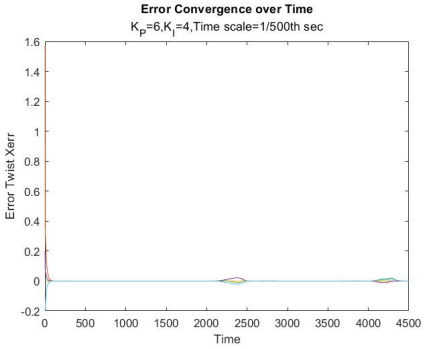

Fig. 5. Best-Case - Error plot

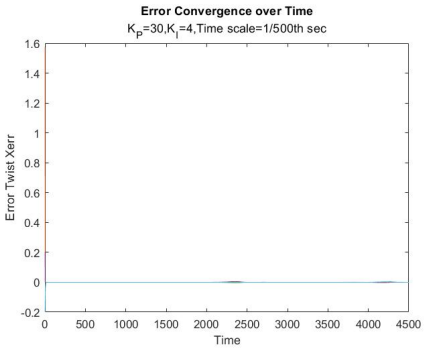

Fig. 6. Overshoot case -Error plot

change the locations of the cube and use the best control tuning parameters (i.e KP=6,KI=4).We see that in Fig 7 controller performs well in this case well. Check appendix New state Case for the Coppelia simulation.

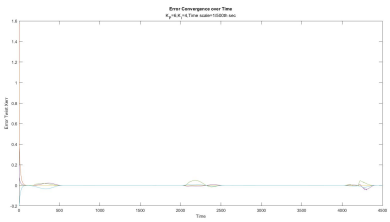

Fig. 7. New State case -Error plot

IV. Appendix

- Milestone 2: https://youtu.be/M0gtFri2spA
- Milestone 1: https://youtu.be/HyU9VQi3Lec
- Case 1 (K_p = 0, K_i = 0): https://youtu.be/9OXJoUogRFw
- Case 2 (K_p = 1, K_i = 0): https://youtu.be/m041o_kKDZk
- Best Case (K_p = 6, K_i = 4): https://youtu.be/8yNOjtRUqck
- Overshoot Case ((K_p = 30, K_i = 4): https://youtu.be/Qmh5OBnZUrU
- New State: https://youtu.be/PzCc6hHafKA