Reinforcement Learning

– Project 1 –

submitted by

Sanket Ankush Salunkhe

Sanket Ankush Salunkhe
sas9908@nyu.edu
Student ID: sas9908

1 Part 1 - Setting up:

The quadrotor dynamics equations are:

$$\dot{x} = v_x \tag{1}$$

$$m\dot{v_x} = -(u_1 + u_2)sin\theta \tag{2}$$

$$\dot{y} = v_y \tag{3}$$

$$m\dot{v_y} = (u_1 + u_2)cos\theta - mg \tag{4}$$

$$\dot{\theta} = \omega \tag{5}$$

$$I\dot{\omega} = r(u_1 - u_2) \tag{6}$$

Where,

x = Horizontal position

y = Vertical position

θ = Orientation

v_x = Velocity in the horizontal plane

v_y = Velocity in the vertical plane

ω = Angular velocity

u_1, u_2 = Forces produced by rotors

I = Moment of inertia

r = Distance from the center of the robot to the robot frame to the propeller

g = Gravity constant

$$State\_vector = \begin{bmatrix} x \\ v_x \\ y \\ v_y \\ \theta \\ \omega \end{bmatrix} \qquad control\_vector = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

1.1 Discretize the system equation:

The discretized system equations are:

$$x_{n+1} = x_n + \Delta t.v_x \tag{7}$$

$$y_{n+1} = y_n + \Delta t.v_y \tag{8}$$

$$v_{x.n+1} = v_{x.n} + \Delta t[\frac{-(u_1 + u_2)\sin\theta_n}{m}] \tag{9}$$

$$v_{y.n+1} = v_{y.n} + \Delta t[\frac{(u_1 + u_2)\cos\theta_n}{m} - g] \tag{10}$$

$$\theta_{n+1} = \theta_n + \Delta t\omega_n \tag{11}$$

$$\omega_{n+1} = \omega_n + \Delta t[\frac{r.(u_1 + u_2)}{I}] \tag{12}$$

The given system is non-linear and thus can't directly convert into $Ax + Bu$ format. It needs to be linearized at each step and use for further operation.

1.2 Compute u_1^* and u_2^*:

The given initial condition is,

$$x_0 = 0,\ y_0 = 0,\ v_{x0} = 0,\ v_{y0} = 0,\ \theta_0 = 0, \text{ and } \omega_0 = 0$$

and assume to remain at rest means,

$$v_x = 0 \text{ and } v_y = 0\ \omega = 0 \text{ always}$$

From eq (10) ,

$$v_{y.n+1} = v_{y.n} + \Delta t[\tfrac{(u_1+u_2)\cos\theta_n}{m} - g] = 0$$

$$u_1 + u_2 = mg \tag{13}$$

From eq (11),

$$\omega_{n+1} = \omega_n + \Delta t[\frac{r.(u_1+u_2)}{I}]$$

$$u_1 = u_2 \tag{14}$$

From eq (13) and (14) we can write u_1^* and u_2^* to remain at rest is,

$$u_1^* = \frac{mg}{2} \text{ and } u_2^* = \frac{mg}{2}$$

1.3 System Dynamics at $\theta = 0$

To check if the motion is possible at $\theta = 0$.

From equation (9) at $\theta = 0$, the each state in x remain same

$$v_{x.n+1} = v_x = 0$$

$$x_{n+1} = x_n = x_0$$

Thus it is not possible to move in the 'x' direction

1.4 System Dynamics at $\theta = \pi/2$

To check if the system remains at rest at $\theta = \pi/2$,

From eq (9) at $\theta = \pi/2$,

$$v_{x.n+1} = v_{xn} - \frac{\Delta t(u_1+u_2)}{m}$$

and from eq (10) at $\theta = \pi/2$

$$v_{y.n+1} = v_{yn} - \Delta t.g$$

From the above two equations, it is clear that it is impossible to keep velocities at zero which is the condition for the rest position. Thus it is impossible to keep the system at rest.

2 Part 2 - LQR to stay in place

Now we have u^* to keep the robot at rest, we will design a simple controller which keeps the robot in place during a random disturbance. Design an LQR controller for a predefined position.

2.1 Linearize the dynamics:

Linearization of the system at z^* and u^*
where $\bar{z_n} = z_n - z_{des}$ and $\bar{u_n} = u_n - u_{des}$

Linearization will convert non-linear function $z_{n+1} = f(z_n, u_n)$ into $\bar{z_{n+1}} = A.\bar{z_n} + B.\bar{u_n}$

where, A and B are the Jacobian of system dynamics w.r.t. z and u.

$$A = [\frac{\partial f}{\partial z}]_{z^*,u^*} = \begin{bmatrix} 1 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \frac{-\Delta t(u_1+u_2)\cos\theta}{m} & 0 \\ 0 & 0 & 1 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 1 & \frac{-\Delta t(u_1+u_2)\sin\theta}{m} & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$B = [\frac{\partial f}{\partial u}]_{z^*,u^*} = \begin{bmatrix} 0 & 0 \\ (\frac{-\Delta t.\sin\theta}{m}) & (\frac{-\Delta t.\sin\theta}{m}) \\ 0 & 0 \\ (\frac{\Delta t.\cos\theta}{m}) & (\frac{\Delta t.\cos\theta}{m}) \\ 0 & 0 \\ (\frac{\Delta t.r}{I}) & (\frac{-\Delta t.r}{I}) \end{bmatrix}$$

$$\begin{bmatrix} x_{n+1} \\ v_{x.n+1} \\ y_{n+1} \\ v_{y.n+1} \\ \theta_{n+1} \\ \omega_{n+1} \end{bmatrix} = \begin{bmatrix} x_n + \Delta t.v_{x.n} \\ v_{x.n} + \Delta t.\frac{-(u_1+u_2)\sin\theta_n}{m} \\ y_n + \Delta t.v_{y.n} \\ v_{y.n} + \Delta t.[\frac{(u_1+u_2)\cos\theta_n}{m} - g] \\ \theta_n + \Delta t.\omega_n \\ \omega_n + \Delta t.\frac{r(u_1+u_2)}{I} \end{bmatrix}$$

2.2 get_linearization(z, u) function:

Please refer to Part 2 in the python file.

2.3 Infinite horizon LQR controller:

Using linearized system dynamics, designing an infinite horizon LOR controller which stabilizes the drone at the resting point.

The equation of control input is $u = K.\bar{z}$

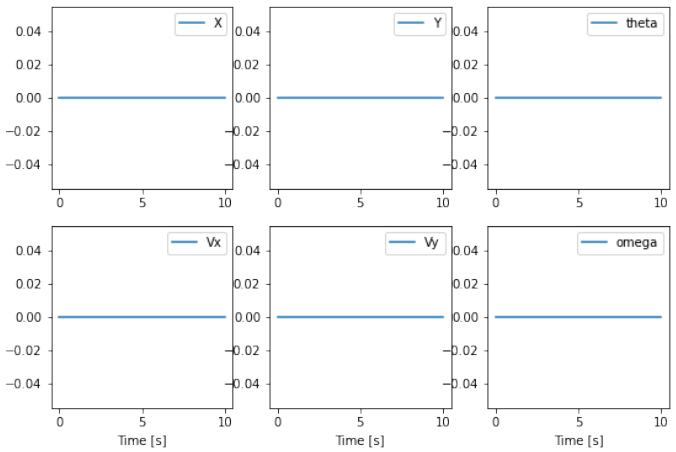
where, $K_n = -(B^T.P_n.B + R)^{-1}.B^T.P_n.A$
and, $P_n = Q + A^T.P_{n+1}.A + A^T.P_{n+1}.B.K$

The value of A and B get from the get_linearization() function. The value of Q and R is constant throughout the procedure.

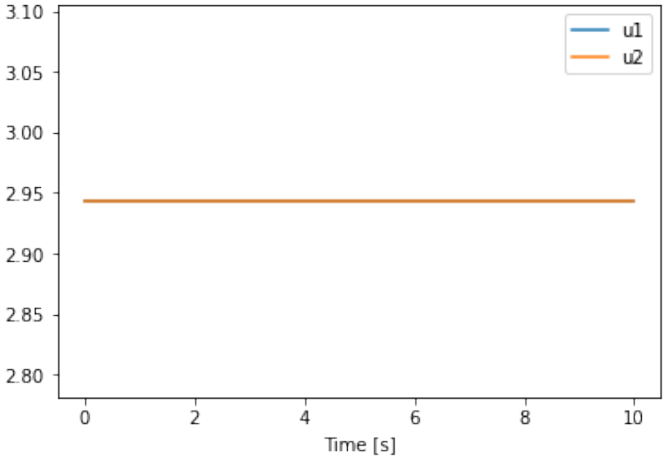
2.4 Design an infinite-horizon LQR controller at z = 0:

The implementation of a controller in the simulator is presented in Part 2 of the python code.

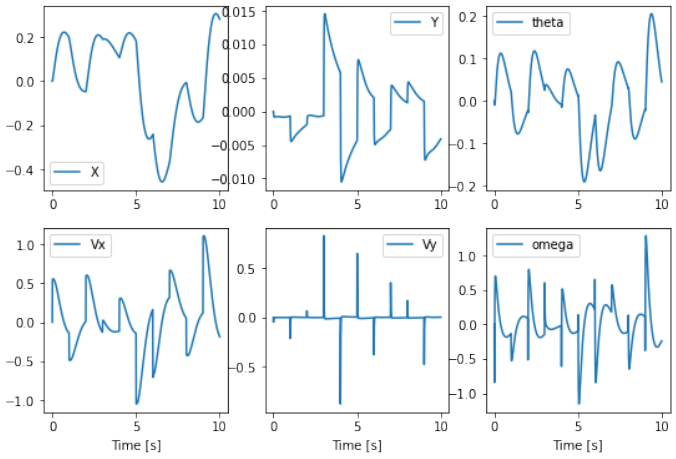
2.5 Result:



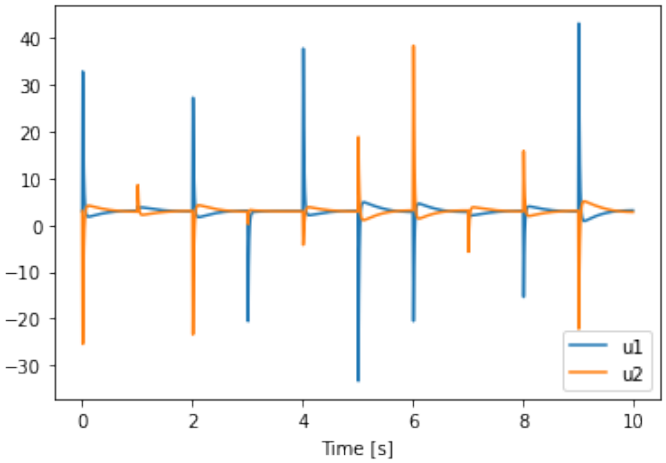
State parameters Disturbance = False



Control parameters Disturbance: False



State parameters Disturbance = True



Control parameters Disturbance: True

3 Part 3 - following a trajectory using linearized dynamics:

In this part, we will follow a desired circular trajectory of radius 1 centered at (0,0) with an orientation of $\theta = 0$

3.1 Linearization of the dynamics:

The equation of linearization which gives values of A and B remains the same as in section 2. The only difference is that we need to calculate it at each stage of the trajectory point. The value of A and B are dependent on value of θ which is constant in this case. Thus for part 3, value of A and B is the same.

3.2 Design a tracking controller:

The design of the controller is similar to the part 2 controller. The main difference is about z_{des} which will be the points of circular trajectory defined by the following equations,

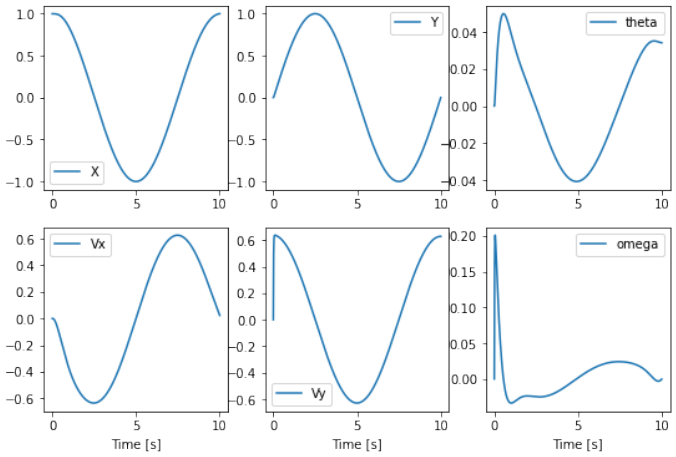
$$z_{des} = \begin{bmatrix} \cos\frac{2.\pi.n}{1000} \\ -0.628\sin\frac{2.\pi.n}{1000} \\ \sin\frac{2.\pi.n}{1000} \\ 0.628\cos\frac{2.\pi.n}{1000} \\ 0 \\ 0 \end{bmatrix}_{6\times N}$$

Then initiate the value of $u = mg/2$. After that compute the value of A and B which will remain the same over the horizon as explained in section 3.1. Based on these values compute K_t, k_t using the backward Recatti equation. Based on this compute u for the circular trajectory controller.

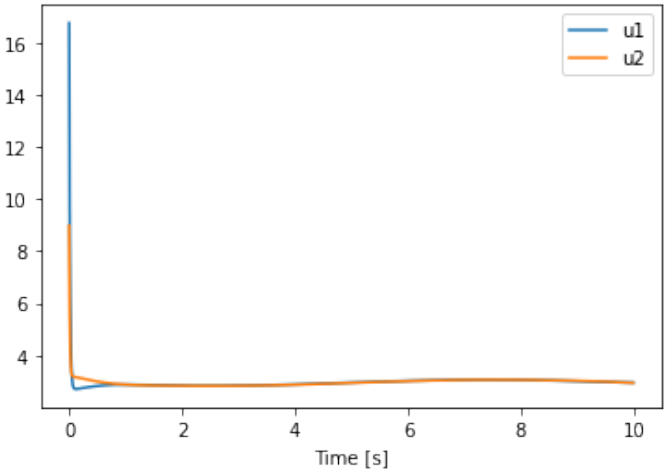
3.3 Tracking controller Python:

Implementation of the tracking controller with simulation is done in part 3 of the python code. The tracking of $\theta = 0$ at each point is impossible. To move the robot in the x and y directions we need some orientation in that direction. As we can see in the result the value of θ changes a bit to get velocity.

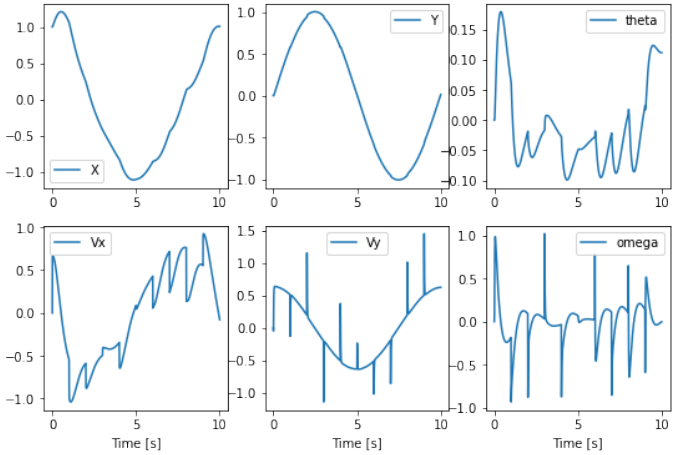
3.4 Result:



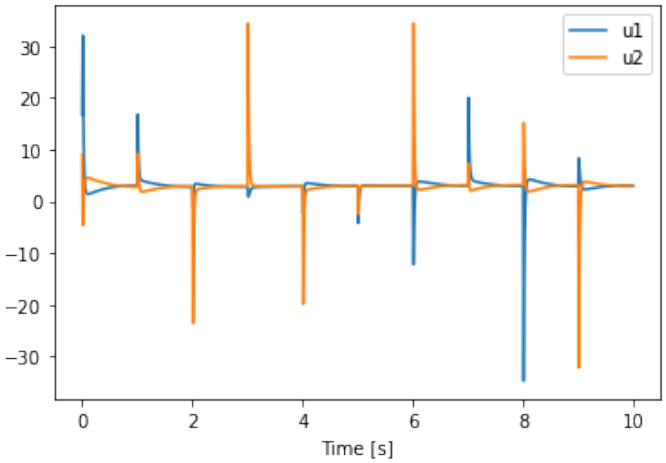
State parameters Circular Trajectory Disturbance = False



Control parameters Circular Trajectory Disturbance: False



State parameters Circular Trajectory Disturbance = True



Control parameters Circular Trajectory Disturbance: True

3.5 Tracking at $\theta = \frac{\pi}{4}$:

It is not possible to keep $\theta = \pi/4$ at each location. A robot is still following a circular trajectory but not always with $\theta = \pi/4$. From start to end it is following a trajectory and at the end, it tries to reach pi/4 orientation. In each quadrant to change direction value of θ needs to change and thus it can't be remain same.

4 Part 4 - iterative LQR:

4.1 Task 1: Reaching a vertical orientation

In this task, the robot should reach a vertical orientation $\theta = \frac{\pi}{2}$ at the location $x = 3$ and $y = 3$ at time $t = 5$ sec. During the rest of the motion, the robot should remain stay close to the origin.

4.1.1 Cost function:

The cost function computes the total cost of trajectory z with control trajectory u. It consists of two parts, i.e. cost to go and terminal cost. This cost function will be used in the next steps to compute Q, R, q, r using Hessian and Jacobian.

The equation of the quadratic cost function is:

$$J = \bar{z_N}^T.Q.\bar{z_N} + \sum_{i=0}^{N-1} \bar{z_i}^T.Q.\bar{z_i} + \bar{u_i}^T.R.\bar{u_i}$$

where,

$$\bar{z} = (z_N^* - z_{des})$$
$$\bar{z_i} = (z_i^* - z_{des})$$
$$\bar{u_i} = (u_i^* - u_{des})$$

4.1.2 Cost function Python:

Refer to part 4 (2) for the python implementation of cost_function($z^*, u^*, horizon\_length$)

4.1.3 Quadratic approximation of Cost function:

Quadratic approximation of the Cost function helps to find the value of Q, q, R, and r.

Q and R are the hessian of cost function J while
q and r are the jacobians of cost function J.

Where,

$$Q = \frac{\partial^2 J}{\partial z^2} = 2\bar{Q}$$
$$R = \frac{\partial^2 J}{\partial u^2} = 2\bar{R}$$
$$q = \frac{\partial J}{\partial z} = 2\bar{Q}\bar{Z} \text{ where } \bar{Z} = (z^* - z_{des})$$
$$r = \frac{\partial J}{\partial u} = 2\bar{R}\bar{U} \text{ where } \bar{U} = (u^* - u_{des})$$

More explanation in section 4.1.5

4.1.4 Quadratic approximation of Cost function Python:

Refer to part 4 (2) for the python implementation of

get_quadratic_approximation_cost($z^*, u^*, horizon\_length$)

4.1.5 Implementation of iLQR:

1. Initialize u^* and z_0^*:
   $u^* = \begin{bmatrix} 0 & 0 & .... & 0 \\ 0 & 0 & .... & 0 \end{bmatrix}_{2\times N}$ and $z_0^* = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{6\times 1^T}$
2. Define the desired z_{des} The timeline of implementation is 10 sec which is divided into 1000 steps. Based on the desired position it is divided into 3 steps.
   0^{th} to 400^{th}: $z_{des} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$
   400^{th} to 500^{th}: $z_{des} = \begin{bmatrix} 3 & 0 & 3 & 0 & \frac{\pi}{2} & 0 \end{bmatrix}^T$
   500^{th} to N^{th}: $z_{des} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$
3. Compute z^*: Using the get_next_state() function find the next states in z^* of size $6 \times N + 1$

$$\begin{bmatrix} x_{n+1} \\ v_{x.n+1} \\ y_{n+1} \\ v_{y.n+1} \\ \theta_{n+1} \\ \omega_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ v_{x.n} \\ y_n \\ v_{y.n} \\ \theta_n \\ \omega_n \end{bmatrix} + \begin{bmatrix} v_{x.n} \\ \frac{-(u_1+u_2)\sin\theta_n}{m} \\ v_{y.n} \\ \frac{(u_1+u_2)\cos\theta_n}{m} - g \\ \omega_n \\ \frac{r(u_1+u_2)}{I} \end{bmatrix} .\Delta t$$

4. Compute J_{old} cost using compute_cost() which is described in section 4.1.1 and 4.1.2.

5. Compute Q, q, R, r using get_quadratic_approximation_cost() function.

   Q and R are the hessian of cost function J while
   q and r are the jacobians of cost function J.

   Where,

$$Q = \frac{\partial^2 J}{\partial z^2} = 2\bar{Q} = \begin{bmatrix} Q_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & Q_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & Q_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & Q_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & Q_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & Q_6 \end{bmatrix}_{6\times 6}$$

   The $\begin{bmatrix} Q_1 & Q_2 & Q_3 & Q_4 & Q_5 & Q_6 \end{bmatrix}^T$ is a diagonal gain value that changes as per steps. Each value gives gain to different state parameters and as per need changes.

   0^{th} to 400^{th}: $Q_i = \begin{bmatrix} 5 & 2 & 5 & 2 & 2 & 2 \end{bmatrix}$
   More gain to position x and y to remain at rest.

   400^{th} to 500^{th}: $Q_i = \begin{bmatrix} 100 & 2 & 100 & 2 & 200 & 2 \end{bmatrix}$
   More gain to orientation to achieve $\theta = \frac{\pi}{2}$.

   500^{th} to 600^{th}: $Q_i = \begin{bmatrix} 200 & 2 & 200 & 2 & 100 & 2 \end{bmatrix}$
   More gain to position and orientation to get to the stable position.

   600^{th} to N^{th}: $Q_i = \begin{bmatrix} 5 & 2 & 5 & 2 & 2 & 2 \end{bmatrix}$
   More gain to position x and y to come back at rest.

$$R = \frac{\partial^2 J}{\partial u^2} = 2\bar{R} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_{2\times 2}$$

$$q = \frac{\partial J}{\partial z} = 2\bar{Q}\bar{Z} = \begin{bmatrix} Q_1.x^* - Q_1.x_{des} \\ Q_2.v_x^* \\ Q_3.y^* - Q_3.y_{des} \\ Q_4.v_y^* \\ Q - 5.\theta^* - Q_5.\theta_{des} \\ Q_6.\omega^* \end{bmatrix}_{6\times 1}$$

$$r = \frac{\partial J}{\partial u} = 2\bar{R}\bar{U} = \begin{bmatrix} 2R_1.u_1^* - 2R_1.m.g \\ 2R_2.u_2^* - 2R_2.m.g \end{bmatrix}_{2\times 1}$$

In the end we will get $Q_{N+1\times 6\times 6}$, $R_{N\times 2\times 2}$, $q_{N+1\times 6\times 1}$ and $r_{N\times 6\times 1}$

6. Linearize the system dynamics using the equation from section 2.1 and python code from section 2.2

7. Compute z_{new}^* and u_{new}^*
   Initiate the $z_{new.0}^* = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{6\times 1}^T$
   Compute the u_{new}^* in iterative way by

$$u_{new}^* = u^* + K_t(z_{new}^* - z^*) + \alpha.k_t$$

$$z_{new(i)}^* = \text{get\_new\_state}(u_{new(i)}^*, z_{new(i)}^*)$$

   Where, α = linear search parameter

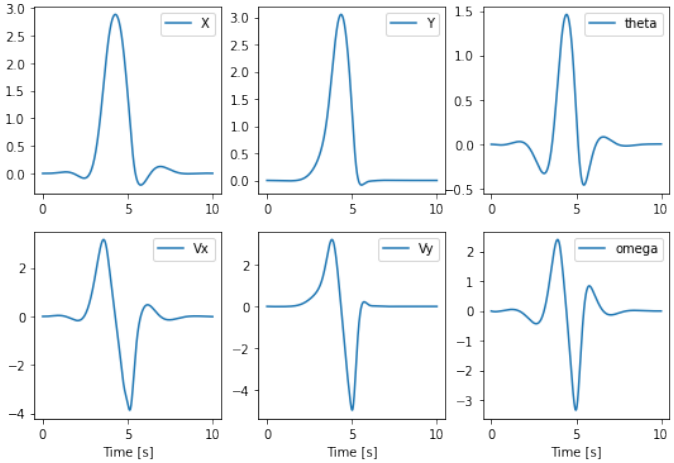
8. Linear search algorithm:
   Compute the new cost J_{new} using z_{new}^* and u_{new}^* using compute_cost() function. Initially $\alpha = 1$. If cost is decreasing then make $J_{old} = J_{new}$ and if it is not decreasing then make it half.

9. Update the value of u^* and iterate till it gets converges.

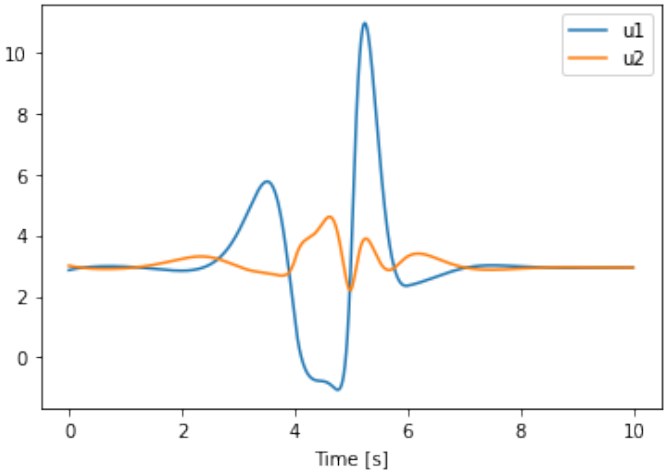
$$u^* = u_{new}^*$$

   Iterate the procedure until the cost is less than the specified tolerance value. Mostly it gets converges within 100 iterations. Iterate from step (3) until convergence.

4.1.6 Results:



State parameters vertical orientation



Control parameters vertical orientation

4.1.7 Conclusion:

This method is computationally heavy and needs to be done before an actual run of the robot and can't be used for online implementation. Also, to track a point we need to provide some approximation as a desired one and need to tune Q and R gains manually to achieve that position.

When we know the equations of dynamics, the implementation becomes easy. Also, to track desired points we don't need to provide a complete trajectory. It will automatically compute that. Also, if at any time if the system goes away from its desired point, then instead of coming back to it, this method will compute a new one from the current position.

4.2 Task 2 - doing a full flip:

The procedure for task 2 is similar to task 1. The only change is regarding the value of z_{des} at each stage and corresponding values of Q_i at the diagonal of gain. These values are:

0^{th} to 400^{th}:

$$Q_i = \begin{bmatrix} 5 & 2 & 5 & 2 & 2 & 2 \end{bmatrix}$$

$$z_{des} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

400^{th} to 500^{th}:

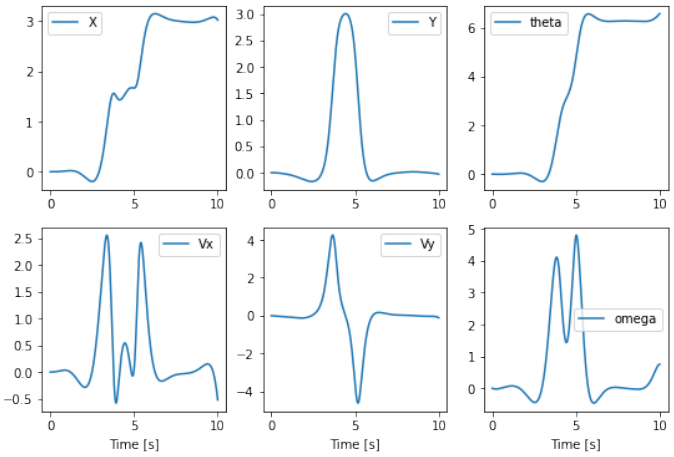
$$Q_i = \begin{bmatrix} 100 & 2 & 100 & 2 & 200 & 2 \end{bmatrix}$$

$$z_{des} = \begin{bmatrix} 1.5 & 0 & 3 & 0 & \pi & 0 \end{bmatrix}^T$$

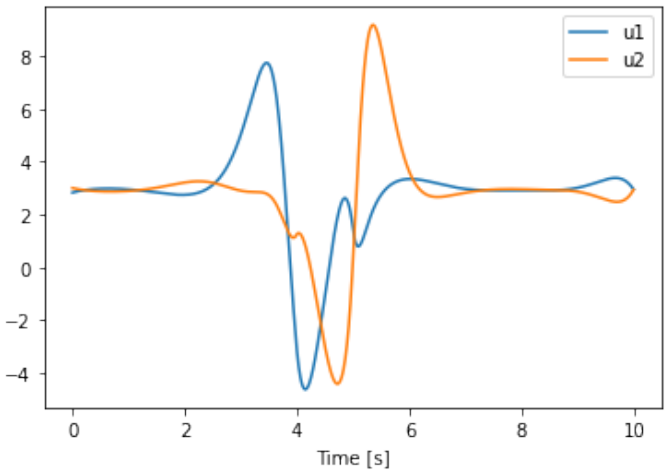
500^{th} to N^{th}:

$$Q_i = \begin{bmatrix} 20 & 2 & 20 & 2 & 100 & 2 \end{bmatrix}$$

$$z_{des} = \begin{bmatrix} 3 & 0 & 0 & 0 & 2\pi & 0 \end{bmatrix}^T$$



State parameters Full flip



Control parameters Full flip