Final Project Ashlee Tiwari

In this project, I wrote a software that plans a trajectory for the end-effector of the youBot, an omnidirectional mobile robot with a 5-DOF robotic arm, performs an odometry as the chassis moves and performs feedback control to drive the robot in the desired trajectory.The simulate planned trajectory in demonstrated in CoppeliaSim.

Milestone 1

Next_State.py - computes the configuration of the robot in the next time step.The function inputs the current robot configuration and joint speeds. Given A 12-vector representing the current configuration of the robot, a 9-vector of controls indicating the arm and wheel twist, a timestep Δt and a speed limit, output a 12-vector representing the configuration of the robot time Δt later.The function uses first-order Euler integration to calculate wheel and joint angles at the next time step.

Testing milestone 1:

To test the NextState function, I Simulated the following controls for 1 second and watch the results in the CoppeliaSim capstone scene.

max_ang_speed = 5

dt=0.01

Total time = 1 sec

Test 1: Driving the robot chassis forward

u=[0, 0, 0, 0, 0, 10, 10, 10, 10]

Test 2 : Sliding the robot chassis sideways

u=[0, 0, 0, 0, 0, -10, 10, -10, 10]

Test 3: Spinning the robot chassis in counterclockwise direction

u=[0, 0, 0, 0, 0, -10, 10, 10, -10]

To run the file : python code/next_state.py
The test results are in folder testing_milestones_results/milestone 1

Milestone 2

Trajectory_Generator.p y - generates the reference trajectory for the end-effector frame {e}.The function inputs the initial end-effector configuration, initial cube configuration, desired cube configuration, and number of configurations per second. The function outputs a matrix containing the end effector configuration at all time steps throughout the trajectory.

I am using gripper_state as 0 = open

1 = close

THe trajectory can be divided into smaller goals :

- Initial to standoff position
- Standoff position to grasp position
- Closing the gripper
- Grasp to standoff position
- Standoff to goal position
- From goal position to final position
- open the gripper
- From final position to standoff position

Testing milestone 2:

total time of the motion in seconds=3

time-scaling method : considered fifth-order polynomial

no of points (Start and stop) in the discrete representation of the trajectory :

For driving : 500 points

For picking/dropping : 100 points

The number of trajectory reference configurations per 0.01 seconds i.e k =1

Input for the functions are :

The initial configuration of the end-effector in the reference trajectory:Tse_initial = np.array([[0, 0, 1, 0], [0, 1, 0, 0], [-1, 0, 0, 0.5],[0, 0, 0, 1]])

The cube's initial configuration:Tsc_initial =

```
np.array([[1, 0, 0,     1],
          [0, 1, 0,     0],
          [0, 0, 1, 0.025],
          [0, 0, 0,     1]])
```

The cube's desired final configuration:Tsc_goal =

```
np.array([[0,  1, 0,     0],
          [-1, 0, 0,    -1],
          [0,  0, 1, 0.025],
          [0,  0, 0,     1]])
```

The end-effector's configuration relative to the cube when it is grasping the cube (the two frames located in the same coordinates, rotated about the y axis):

```
Tce_grasp = np.array([[-1/np.sqrt(2), 0,  1/np.sqrt(2), 0],
          [0,             1,  0,            0],
          [-1/np.sqrt(2), 0, -1/np.sqrt(2), 0],
          [0,             0, 0,             1]])
```

The end-effector's standoff configuration above the cube, before and after grasping, relative to the cube (the {e} frame located 0.1m above the {c} frame, rotated about the y axis):

```
Tce_standoff = np.array([[-1/np.sqrt(2), 0,  1/np.sqrt(2),   0],
             [0,             1,  0,              0],
             [-1/np.sqrt(2), 0, -1/np.sqrt(2), 0.1],
             [0,             0,  0,              1]])
```

To run the file : python code/generate_tracjectory.py

Results for the tests are in testing_milestone_results/milestone 2

Milestone 3

Feedback_Control.py - calculates the kinematic task-space feedforward plus feedback control law.

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

$$[V_d] = \frac{1}{\Delta t} log(X^{-1} X_d)$$

V is the calculated end-effector twist

V_d is the reference twist

X is the current end-effector configuration

X_d is the desired end-effector configuration

X_{err} is the error between the current end-effector configuration and the reference trajectory

K_p is the proportional gain

K_i is the integral gain

Once I calculated the end-effector twist, I needed to convert this to commanded wheel and arm joint speeds (u,$\dot{\theta}$). To do so, I used the pseudoinverse of the jacobian matrix:

$$(u,\dot{\theta}) = J_e^{+}(\theta)V$$

Testing milestone 3:

KP = 0

KI = 0
dt = 0.01

curr_config = np.array([0, 0, 0, 0, 0, 0.2, -1.6, 0, 0, 0, 0, 0, 0])

actual end-effector configuration

```
X = np.array([[0.170, 0, 0.985, 0.387],
              [0, 1,     0,     0],
              [-0.985, 0, 0.170, 0.570],
              [0, 0,     0,     1]])
```

end-effector reference configuration

```
Xd = np.array([[0, 0, 1, 0.5],
               [0, 1, 0,   0],
               [-1, 0, 0, 0.5],
               [0, 0, 0,   1]])
```

end-effector reference configuration at the next timestep in the reference trajector

```
Xd_next = np.array([[0, 0, 1, 0.6],
                    [0, 1, 0,   0],
                    [-1, 0, 1, 0.3],
                    [0, 0, 0,   1]])
```

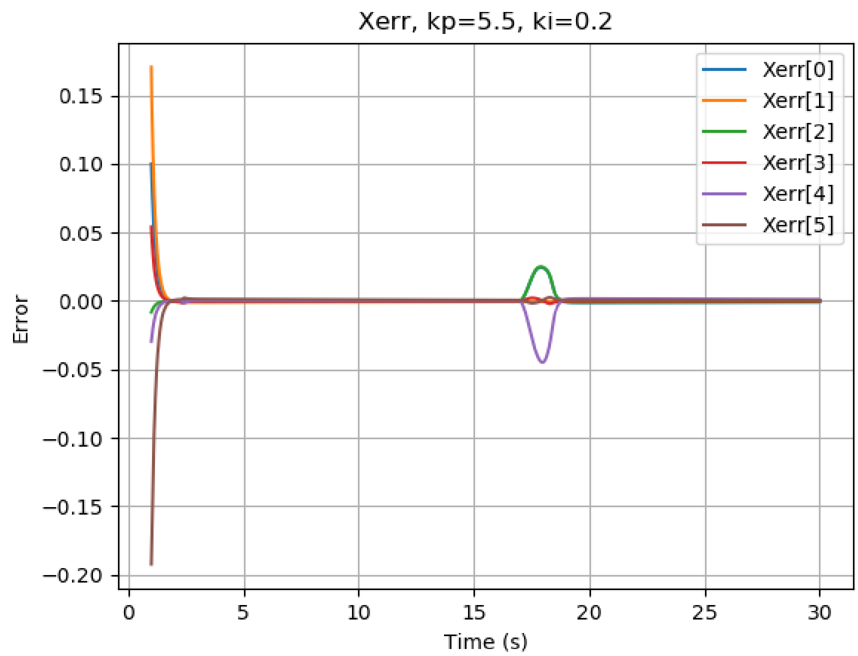
To run the file : python code/feedback_control.py
Results for the tests are in testing_milestone_results/milestone 3

Full code Python code generates a reference trajectory and uses a PI controller to follow the reference trajectory.Code outputs a csv file containing robotic configuration at each time step .

The results for this project can be split into 3 categories:

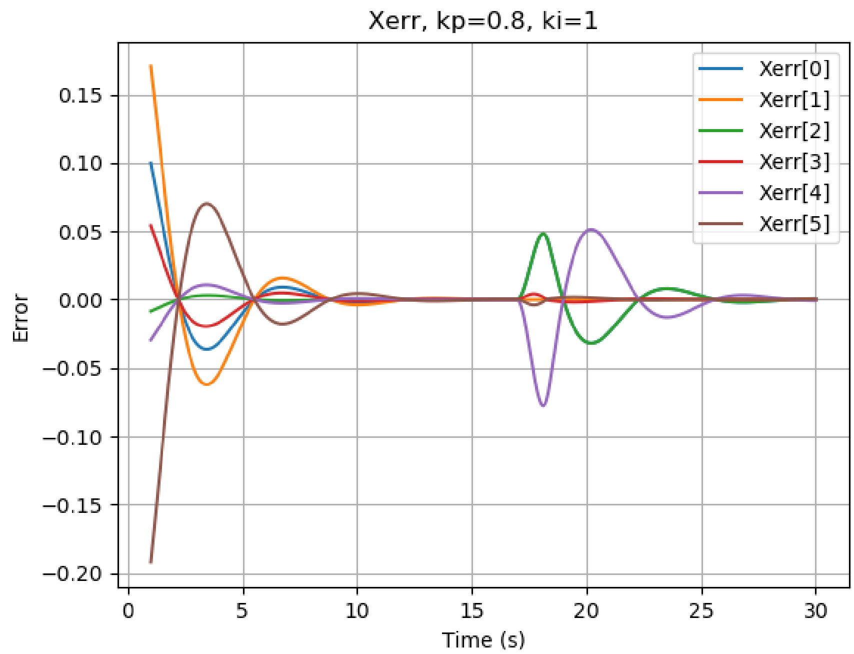
1.)Best : Planning and executing a motion without overshoot or steady-state error.

PI controller with feedback gains of Kp = 5.5 and Ki = 0.2



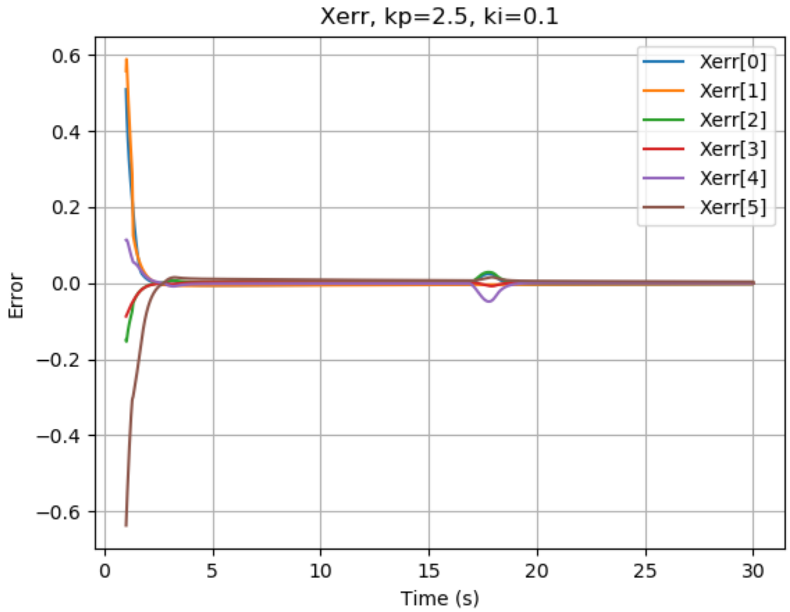
We can see that there is no overshoot, no steady-state error, and fast settling time with a little bump in between after which the plot PI again converges to 0

2.)Overshoot - Planning and executing a motion with overshoot but without steady-state error.
PI controller with feedback gains of Kp = 0.8 and Ki = 1



From the plot we can observe that there is an overshoot at the beginning of the motion and when there is a bump in between .The plot converges with no steady-state error

3.) NewTast results - Planning and executing the trajectory with different start and finish configuration and Kp=2.5 , Ki=0.1



We can see that there is no overshoot, no steady-state error, and fast settling time with a little bump in between after which the plot PI again converges to 0

Additional :

Implementation of singularity avoidance : The tolerance option allows you to specify how close to zero a singular value must be to be treated as zero.

Tolerance =1e-3 added in Pinv : Je_inv = np.linalg.pinv(Je,rcond=1e-3)

Where rcond is used to zero out small entries in a diagonal matrix with positive real values (singular values)

By treating small singular values (that are greater than the default tolerance) as zero, I want tol avoid having pseudoinverse matrices with unreasonably large entries

Implementation of self-collision avoidance

My testJointLimits function is checking for Joint 3 and Joint 4
I check joint limit before calculating Jacobian .If the joint value is higher than the joint limit , I make the column of that joint in Jacobian 0
After implementing the self-collision, controller values need to be further tuned to work properly .That is why the self-collision implementation code is commented out