2.1)

a) At equilibria, time derivatives of state equal to zero.

$$\begin{bmatrix}\dot{y}(t)\\ \dot{h}(t)\end{bmatrix} = \begin{bmatrix}v(t)sin(h(t))\\ w(t)\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

$$v(t)sin(h(t)) = 0 \quad (1)$$
$$w(t) = 0 \quad (2)$$

From (1) and (2),

$sin(h(t)) = 0$ which implies: $h(t) = \{0,\pi,2\pi\}, w(t) = 0$

The equilibria are:

$$x_{eq} = \begin{bmatrix}y\\0\end{bmatrix}, \begin{bmatrix}y\\\pi\end{bmatrix}, \begin{bmatrix}y\\2\pi\end{bmatrix} \quad \forall y \in \mathbb{R}$$

Linearising the system around $x_{eq} = \begin{bmatrix}y\\0\end{bmatrix}$,

Linearised matrices $A_{lin} = \nabla_x f(x_{eq}, u_{eq})$, $B_{lin} = \nabla_u f(x_{eq}, u_{eq})$
The linear dynamic equation is of the form:

$$\dot{x} = f(x_{eq}, u_{eq}) + A_{lin}\delta x + B_{lin}\delta u \quad (3)$$

$$\delta x = x - x_{eq} \quad (4)$$
$$\delta u = u - u_{eq}$$

$$f(x_{eq}, u_{eq}) = 0 \quad (5)$$

$$\dot{x} = A_{lin}\delta x + B_{lin}\delta u$$

$$A_{lin} = \nabla_x f(x_{eq}, u_{eq}) = \begin{bmatrix}0 & v\\0 & 0\end{bmatrix} \quad (6)$$

$$B_{lin} = \nabla_u f(x_{eq}, u_{eq}) = \begin{bmatrix}0\\1\end{bmatrix} \quad (7)$$

b) $Q_t = Q_s = diag(1,1) \quad R_s = 20$ (1)

$$u(t) = u_0 + K(x - x_{eq}) \\ = u_{ff} + u_{fb} \quad (2)$$

$$u_{ff} = -Kx_{eq} \quad (3)$$

$$u_{fb} = Kx \quad (4)$$

$$u(t) = \begin{bmatrix}\theta_{ff} & \theta_{fb}\end{bmatrix}\begin{bmatrix}1 \\ x(t)\end{bmatrix} \quad (5)$$

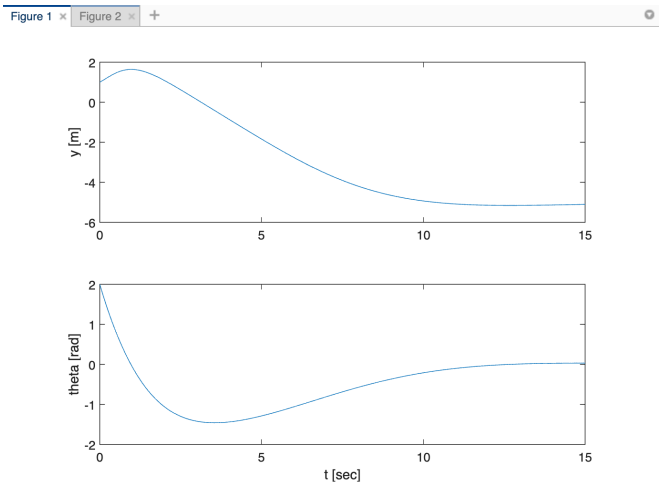
$$\theta_{ff} = -Kx_{eq} = \text{-1.118} \quad (6)$$

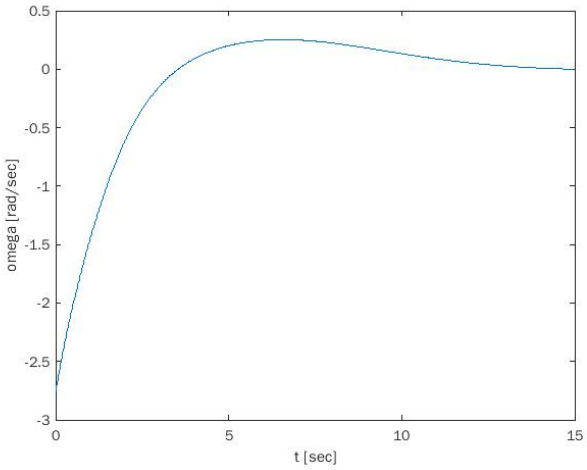
$$\theta_{fb} = -K = [-0.2236 \quad -0.7051] \quad (7)$$

c) Code In Appendix:

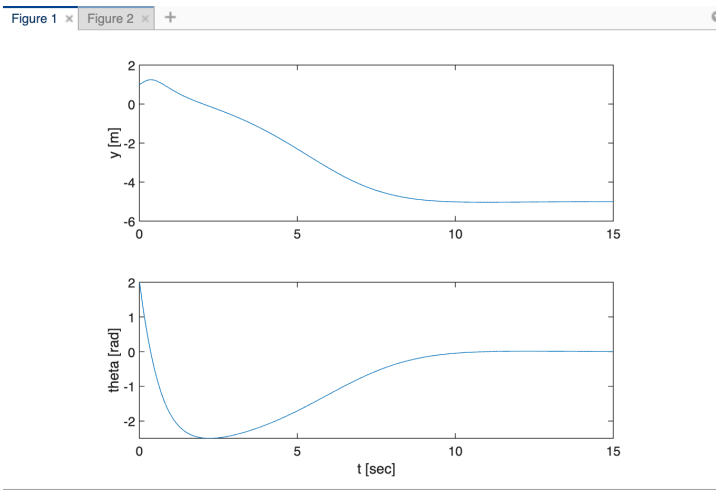
1.
Vary Q_s

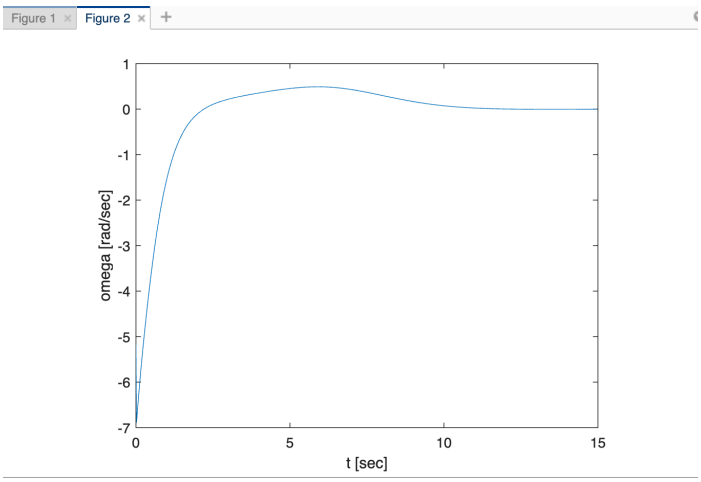
Q = diag(1,1), R_s = 20





Q = diag(10,10), R_s = 20

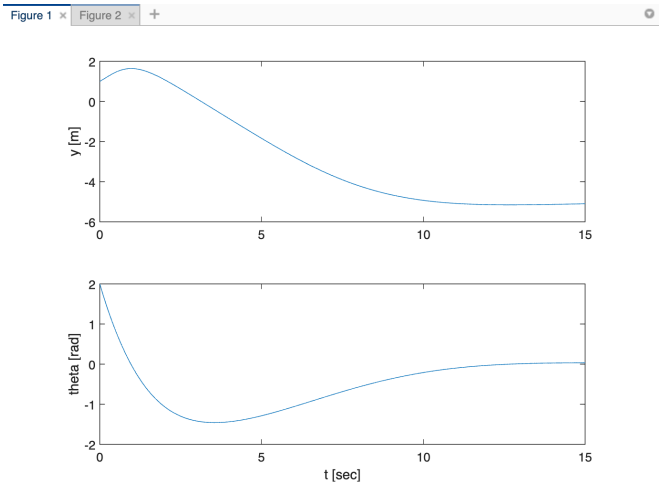


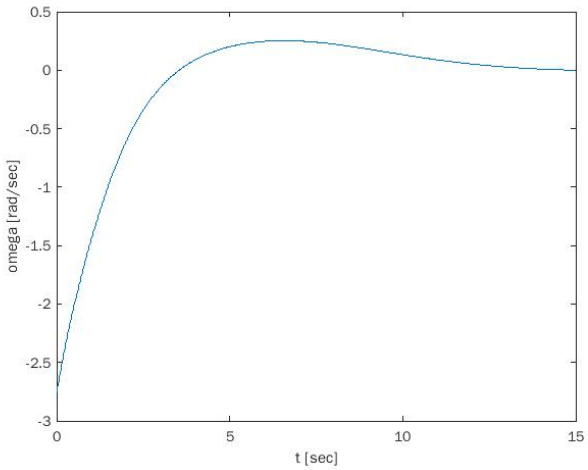


When Q is increased from diag(1,1), to diag(10,10), the robot moves to goal position aggressively as the penalty for error in state is increased. In this case, the control input value also increases, when compared to the previous case. By increasing Q_s, the system can be forced to reach goal state quickly, at the cost of increasing actuation inputs.

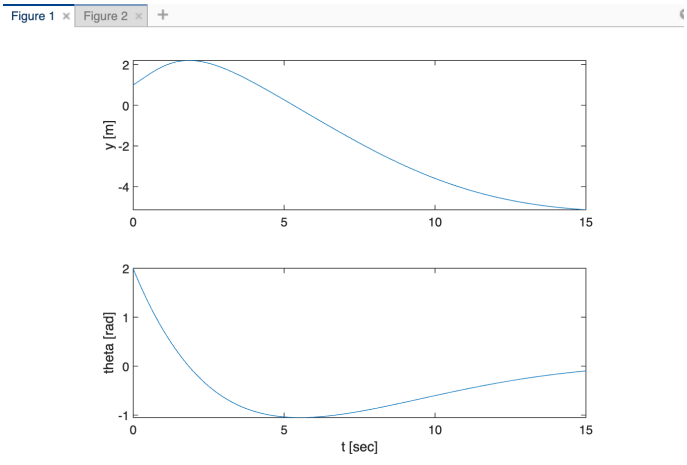
Vary R_s

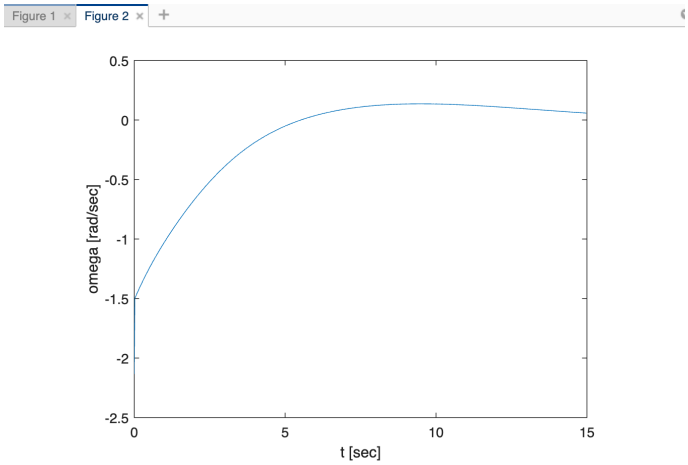
Q = diag(1,1), R_s = 20





Q = diag(1,1), R_s = 100



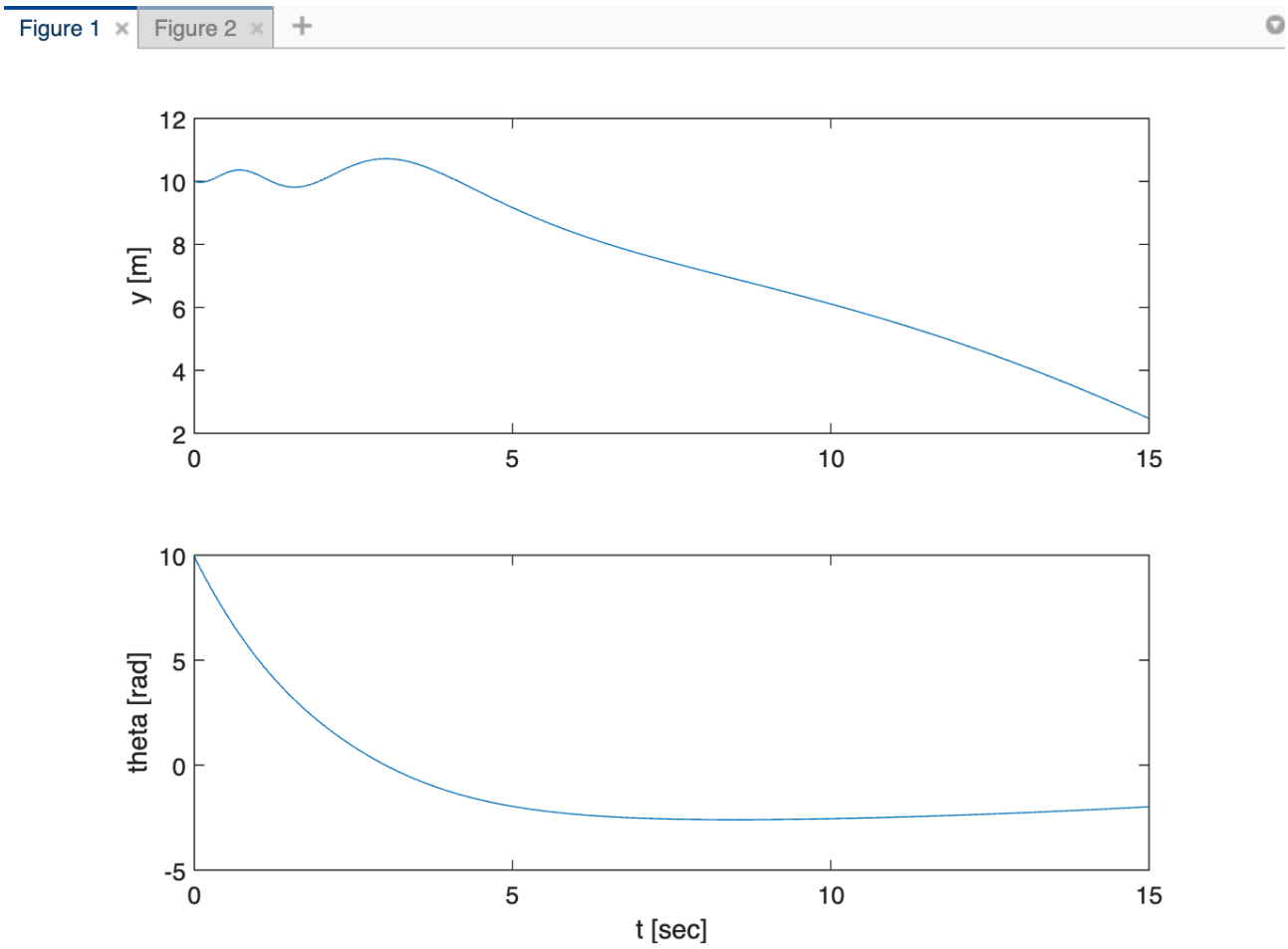


When R_s is increased from 20 to 100, the system takes more time to reach the goal state, as higher actuation input is penalised more. For R_s = 100, magnitude maximum actuation input = 1.5, while magnitude maximum actuation input = 2.8 when R_s is 20.

2.

If x_0 is changed, the performance of the LQR controller decreases. As the Linear dynamics model is only valid around a small region of state space around the linearising point, the dynamic model would be incorrect and LQR feedback policy would not drive the system towards goal state.

See figure below, for $x_0 = [10,10]$



The robot does not reach x_{goal}

d) See Code in Appendix

Linearising the system around $\bar{x_k} = \begin{bmatrix} \bar{y_k} \\ \bar{h_k} \end{bmatrix}$

Linearised matrices for continuous time system: $A_{lin} = \nabla_x f(x_{eq}, u_{eq})$,
$B_{lin} = \nabla_u f(x_{eq}, u_{eq})$

$$\delta x_k = x_k - \bar{x_k} \quad (4)$$
$$\delta u_k = u_k - \bar{u_k}$$

The linear dynamic equation is of the form:

$$\bar{x}_{k+1} + \delta x_{k+1} = f(\bar{x_k}, \bar{u_k}) + A_k \delta x + B_k \delta u$$

$$\delta x_{k+1} = A_k \delta x + B_k \delta u \quad (3)$$

$$A_k = I + \delta t A_{lin} = \begin{bmatrix} 1 & v\delta t \cos(\bar{h_k}) \\ 0 & 1 \end{bmatrix}$$

$$B_k = B_{lin}\delta t = \begin{bmatrix} 0 \\ \delta t \end{bmatrix}$$

e) See Code in Appendix

$$J = g_N(x_N) \; + \; \delta t \sum_{k=1}^{N-1} g_k(x_k, u_k) \tag{1}$$

Where

$$g_N(x_N) = \frac{1}{2}(x_N - x_{goal})^T Q_t (x_N - x_{goal}) \tag{2}$$

$$g_k(x_k, u_k) = \frac{1}{2}(x_k - x_{goal})^T Q_t (x_k - x_{goal}) + \frac{1}{2} u_k^T R_s u_k \tag{3}$$

Substituting (2), (3) in (1),

$$J = \frac{1}{2}(x_N - x_{goal})^T Q_t (x_N - x_{goal}) + \; \delta t \sum_{k=1}^{N-1} \frac{1}{2}(x_k - x_{goal})^T Q_s (x_k - x_{goal}) + \frac{1}{2} u_k^T R_s u_k$$

Second Order Approximations of cost functions:

$$g_N(x_N) \approx q_N + \delta x_N^T \mathrm{q}_N + \frac{1}{2} \delta x_N^T Q_N \delta x_N \tag{4}$$

$$g_k(x_k, u_k) \approx q_k + \delta x_k^T \mathrm{q}_k + \delta u_k^T \mathrm{r}_k + \frac{1}{2} \delta x_k^T Q_k \delta x_k + \frac{1}{2} \delta u_k^T R_k \delta u_k + \delta u_k^T P_k \delta x_k \tag{5}$$

Where,

$$q_N = g_N(\bar{x}_N) = \frac{1}{2}(\bar{x}_N - x_{goal})^T Q_t (\bar{x}_N - x_{goal})$$

$$\mathrm{q}_N = \nabla_{x_N} g_N(\bar{x}_N) = Q_t(\bar{x}_N - x_{goal})$$

$$\mathrm{Q}_N = \nabla^2_{x_N} g_N(\bar{x}_N) = Q_t$$

And stage cost terms:

$$q_k = g_k(\bar{x}_k, \bar{u}_k) = \frac{1}{2}(\bar{x}_k - x_{goal})^T Q_t (\bar{x}_k - x_{goal}) + \bar{u}_k^T R_s \bar{u}_k$$

$$\mathrm{q}_k = \nabla_{x_k} g_k(\bar{x}_k, \bar{u}_k) = Q_k(\bar{x}_k - x_{goal})$$

$$r_k = \nabla_{u_k} g_k(\bar{x}_k, \bar{u}_k) = R_s \bar{u}_k$$

$$\mathrm{Q}_k = \nabla^2_{x_k} g_k(\bar{x}_k, \bar{u}_k) = Q_s$$

$$R_k = \nabla^2_{u_k} g_k(\bar{x}_k, \bar{u}_k) = R_s$$

$$P_k = \nabla_{u_k}(\nabla^T_{x_k} g_k(\bar{x}_k, \bar{u}_k)) = [0,0]$$

f)

$$V^*(k, x_k) = min_{u_k}[g_k(x_k, u_k) + V^*(k+1, x_{k+1})] \quad (1)$$

Where $V^*(k+1, x_{k+1})$ can be approximated as:

$$V^*(k+1, x_{k+1}) = s_{k+1} + \delta x_{k+1}^T s_{k+1} + \frac{1}{2}\delta x_{k+1}^T S_{k+1} \delta x_{k+1} \quad (2)$$

Substituting $V^*(k+1, x_{k+1})$ in (1):

$$V^*(k, x_k) = min_{u_k}[q_k + \delta x_k^T q_k + \delta u_k^T r_k + \frac{1}{2}\delta x_k^T Q_k \delta x_k + \frac{1}{2}\delta u_k^T R_k \delta u_k + \delta u_k^T P_k \delta x_k + s_{k+1} + \delta x_{k+1}^T s_{k+1} + \frac{1}{2}\delta x_{k+1}^T S_{k+1} \delta x_{k+1}]$$

Relabel terms dependent on control:

$$g_k = r_k + B_k^T s_{k+1}$$

$$G_k = P_k + B_k^T s_{k+1} A_k$$

$$H_k = R_k + B_k^T s_{k+1} B_k$$

$$V^*(k, x_k) = min_{u_k}[constant\ terms\ +\ terms\ dependent\ only\ on\ x + \delta u_k^T (g_k + G_k \delta x_k) + \frac{1}{2}\delta u_k^T H_k \delta u_k]$$

Upon taking first derivative = 0,

$$\delta u_k^* = -H_k^{-1} g_k - H_k^{-1} G_k \delta x_k$$

$$u_k = \bar{u}_k + \delta u_k^*$$

Feed forward term,

$$\theta_{k_{ff}} = \bar{u}_k - H_k^{-1} g_k - H_k^{-1} G_k \bar{x}_k$$

Feed back term:

$$\theta_{k_{fb}} = -H_k^{-1} G_k$$

g)

Substitute $\delta u_k^* = -H_k^{-1} g_k - H_k^{-1} G_k \delta x_k$,
$V^*(k+1, x_{k+1}) = s_{k+1} + \delta x_{k+1}^T \mathsf{s}_{k+1} + \frac{1}{2} \delta x_{k+1}^T S_{k+1} \delta x_{k+1}$ in
$V^*(k, x_k) = min_{u_k}[g_k(x_k, u_k) + V^*(k+1, x_{k+1})]$

To obtain:

$$S_k = Q_k + A_k^T S_{k+1} A_k + K_k^T H_k K_k + K_k^T G_k + G_k^T K_k$$

$$\mathsf{s}_k = \mathsf{q}_k + A_k^T \mathsf{s}_{k+1} + K_k^T H_k \delta u_{ff_k} + K_k^T g_k + G_k^T \delta u_{ff_k}$$

$$s_k = q_k + s_{k+1} + \frac{1}{2} \delta u_{ff_k}^T H_k \delta u_{ff_k} + \delta u_{ff_k}^T g_k$$

To initialise S_k, s_k, s_k , start with backward pass:
Set: $J_N^*(x_N) = g_N(x_N)$.

Which implies:

$$s_N + \delta x_N^T \mathsf{s}_N + \frac{1}{2} \delta x_N^T S_N \delta x_N = q_N + \delta x_N^T \mathsf{q}_N + \frac{1}{2} \delta x_N^T Q_N \delta x_N$$

which can be solved to:

$$s_N = q_N$$

$$\mathsf{s}_N = \mathsf{q}_N$$

$$S_N = Q_N$$

h) Code in Appendix

i)

Pseudocode:

Compute Initial policy from LQR $\mu_k^0(x_k)_{k=0}^{N-1}$

Recursion l = {0, 1, 2, …..}

1. Forward Pass: Apply current control policy to the nonlinear system and obtain state and input trajectories . $\{\bar{x_0}^l, \bar{x_1}^l, \ldots . \bar{x_N}^l\}$ & $\{\bar{u_0}^l, \bar{u_1}^l, \ldots . \bar{u_N}^l\}$

2. Compute quadratic approximation of terminal cost and Initialise backward pass

$[q_N, q_N, Q_N] = \text{terminal cost quad}(Q_t, x_{goal}, \bar{x}_N)$

1. $s_N = q_N$
2. $s_N = q_N$
3. $S_N = Q_N$

3. Backward pass, k = {N-1, N-2,……,0}:

3.1) Linearize dynamics around $(\bar{x}_k, \bar{u}_k)$

[A_k, B_k] = mobile_robot_lin $(\bar{x}_k, \bar{u}_k, \delta t, \bar{v})$

3.2 Approximate stage cost with second order Taylor expansion around $(\bar{x}_k, \bar{u}_k)$.

$[q_k, q_k, Q_k, r_k, R_k, P_k] = \text{stage cost quad}(Q_s, R_s, x_{goal}, \delta t, \bar{x}_k, \bar{u}_k)$

3.3 Update policy:

$[\theta_{k,ff}, \theta_{k,fb}, s_k, s_k, S_k] = \text{update policy}(A_k, B_k, q_k, q_k, Q_k, r_k, R_k, P_k, s_{k+1}, s_{k+1}, S_{k+1}, \bar{x}_k, \bar{u}_k)$

3.4 Store New theta

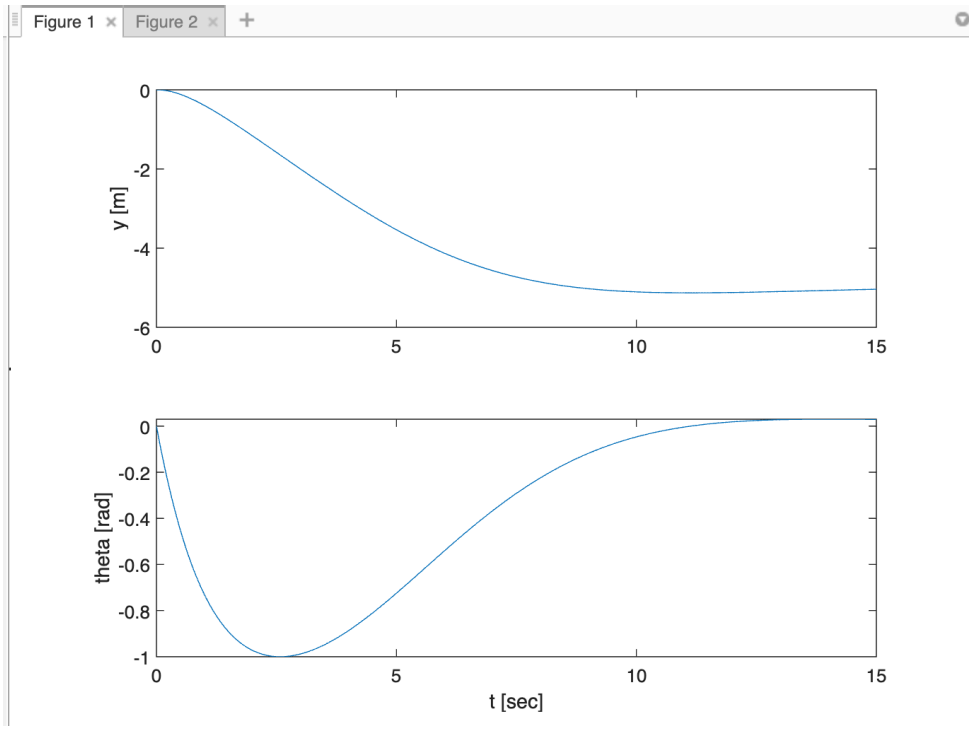
$\theta = [\theta_{k,ff}, \theta_{k,fb}]$

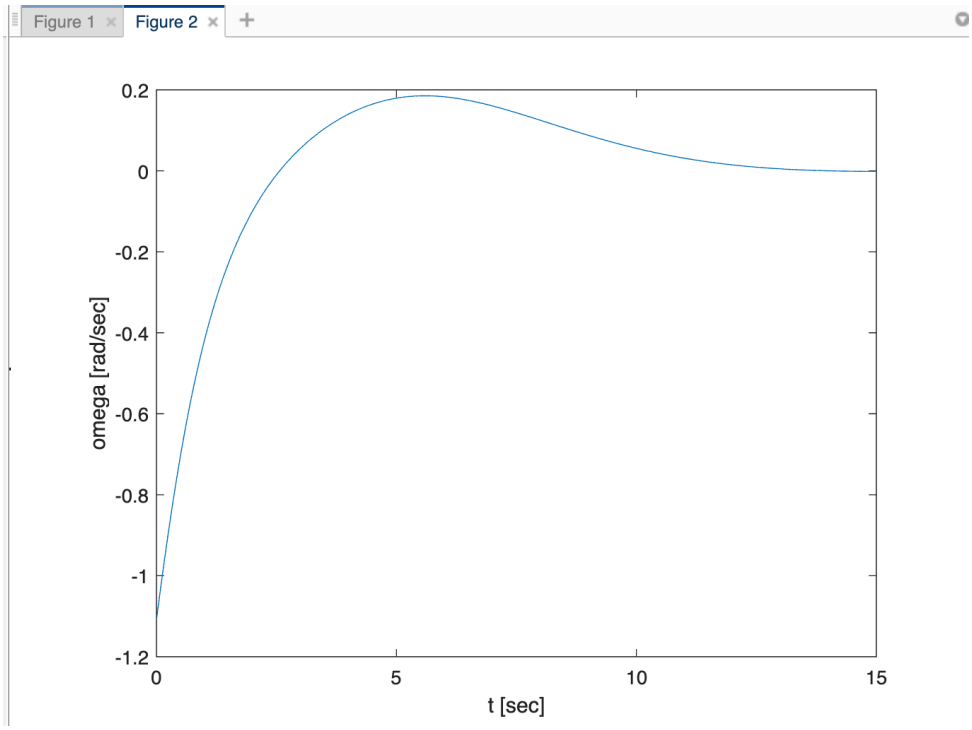
Repeat until termination condition is satisfied and return optimal policy.

j) Code in Appendix

1.

x_0 = [0,0]’





x_0 = [0,pi]’

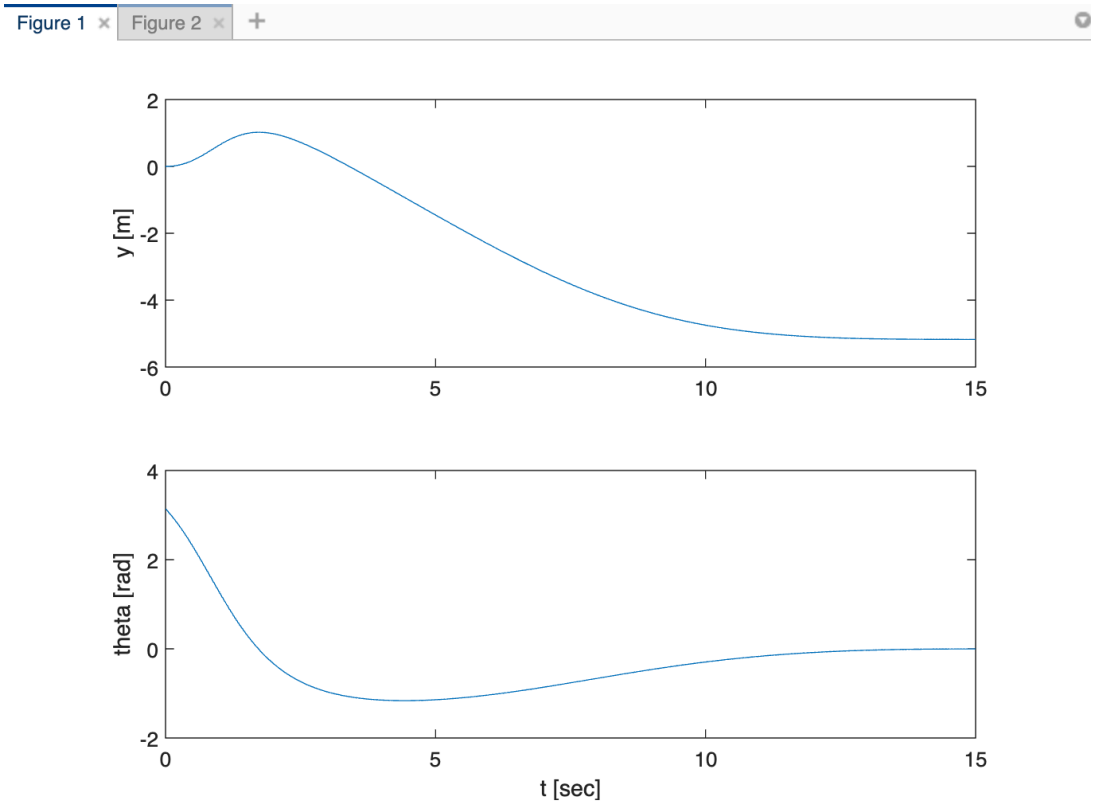


Figure 1
Figure 2
y [m]
theta [rad]
t [sec]

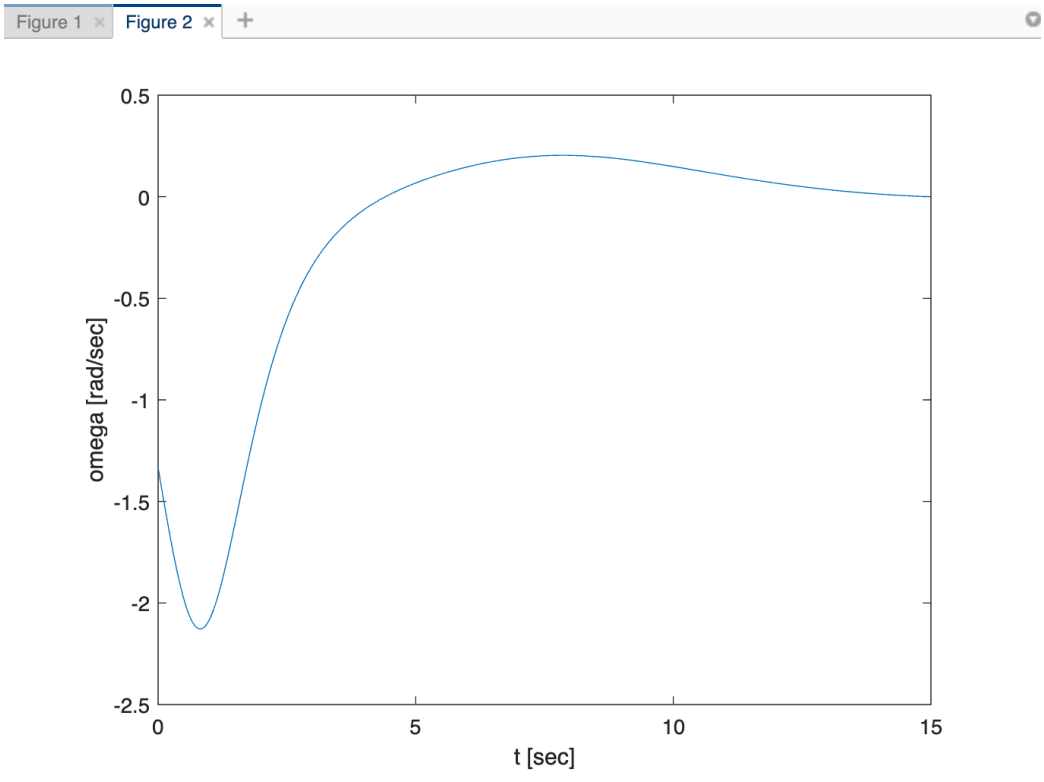
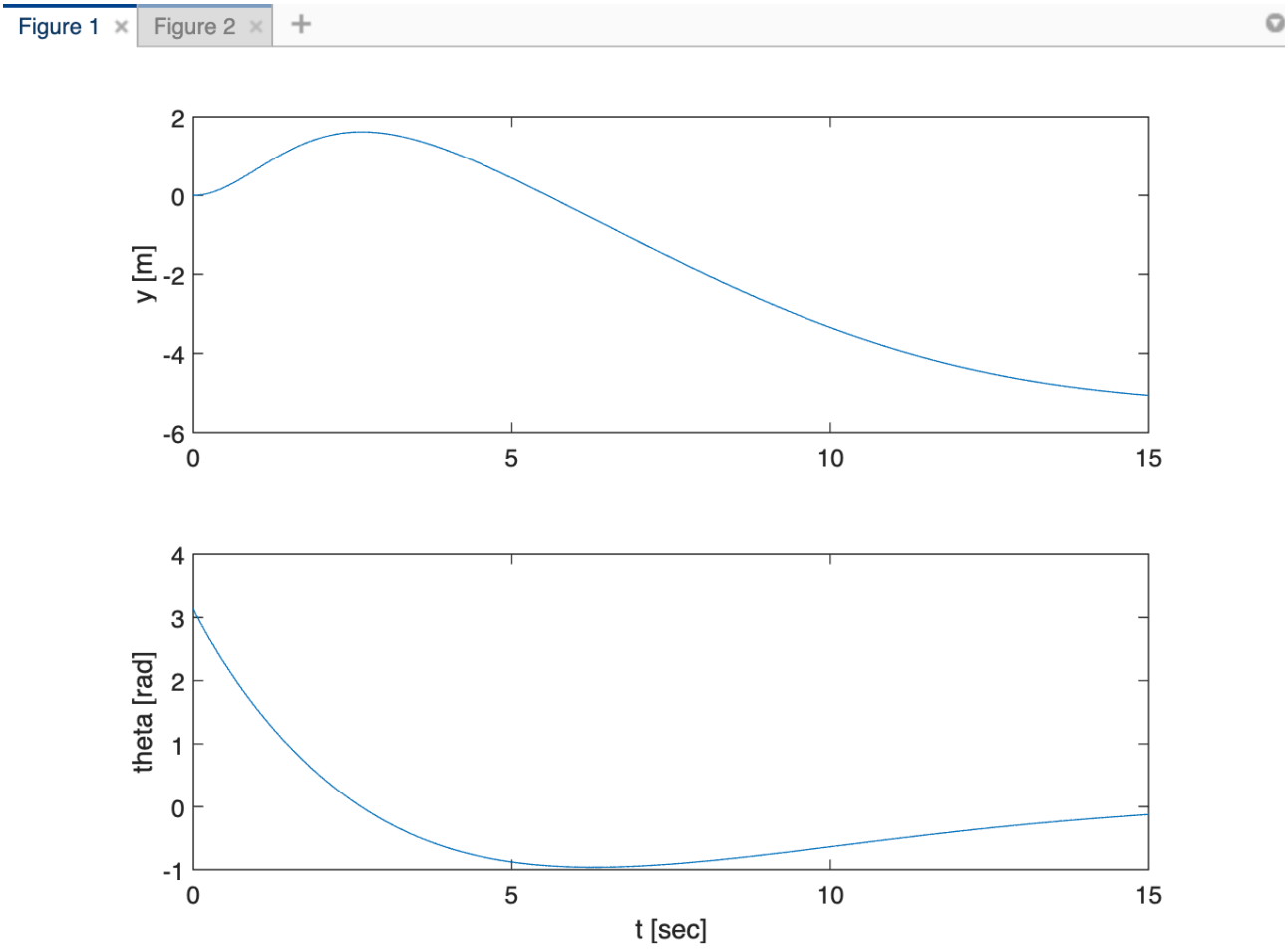


Figure 1
Figure 2
omega [rad/sec]
t [sec]

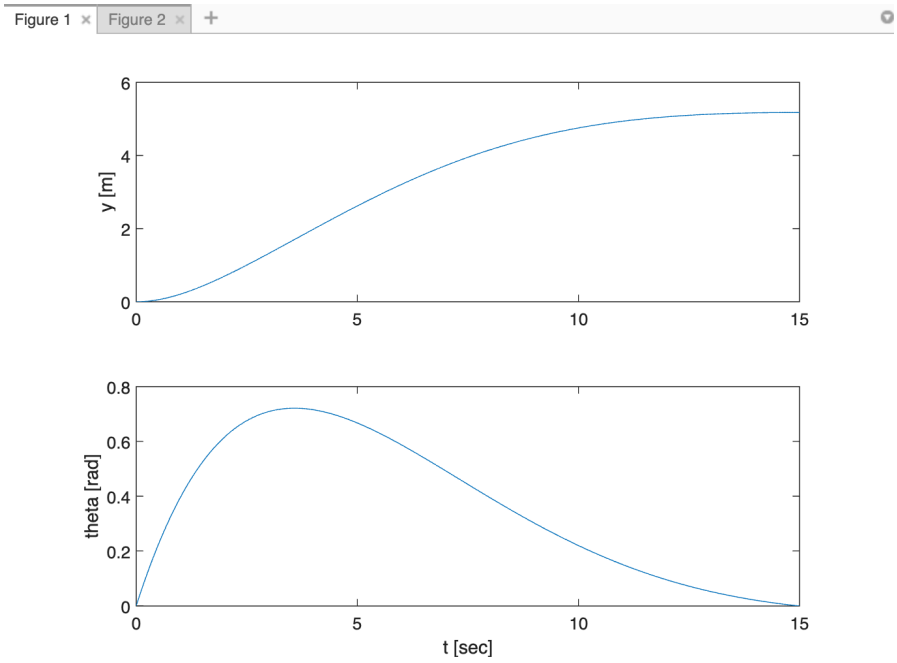
LQR controller was also able to generate stabilising control policies for both cases. However, the LQR controller took more time to reach the goal state. (Refer figure below) (For x_0 = [0 , pi]'



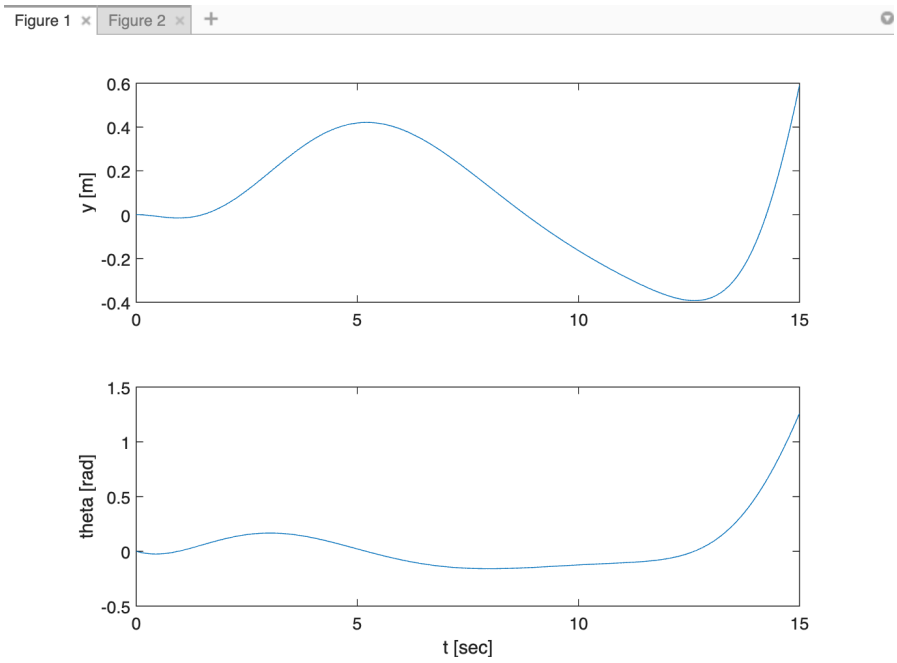
2.

Performance of controllers when there is model mismatch:
(disturbance = 1)

LQR:



iLQC:



Both controllers did not perform well when there is a model mismatch. However iLQC performed slightly better when compared to LQR, in terms of reaching close to the goal. iLQC converged to a lower overall cost than LQR.

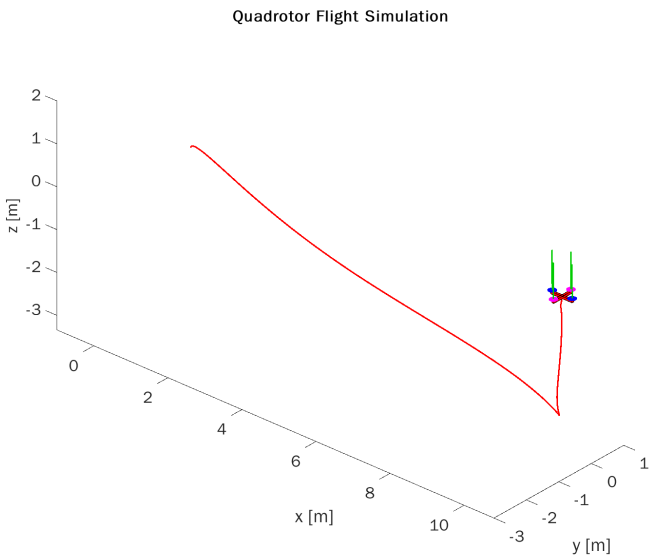
2.2

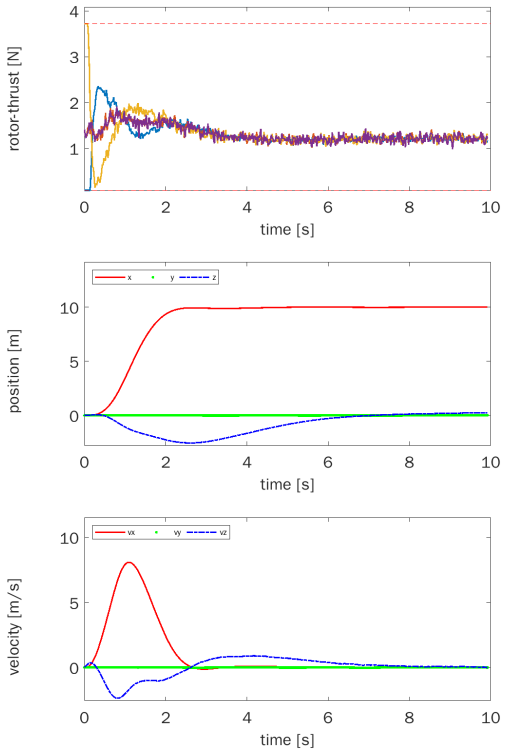
a) Code in Appendix:

b) Code in appendix:

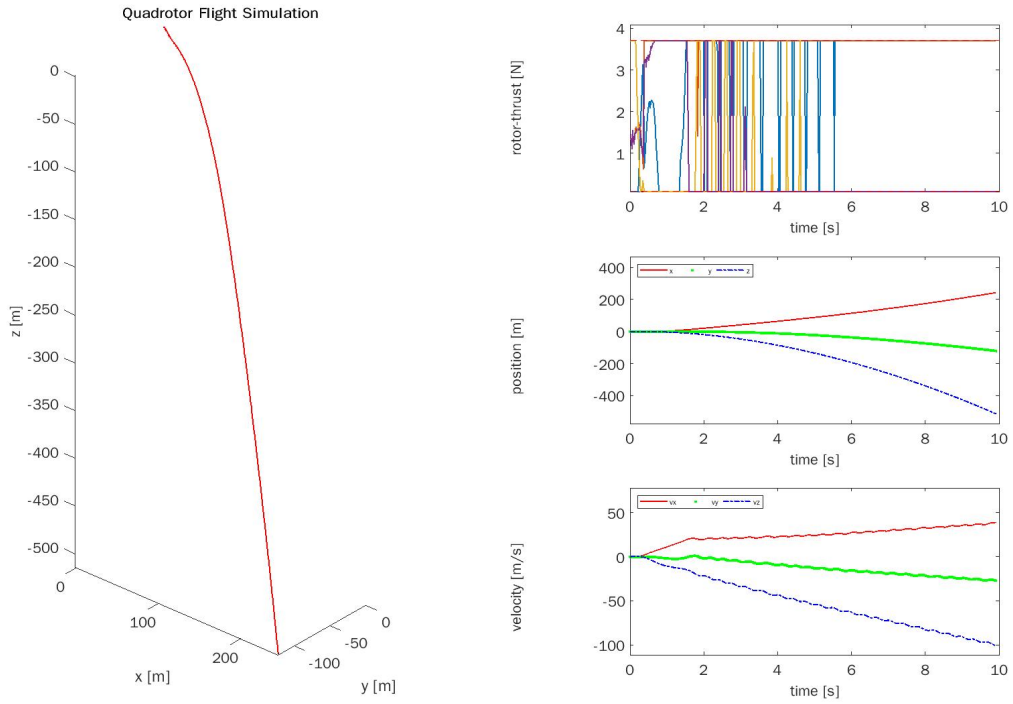
1. Plots:

Plot for $x_{goal} = [10,0,0,0,0,0,0,0,0,0,0,0]^T$ and $t_{goal} = 10$.





Plot for $x_{goal} = [15,0,0,0,0,0,0,0,0,0,0,0]^T$ and $t_{goal} = 10$.



When $x_{goal} = [15,0,0,0,0,0,0,0,0,0,0,0]^T$

The quadrotor does not reach goal state in 10 s. The LQR controller does not converge to an optimal policy. Hence the LQR controller only works well for a bounded region of state space near to the linearisation point.

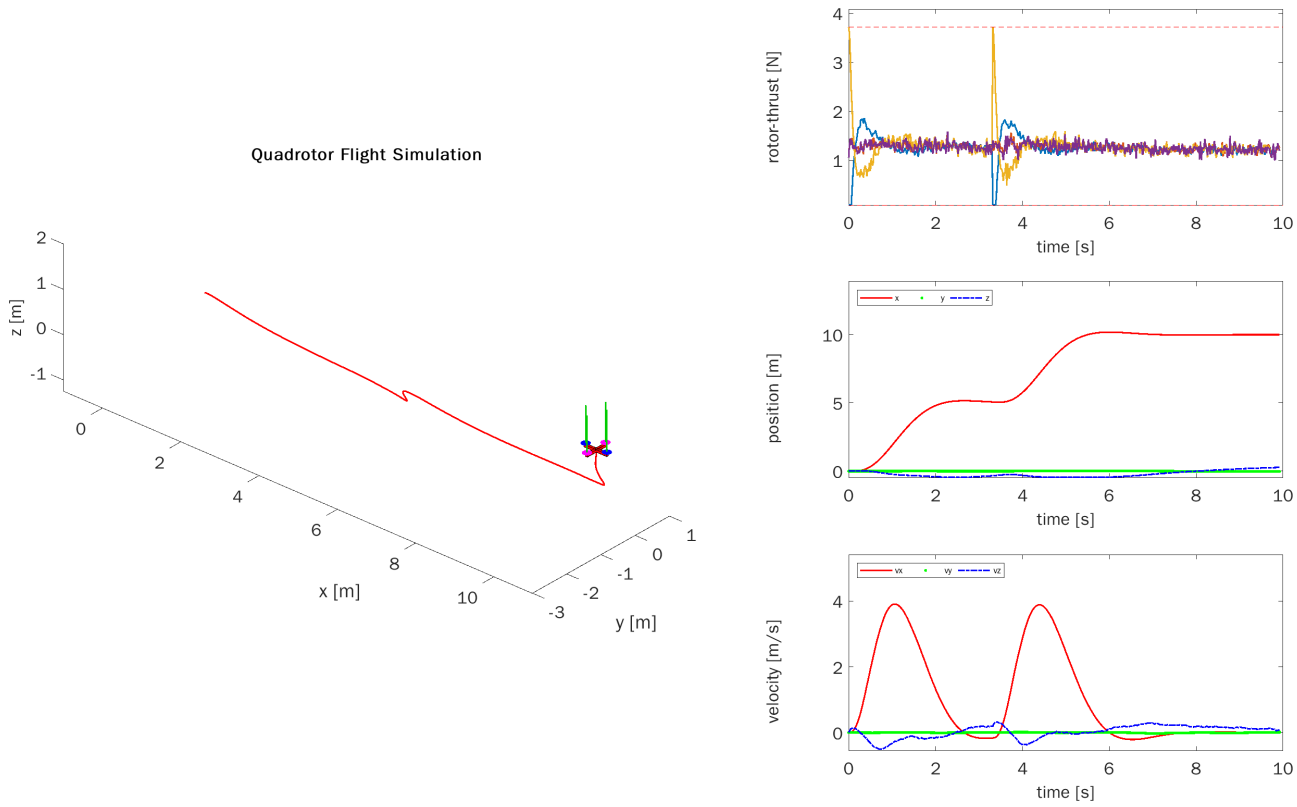
2.

The cause of the varying performance is the linear model used in computing LQR feedback gains. Th dynamics of quadrotor is nonlinear, and a linear dynamic model is only valid within a small region around linearisation point. Hence for goal states that are further away from linearisation point, the LQR controller fails to generate a stable policy.

c)

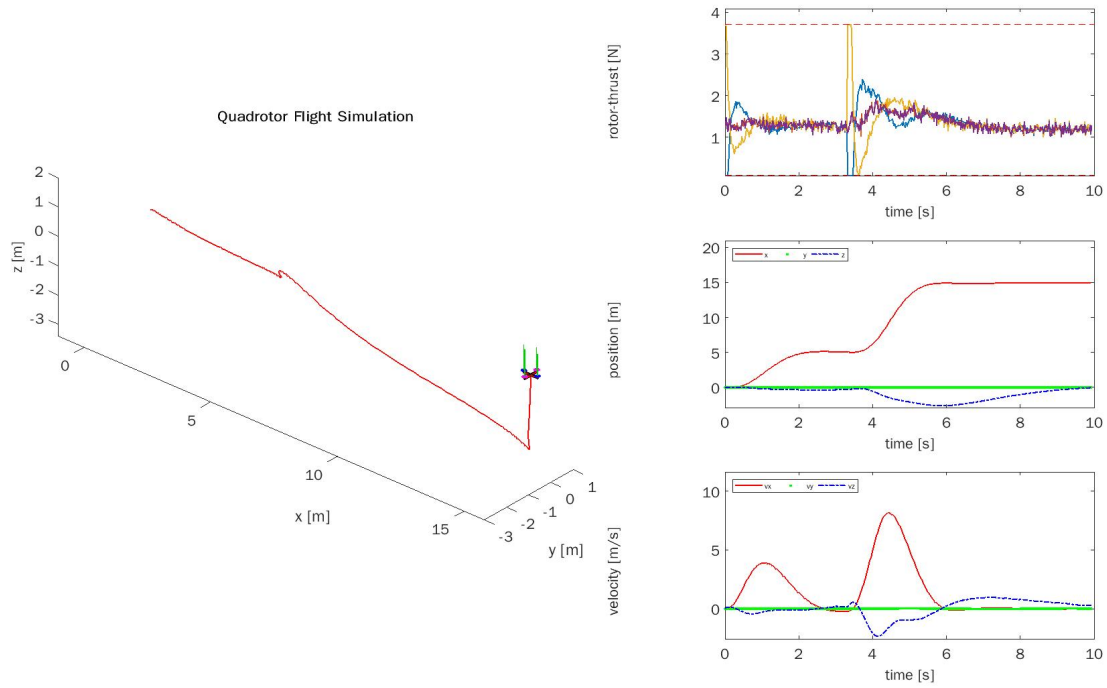
Code in Appendix:

1.
LQR with via point:



With the addition of via point, the quad rotor first reaches the via point and then moves on to the goal state. In contrast with (b), the z value remained around zero, while in (b), the quad rotor lost altitude while trying to move to the goal state, then climbed up in the final phase of trajectory. In the case with via point, the altitude loss is minimal.

2. In earlier case, $x_{goal} = [15,0,0,0,0,0,0,0,0,0,0,0]^T$
Was unreachable. This state is reachable now: Plots:



The system is able to reach farther states because the entire trajectory is split into two parts, with separate LQR feedback gains for traveling upto via point and from via point to goal state.

3. In first case, x_ref was set as goal_point from the beginning. Hence, the distance between x_ref and x_initial was larger than the radius of Lyapunov stability funnel of LQR controller.
In the second case, by adding a way point, the control problem breaks down to two separate control problems:

1. x_initial — ————————> x_vp
2. x_vp. ————————> x_goal

The euclidean distance between x_initial and x_vp and x_vp and x_goal, is lesser than the radius of Lyapunov stability funnel of LQR controller. Hence there appears to be a positive impact on system behaviour and stability.

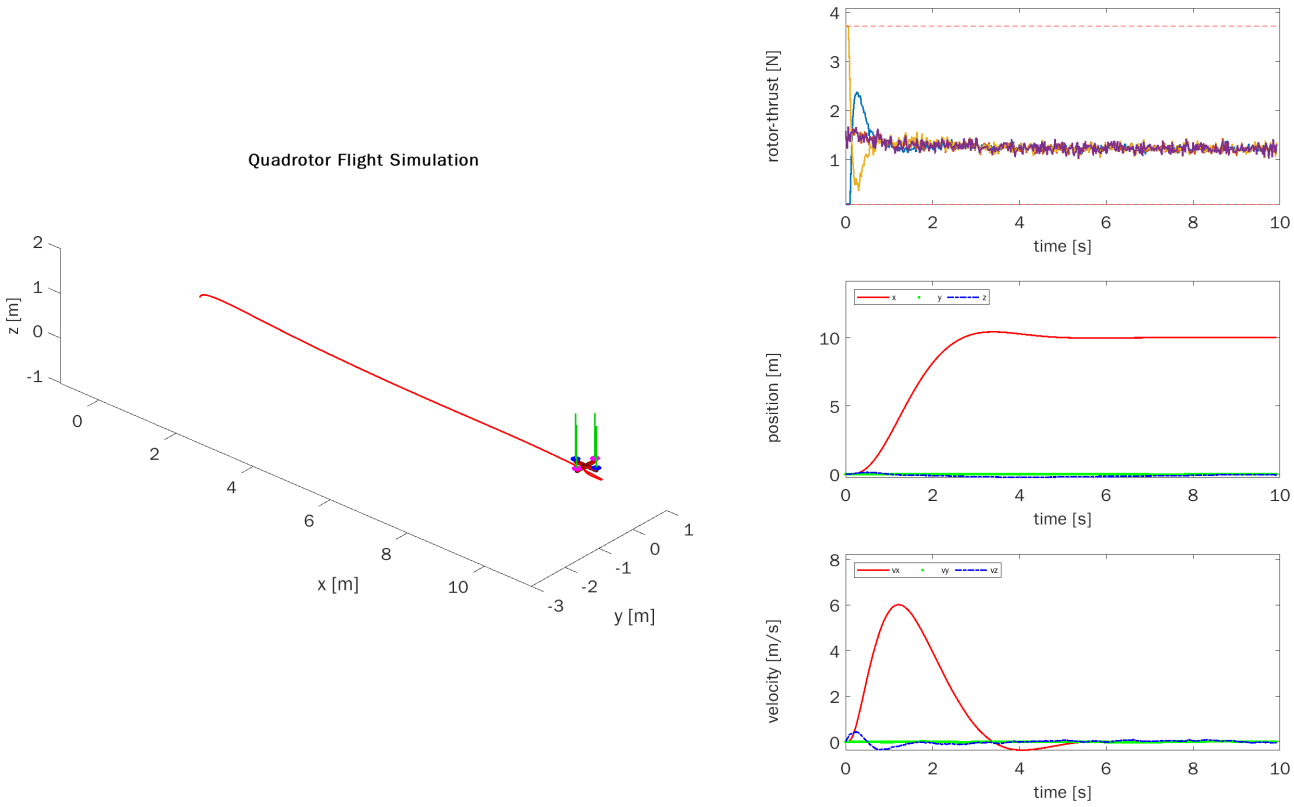
Disadvantages:

Adding a via point might not yield an optimal trajectory towards the goal point, as the quadrotor first try to reach vp and then moves to the goal state.

Furthermore, if the via point itself is difficult to reach, then the quad rotor might not reach the goal point.

d) Code in Appendix

1.



ILQC Controller for reaching goal state.

When ILQC controller is implemented, the altitude drop is lower than that of the LQR controller implementation. Furthermore, the cost of ILQC controller is lower than that of LQR controller.

2. ILQC perform better for distant and close goal states because:
The system dynamics is much more accurate in the ILQC formulation, when compared to the standard LQR formulation. In LQR, the system is linearised around a single equilibrium point, and if the goal states are further away from the start point, the system may not reach the goal state. On the other hand, in ILQC the system dynamics is linearised at each timestep and an affine control law is computed. Because of a much more accurate dynamics model in the ILQC formulation, the ILQC controller is able to reach farther goal states.

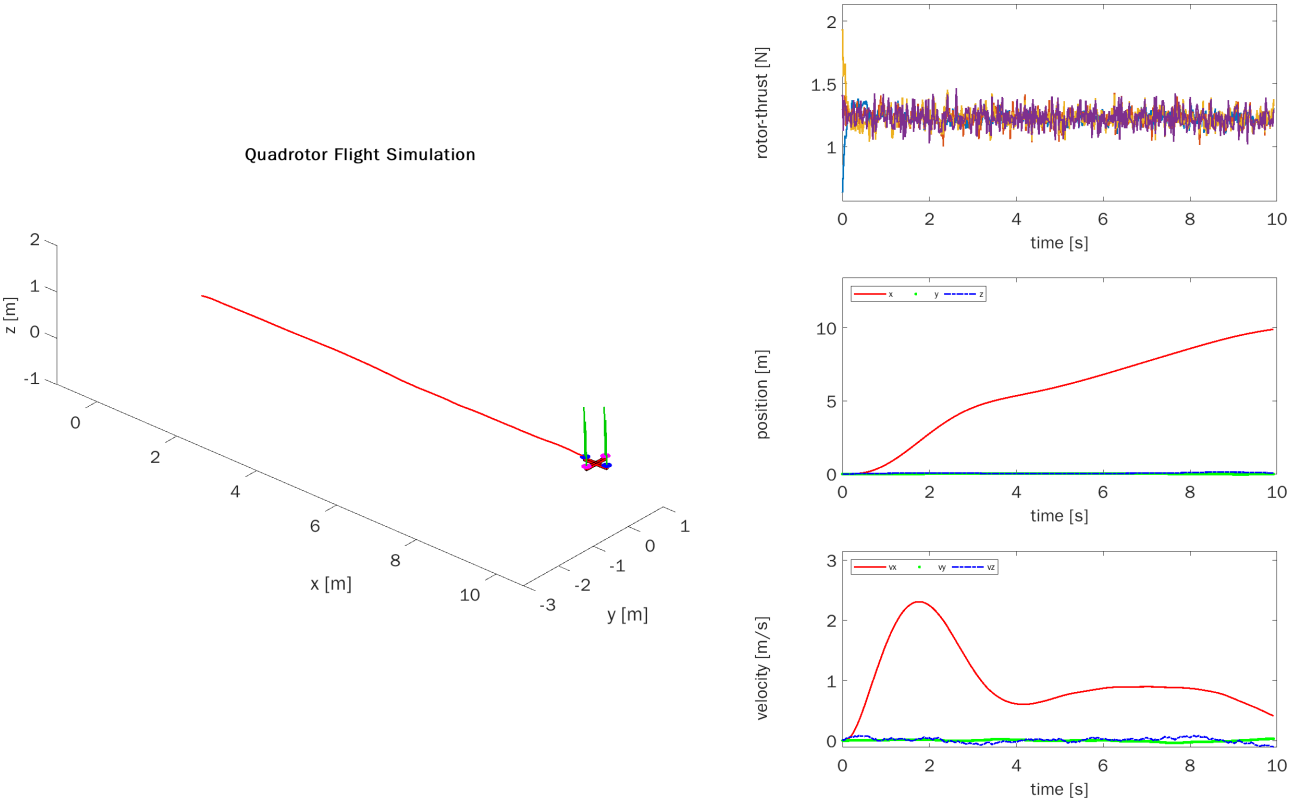
e)

Code in Appendix

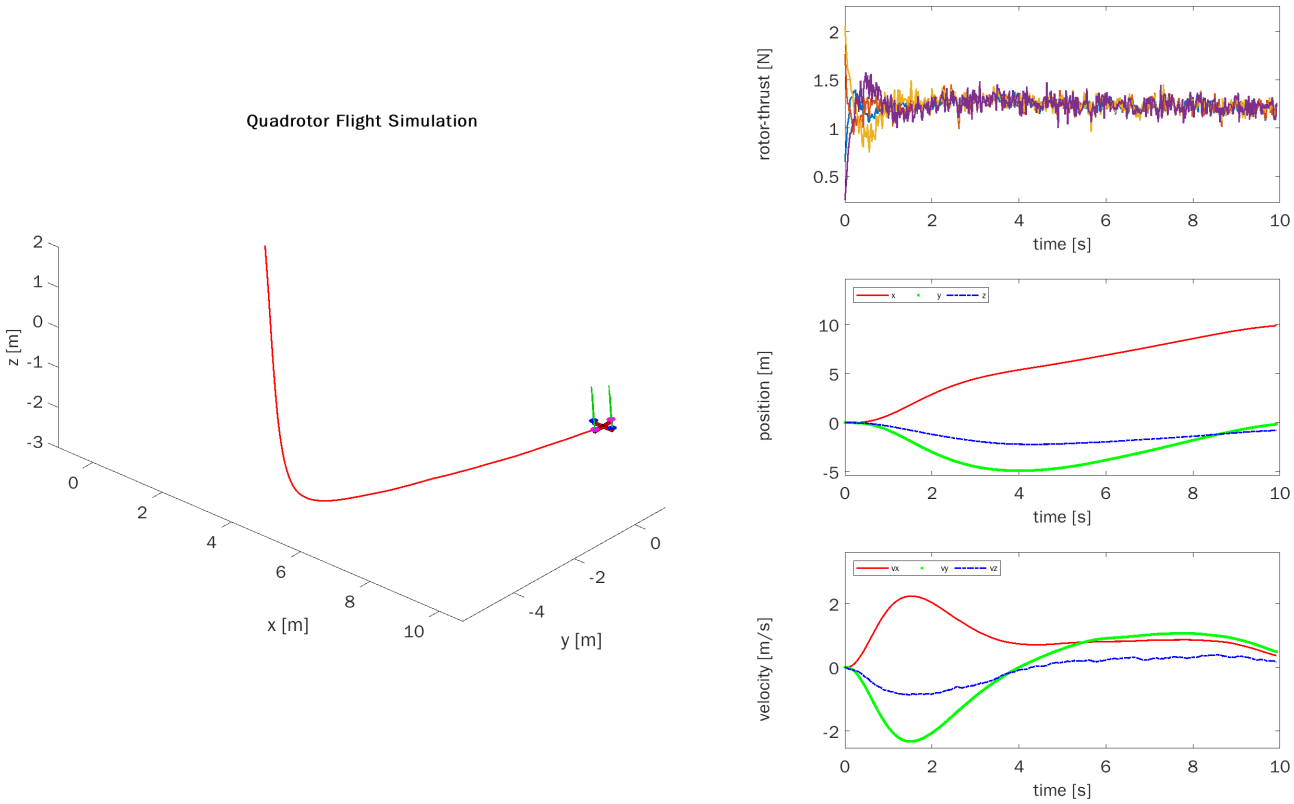
1. For the algorithm to determine optimal velocities error in velocities must not be penalised. This implies that corresponding terms in Q_vp should be equal to zero.

```
Q_vp = diag([  1   1    1   ...   % penalize positions
               0   0    0   ...   % orientations
               0   0    0  ...   % don't penalize linear velocities
               0   0    0 ]); % don't penalize angular velocities
```

Via_point 1: Plots:



Via_point 2: Plots:



2. Exact passing through the via point can be enforced by two methods:

1. Add a constraint to the optimisation problem with x(t_vp) = x_vp, and solve a constrained optimisation problem to compute optimal policy. This would deviate from the ILQC formulation, and also sometimes it may not be feasible to solve the constrained optimisation problem.
2. Exact passing through the via point can also be enforced by increasing ρ, as this would make the via point cost steeper close to t_vp. However this may result in numerical issues in ILQC gradient computations.

For including the via point only as a soft suggestion, low values of ρ can. Be used. In this case the cost becomes much more spread out, thereby enabling the algorithm to compute trajectories which pass close to the via point, around t_vp , without exactly reaching x_vp at t_vp.

APPENDIX

CODE:

2.1 b)

```
A = [0 const_vel; 0 0];
B = [0;1];
Q = [1 0;0 1];
R_s = 20;
[K,S,e] = lqr(A,B,Q,R_s)
x_goal = [-5;0]

theta_ff = K*x_goal

control_lqr = [theta_ff; -K(1,1); -K(1,2)];

for i = 1:N-2
temp = [theta_ff; -0.2236; -0.7051];
control_lqr = [control_lqr, temp];

end
controller_lqr = control_lqr ;
```

2.1 c)

```
% main_p1_lqr: Main script for Problem 2.1 LQR controller design.
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 2
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% --
% Revision history
% [20.01.31, SZ]    first version

clear all;
close all;
clc;

%% General
% add subdirectories
addpath(genpath(pwd));

% define task
task_lqr = task_design();
N = length(task_lqr.start_time:task_lqr.dt:task_lqr.end_time);

% add model
const_vel = 1; % desired forward speed
model = generate_model(const_vel);

% initialize controller
controller_lqr = zeros(3, N-1);

% save directory
save_dir = './results/';

% flags
plot_on = true;
save_on = true;
%% [Problem 2.1 (b)] LQR Controller
% ============================ [TODO] LQR Design ============================
% Design an LQR controller based on the linearization of the system about
% an equilibrium point (x_eq, u_eq). The cost function of the problem is
% specified in 'task_lqr.cost' via the method 'task_design()'.
```

```
%
%
% ========================================================================
A = [0 const_vel; 0 0];
B = [0;1];
Q = [1 0;0 1];
R_s = 20;
[K,S,e] = lqr(A,B,Q,R_s)
x_goal = [-5;0]
theta_ff = K*x_goal

%% [Problem 2.1 (c)] LQR Controller
% ============================ [TODO] LQR Design ============================
% Design an LQR controller based on the linearization of the system about
% an equilibrium point (x_eq, u_eq). The cost function of the problem is
% specified in 'task_lqr.cost' via the method 'task_design()'.
%
%
% ========================================================================
control_lqr = [theta_ff; -0.2236; -0.7051];
for i = 1:N-2
temp = [theta_ff; -K(1,1); -K(1,2)];
control_lqr = [control_lqr, temp];

end
controller_lqr = control_lqr
%% Simulation
sim_out_lqr = mobile_robot_sim(model, task_lqr, controller_lqr);
fprintf('--- LQR ---\n\n');
fprintf('trajectory cost: %.2f\n', sim_out_lqr.cost);
fprintf('target state [%.3f; %.3f]\n', task_lqr.goal_x);
fprintf('reached state [%.3f; %.3f]\n', sim_out_lqr.x(:,end));

%% Plots
if plot_on
    plot_results(sim_out_lqr);
end

%% Save controller and simulation results
if save_on
    if ~exist(save_dir, 'dir')
       mkdir(save_dir);
    end

    % save controller and simulation results
      save(strcat(save_dir, 'lqr_controller'), 'controller_lqr', ...
        'sim_out_lqr', 'task_lqr');
end
```

2.1 d)

```
function [A_k, B_k] = mobile_robot_lin(x_bar_k, u_bar_k, delta_t,v_bar)

          A_lin = [0 v_bar*cos(x_bar_k(2,1));0 0];
          I = [1 0;0 1]
          A_k = I + A_lin*delta_t
          B_k = delta_t*[0;1];

end
```

2.1 e)

Terminal Cost:

```
function [q_N,q_N_bold,Q_N] = terminal_cost_quad(Q_t, x_goal, x_N_bar)
Q_N = Q_t;
q_N_bold = Q_t*(x_N_bar - x_goal);
q_N = 0.5*((x_N_bar - x_goal)'*Q_t*(x_N_bar - x_goal));
end
```

Stage Cost:

```
function [q_k,q_k_bold,Q_k,r_k,R_k,P_k] =
stage_cost_quad(Q_s,R_s,x_goal,delta_t, x_k_bar, u_k_bar)
q_k = delta_t*(0.5*(transpose(x_k_bar - x_goal)*Q_s*(x_k_bar - x_goal) +
R_s*u_k_bar*u_k_bar));
q_k_bold = delta_t*(Q_s*(x_k_bar - x_goal));
Q_k = delta_t*Q_s;
r_k = delta_t*R_s*u_k_bar;
R_k = delta_t*R_s;
P_k = [0 0];

end
```

2.1 h)

```
function [theta_k_ff,theta_k_fb,s_k,s_k_bold,S_k] =
update_policy(A_k,B_k,q_k,q_k_bold,Q_k,r_k,R_k,P_k,s_k_next,s_k_bold_next,S_k_ne
xt,x_bar_k,u_bar_k)

g_k = r_k + (transpose(B_k))*s_k_bold_next ;% g_k equation  %correct
% (transpose(B_k))*S_k_next*A_k;

G_k = P_k + (transpose(B_k))*S_k_next*A_k   % G_k equation. Note: %correct

H_k = R_k + (transpose(B_k))*S_k_next*B_k % H_k equation %correct

K_k = -(inv(H_k))*G_k     % feed back term, equation for K_t %correct

delta_u_k_ff = -(inv(H_k))*g_k; %delta u feedforward term (Note: delta_u_k =
delta_u_k_ff + K_k*delta_x_k %correct

theta_k_ff = u_bar_k +  delta_u_k_ff - K_k*x_bar_k; % actual feedforward term
u_k_ff (Note: theta_k_ff in this code)

S_k = Q_k + (transpose(A_k))*S_k_next*A_k + transpose(K_k)*H_k*K_k +
transpose(K_k)*G_k + transpose(G_k)*K_k ;% Capital S_k bold

s_k_bold = q_k_bold + transpose(A_k)*s_k_bold_next +
transpose(K_k)*H_k*delta_u_k_ff + transpose(K_k)*g_k +
transpose(G_k)*delta_u_k_ff;

s_k = q_k + s_k_next + 0.5*(transpose(delta_u_k_ff))*H_k*delta_u_k_ff +
transpose(delta_u_k_ff)*g_k;

theta_k_fb = K_k;
end
```

2.1 j)

```
% main_p1_ilqc: Main script for Problem 2.1 ILQC controller design.
%
% --
% Control for Robotics
% AER1517 Spring 2022
% Assignment 2
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% --
% Revision history
% [20.01.31, SZ]    first version

clear all;
close all;

%% General
% add subdirectories
addpath(genpath(pwd));

% add task
task_ilqc = task_design();
N = length(task_ilqc.start_time:task_ilqc.dt:task_ilqc.end_time);

% add model
const_vel = 1; % assume constant forward speed
model = generate_model(const_vel);

% save directory
save_dir = './results/';

% initialize controller
load(strcat(save_dir, 'lqr_controller'));
 controller_ilqc = controller_lqr;
```

```
% flags
plot_on = true;
save_on = true;

%% [Problem 2.1 (j)] Iterative Linear Quadratic Controller
% ========================== [TODO] ILQC Design ==========================
% Design an ILQC controller based on the linearized dynamics and
% quadratized costs. The cost function of the problem is specified in
% 'task_ilqc.cost' via the method 'task_design()'.
%
%
% ========================================================================
theta = controller_ilqc
v_bar = const_vel;      %constant velocity
iterations = 0;
max_iterations = 20;
delta_t = task_ilqc.dt;

[ sim_out ] = mobile_robot_sim(model, task_ilqc, theta);
u_computed = sim_out.u;
x_computed = sim_out.x;


while iterations < max_iterations
iterations = iterations + 1;
%forward pass with current policy

[ sim_out ] = mobile_robot_sim(model, task_ilqc, theta);
u_computed = sim_out.u;
x_computed = sim_out.x;
% x_bar_k = sim_out.x(:,k);
% u_bar_k = sim_out.u(k);
% theta_k = sim_out.controller(:,k);
%%%%%%%%%% end of forward pass %%%%%%%%%%%%%%%

%%%%%%%%%% initialize backward pass %%%%%%%%%%%
%%%%%%%%%%initialize parameters %%%%%%%%%%%%%%%
 Q_t = task_ilqc.cost.params.Q_s;
 x_goal  =  task_ilqc.goal_x;
 x_N_bar =  sim_out.x(:,N);
[q_N,q_N_bold,Q_N] = terminal_cost_quad(Q_t, x_goal, x_N_bar);
Q_s = Q_t
R_s = task_ilqc.cost.params.R_s
s_N = q_N;
s_N_bold = q_N_bold;
S_N = Q_N;
%%%%%%%%%%  backward pass initialized %%%%%%%%%%%
%%%%%%%%%% initialize s_k_next,s_k_bold_next,S_k_next %%%%%%%%%
s_k_next = s_N
s_k_bold_next = s_N_bold
S_k_next = S_N

for k = N-1:-1:1
%compute x_bar_k , u_bar_k
x_bar_k = sim_out.x(:,k)
u_bar_k = sim_out.u(1,k)

%linearize dynamics around x_bar_k,u_bar_k
[A_k, B_k] = mobile_robot_lin(x_bar_k, u_bar_k, delta_t,v_bar);
%Approximate stage cost with second order taylor expansion to construct
%q_k,q_k_bold,Q_k,r_k,R_k,P_k
[q_k,q_k_bold,Q_k,r_k,R_k,P_k] = stage_cost_quad(Q_s,R_s,x_goal,delta_t,
x_bar_k, u_bar_k);
%%%%% update policy,s_k,s_k_bold,S_k
```

```
[theta_k_ff,theta_k_fb,s_k,s_k_bold,S_k] =
update_policy(A_k,B_k,q_k,q_k_bold,Q_k,r_k,R_k,P_k,s_k_next,s_k_bold_next,S_k_ne
xt,x_bar_k,u_bar_k);
theta_k = [theta_k_ff;theta_k_fb(1);theta_k_fb(2)];
theta(:,k) = theta_k;
s_k_next = s_k;
s_k_bold_next = s_k_bold;
S_k_next = S_k;

end

end

controller_ilqc = theta;
%% Simulation
sim_out_ilqc = mobile_robot_sim(model, task_ilqc, controller_ilqc);
fprintf('\n\ntarget state [%.3f; %.3f]\n', task_ilqc.goal_x);
fprintf('reached state [%.3f; %.3f]\n', sim_out_ilqc.x(:,end));

%% Plots
if plot_on
    plot_results(sim_out_ilqc);
end

%% Save controller and simulation results
if save_on
    if ~exist(save_dir, 'dir')
       mkdir(save_dir);
    end

    % save controller and simulation results
      save(strcat(save_dir, 'ilqc_controller'), 'controller_ilqc', ...
        'sim_out_ilqc', 'task_ilqc');
end
```

2.2 a)

```
function [LQR_Controller, Cost] = LQR_Design(Model,Task)
% LQR_DESIGN Creates an LQR controller to execute the Task
% Controller: Controller structure
%             u(x) = theta([t]) ' * BaseFnc(x)
%             .theta:   Parameter matrix (each column is associated with
%                       one control input)
%             .BaseFnc
%             .time

%% Initializations
state_dim = 12;
input_dim = 4;
K = zeros(input_dim, state_dim);
theta = init_theta();
theta_k = theta;

%% [Problem 2.2 (a)] Find optimal feedback gains according to an LQR controller
% Make use of the elements in Task.cost for linearization points and cost
% functions.

% ========================================================================
% [Todo] Define the state input around which the system dynamics should be
% linearized assuming that most of the time the quadrotor flies similar to
% hovering (Fz != 0)
%
% x_lin = ...;
% u_lin = ...;
  x_lin = [0;0;0;0;0;0;0;0;0;0;0;0];
  u_lin = [5;0;0;0];

% [Todo] The linearized system dynamics matrices A_lin B_lin describe the
% dynamics of the system around the equilibrium state. Use Model.Alin{1}
% and Model.Blin{1}.
%
% A_lin = ...;
% B_lin = ...;
A_lin = Model.Alin{1}(x_lin, u_lin, Model.param.syspar_vec)
B_lin = Model.Blin{1}(x_lin, u_lin, Model.param.syspar_vec)

% [Todo] Compute optimal LQR gain (see command 'lqr')
% Quadratic cost defined as Task.cost.Q_lqr,Task.cost.R_lqr
%
% K = ...;
Q = Task.cost.Q_lqr
R_s = Task.cost.R_lqr
K = lqr(A_lin,B_lin,Q,R_s)
% ========================================================================

%% Design the actual controller using the optimal feedback gains K
% The goal of this task is to correctly fill the matrix theta.
LQR_Controller.BaseFnc = @Augment_Base;
LQR_Controller.time    = Task.start_time:Task.dt:(Task.goal_time-Task.dt);

% The quadrotor controller produces inputs u from a combination of
% feedforward uff and feedback elements as follows:
%   u = [Fz, Mx, My, Mz]' = uff + K'(x_ref - x)
%                         = uff + K'x_ref - K'x
%                         = [uff + K'x_ref, -K' ]' * [1,x']'
%                         =        theta'          * BaseFnc
uff = u_lin
x_ref = Task.goal_x
```

```
theta_ff = uff + K*x_ref
theta_fb = -K
theta = [theta_ff,theta_fb]
theta = theta'
```

2.2 b)

```
lqr_type = 'via_point';  % Choose 'goal_state' or 'via_point'
fprintf('LQR controller design type: %s \n', lqr_type);
Nt = ceil((Task.goal_time - Task.start_time)/Task.dt+1);

switch lqr_type
    case 'goal_state'
%% [Problem 2.2 (b)] Drive system to Task.goal_x with LQR Gain + feedforward
        % ================================================================
        % [Todo] Define equilibrium point x_eq correctly and use it to
        % generate feedforward torques (see handout Eqn.(12) and notes
        % above).
        %
        % x_ref = ...; % reference point the controller tries to reach
        % theta = ...; % dimensions (13 x 4)
        % ================================================================

        uff = u_lin
        x_ref = Task.goal_x
        theta_ff = uff + K*x_ref
        theta_fb = -K
        theta = [theta_ff,theta_fb]
        theta = theta'

        % stack constant theta matrices for every time step Nt
        LQR_Controller.theta = repmat(theta,[1,1,Nt]);
```

2.2 c)

```
    case 'via_point'
%% [Problem 2.2 (c)] Drive system to Task.goal_x through via-point p1.
        t1 = Task.vp_time;
        p1 = Task.vp1; % steer towards this input until t1

        for t=1:Nt-1
           % ============================================================
           % [Todo] x_ref at current time
           % ...
           t_1_k = t1/Task.dt
            if t <= t_1_k

                x_ref = Task.vp1;

            else

                x_ref = Task.goal_x;
            end

           %
           % [Todo] time-varying theta matrices for every time step Nt
           % (see handout Eqn.(12)) (size: 13 x 4)
           % theta_k = ...;
           % ============================================================
            uff = u_lin;
            theta_ff = uff + K*x_ref;
            theta_fb = -K;
            theta = [theta_ff,theta_fb];
            theta_k = theta';
           % save controller gains to struct
           LQR_Controller.theta(:,:,t) = theta_k;
        end

    otherwise
        error('Unknown lqr_type');
end
```

LQR Design Complete code:

```
% LQR_Design: Implementation of the LQR controller.
%
% Control for Robotics
% AER1517 Spring 2022
% Assignment 2
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% This script is adapted from the course on Optimal & Learning Control for
% Autonomous Robots at the Swiss Federal Institute of Technology in Zurich
% (ETH Zurich). Course Instructor: Jonas Buchli. Course Webpage:
% http://www.adrlab.org/doku.php/adrl:education:lecture:fs2015
%
% --
% Revision history
% [20.01.31]    first version

function [LQR_Controller, Cost] = LQR_Design(Model,Task)
% LQR_DESIGN Creates an LQR controller to execute the Task
% Controller: Controller structure
%             u(x) = theta([t]) ' * BaseFnc(x)
%             .theta:   Parameter matrix (each column is associated with
%                       one control input)
%             .BaseFnc
%             .time

%% Initializations
state_dim = 12;
input_dim = 4;
K = zeros(input_dim, state_dim);
theta = init_theta();
theta_k = theta;

%% [Problem 2.2 (a)] Find optimal feedback gains according to an LQR controller
% Make use of the elements in Task.cost for linearization points and cost
% functions.

% ========================================================================
% [Todo] Define the state input around which the system dynamics should be
% linearized assuming that most of the time the quadrotor flies similar to
% hovering (Fz != 0)
%
% x_lin = ...;
% u_lin = ...;
  x_lin = [0;0;0;0;0;0;0;0;0;0;0;0];
  u_lin = [5;0;0;0];

% [Todo] The linearized system dynamics matrices A_lin B_lin describe the
```

```
% dynamics of the system around the equilibrium state. Use Model.Alin{1}
% and Model.Blin{1}.
%
% A_lin = ...;
% B_lin = ...;
A_lin = Model.Alin{1}(x_lin, u_lin, Model.param.syspar_vec)
B_lin = Model.Blin{1}(x_lin, u_lin, Model.param.syspar_vec)

% [Todo] Compute optimal LQR gain (see command 'lqr')
% Quadratic cost defined as Task.cost.Q_lqr,Task.cost.R_lqr
%
% K = ...;
Q = Task.cost.Q_lqr
R_s = Task.cost.R_lqr
K = lqr(A_lin,B_lin,Q,R_s)
% ======================================================================

%% Design the actual controller using the optimal feedback gains K
% The goal of this task is to correctly fill the matrix theta.
LQR_Controller.BaseFnc = @Augment_Base;
LQR_Controller.time    = Task.start_time:Task.dt:(Task.goal_time-Task.dt);

% The quadrotor controller produces inputs u from a combination of
% feedforward uff and feedback elements as follows:
%   u = [Fz, Mx, My, Mz]' = uff + K'(x_ref - x)
%                         = uff + K'x_ref - K'x
%                         = [uff + K'x_ref, -K' ]' * [1,x']'
%                         =        theta'          * BaseFnc
uff = u_lin
x_ref = Task.goal_x
theta_ff = uff + K*x_ref
theta_fb = -K
theta = [theta_ff,theta_fb]
theta = theta'

lqr_type = 'goal_state';  % Choose 'goal_state' or 'via_point'
fprintf('LQR controller design type: %s \n', lqr_type);
Nt = ceil((Task.goal_time - Task.start_time)/Task.dt+1);

switch lqr_type
    case 'goal_state'
%% [Problem 2.2 (b)] Drive system to Task.goal_x with LQR Gain + feedforward
        % ================================================================
        % [Todo] Define equilibrium point x_eq correctly and use it to
        % generate feedforward torques (see handout Eqn.(12) and notes
        % above).
        %
        % x_ref = ...; % reference point the controller tries to reach
        % theta = ...; % dimensions (13 x 4)
        % ================================================================
        x_ref = Task.goal_x
        % stack constant theta matrices for every time step Nt
        LQR_Controller.theta = repmat(theta,[1,1,Nt]);

    case 'via_point'
%% [Problem 2.2 (c)] Drive system to Task.goal_x through via-point p1.
        t1 = Task.vp_time;
        p1 = Task.vp1; % steer towards this input until t1

        for t=1:Nt-1
           % ============================================================
           % [Todo] x_ref at current time
           % ...
           t_1_k = t1/Task.dt
            if t <= t_1_k
```

```
                x_ref = Task.vp1;

            else

                x_ref = Task.goal_x;
            end

            %
            % [Todo] time-varying theta matrices for every time step Nt
            % (see handout Eqn.(12)) (size: 13 x 4)
            % theta_k = ...;
            % ============================================================
             uff = u_lin;
             theta_ff = uff + K*x_ref;
             theta_fb = -K;
             theta = [theta_ff,theta_fb];
             theta_k = theta';
            % save controller gains to struct
            LQR_Controller.theta(:,:,t) = theta_k;
        end

    otherwise
        error('Unknown lqr_type');
end

% Calculate cost of rollout
sim_out = Quad_Simulator(Model, Task, LQR_Controller);
Cost = Calculate_Cost(sim_out, Task);
end

function x_aug = Augment_Base(t,x)
% AUGMENT_BASE(t,x) Allows to incorporate feedforward and feedback
%
%   Reformulating the affine control law allows incorporating
%   feedforward term in a feedback-like fashion:
%   u = [Fz, Mx, My, Mz]' = uff + K(x - x_ref)
%                         = uff - K*x_ref + K*x
%                         = [uff - K*x_ref, K ] * [1,x']'
%                         =        theta'       * BaseFnc

number_of_states = size(x,2);      % multiple states x can be augmented at once
x_aug = [ones(1,number_of_states); % augment row of ones
                    x           ];
end


function Cost = Calculate_Cost(sim_out, Task)
% Calculate_Cost(.) Asses the cost of a rollout sim_out
%
%   Be sure to correctly define the equilibrium state (X_eq, U_eq) the
%   controller is trying to reach.
X = sim_out.x;
U = sim_out.u;
Q = Task.cost.Q_lqr;
R = Task.cost.R_lqr;
X_eq = repmat(Task.cost.x_eq,1,size(X,2)-1); % equilibrium state LQR controller
tries to reach
U_eq = repmat(Task.cost.u_eq,1,size(U,2));   % equilibrium input LQR controller
tries to reach
Ex = X(:,1:end-1) - X_eq;                    % error in state
Eu = U - U_eq;                               % error in input

Cost = Task.dt * sum(sum(Ex.*(Q*Ex),1) + sum(Eu.*(R*Eu),1));
end
```

2.2 d)

```
% ILQC_Design: Implementation of the ILQC controller.
%
% Control for Robotics
% AER1517 Spring 2022
% Assignment 2
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% This script is adapted from the course on Optimal & Learning Control for
% Autonomous Robots at the Swiss Federal Institute of Technology in Zurich
% (ETH Zurich). Course Instructor: Jonas Buchli. Course Webpage:
% http://www.adrlab.org/doku.php/adrl:education:lecture:fs2015
%
% --
% Revision history
% [20.01.31]    first version

function [Controller,cost] = ILQC_Design(Model,Task,Controller,Simulator)
% ILQC_DESIGN Implements the Iterative Linear Quadratic Controller (ILQC)
%    (see Ch. 4 notes for a formal description of the algorithm)

% Define functions that return the quadratic approximations of the cost
% function at specific states and inputs (Eqns. (6)-(7) in handout)
% Example usage:
%     xn = [ x y z ... ]';
%     un = [ Fx Mx My Mz]';
%     t  = t;
%     Qm(xn,un) = Qm_fun(t,xn,un);

% stage cost (l) quadratizations
l_ = Task.cost.l*Task.dt;
q_fun   = matlabFunction(  l_,'vars',{Task.cost.t,Task.cost.x,Task.cost.u});
% dl/dx
l_x = jacobian(Task.cost.l,Task.cost.x)'*Task.dt; % cont -> discr. time
Qv_fun  = matlabFunction( l_x,'vars',{Task.cost.t,Task.cost.x,Task.cost.u});
% ddl/dxdx
l_xx = jacobian(l_x,Task.cost.x);
Qm_fun  = matlabFunction(l_xx,'vars',{Task.cost.t,Task.cost.x,Task.cost.u});
% dl/du
l_u = jacobian(Task.cost.l,Task.cost.u)'*Task.dt; % cont -> discr. time
Rv_fun  = matlabFunction( l_u,'vars',{Task.cost.t,Task.cost.x,Task.cost.u});
% ddl/dudu
l_uu = jacobian(l_u,Task.cost.u);
Rm_fun  = matlabFunction(l_uu,'vars',{Task.cost.t,Task.cost.x,Task.cost.u});
% ddl/dudx
l_xu = jacobian(l_x,Task.cost.u)';
Pm_fun  = matlabFunction(l_xu,'vars',{Task.cost.t,Task.cost.x,Task.cost.u});
```

```
% terminal cost (h) quadratizations
h_ = Task.cost.h;
qf_fun  = matlabFunction(  h_,'vars',{Task.cost.x});
% dh/dx
h_x = jacobian(Task.cost.h,Task.cost.x)';
Qvf_fun = matlabFunction( h_x,'vars',{Task.cost.x});
% ddh/dxdx
h_xx = jacobian(h_x,Task.cost.x);
Qmf_fun = matlabFunction(h_xx,'vars',{Task.cost.x});

% dimensions
n = length(Task.cost.x); % dimension of state space
m = length(Task.cost.u); % dimension of control input
N  = (Task.goal_time-Task.start_time)/Task.dt + 1; % number of time steps

% desired value function V* is of the form Eqn.(9) in handout
% V*(dx,n) = s + dx'*Sv + 1/2*dx'*Sm*dx
s    = zeros(1,N);
Sv   = zeros(n,N);
Sm   = zeros(n,n,N);

% Initializations
theta_temp = init_theta();
duff = zeros(m,1,N-1);
K    = repmat(theta_temp(2:end,:)', 1, 1, N-1);
sim_out.t = zeros(1, N);
sim_out.x = zeros(n, N);
sim_out.u = zeros(m, N-1);
X0 = zeros(n, N);
U0 = zeros(m, N-1);

% Shortcuts for function pointers to linearize systems dynamics:
% e.g. Model_Alin(x,u,Model_Param)
Model_Param = Model.param.syspar_vec;
Model_Alin  = Model.Alin{1};
Model_Blin  = Model.Blin{1};

%% [Problem 2.2 (d)] Implementation of ILQC controller
% Each ILQC iteration approximates the cost function as quadratic around the
% current states and inputs and solves the problem using DP.
i  = 1;
while ( i <= Task.max_iteration && ( norm(squeeze(duff)) > 0.01 || i == 1 ))

    %% Forward pass / "rollout" of the current policy
    % ========================================================================
    % [Todo] rollout states and inputs
    % sim_out = ...;
    % ========================================================================
    sim_out = Simulator(Model,Task,Controller)
    % pause if cost diverges
    cost(i) = Calculate_Cost(sim_out, q_fun, qf_fun);
    fprintf('Cost of Iteration %2d (metric: ILQC cost function!): %6.4f \n',
i-1, cost(i));

    if ( i > 1 && cost(i) > 2*cost(i-1) )
        fprintf('It looks like the solution may be unstable. \n')
        fprintf('Press ctrl+c to interrupt iLQG, or any other key to continue.
\n')
        pause
    end

    %% Solve Riccati-like equations backwards in time
      % ========================================================================
    % [Todo] define nominal state and control input trajectories (dim by
    % time steps). Note: sim_out contains state x, input u, and time t
```

```
%
X0 = sim_out.x; % x_bar
U0 = sim_out.u; % u_bar
T0 = sim_out.t; % t
% =========================================================================

Q_t = Task.cost.Qmf;
Q_s = Task.cost.Qm;
R_s = Task.cost.Rm;
state_goal = Task.goal_x;
I = eye(12);
% =========================================================================
% [Todo] Initialize the value function elements starting at final time
% step (Problem 2.2 (g))
%
xf = X0(:,end); % final state when using current controller
Sm(:,:,N) = Qmf_fun(xf);
Sv(:,N) =  Qvf_fun(xf);
s(N) = qf_fun(xf);
% =========================================================================

% "Backward pass": Calculate the coefficients (s,Sv,Sm) for the value
% functions at earlier times by proceeding backwards in time
% (DP-approach)
delta_t = Task.dt;
Sm_next = Sm;
Sv_next = Sv;
s_next = s;
for k = (length(sim_out.t)-1):-1:1

    % state of system at time step n
    x0 = X0(:,k);
    u0 = U0(:,k);

    % =====================================================================
    % [Todo] Discretize and linearize continuous system dynamics Alin
    % around specific pair (x0,u0). See exercise sheet Eqn. (18) for
    % details.
    %
    % Alin = ...;
    % Blin = ...;
    A_lin = Model.Alin{1}(x0, u0, Model.param.syspar_vec);
    B_lin = Model.Blin{1}(x0, u0, Model.param.syspar_vec);
    % A = ...;
    % B = ...;
    A = I + A_lin*delta_t;
    B = B_lin*delta_t;
    % =====================================================================

    % =====================================================================
    % [Todo] quadratize cost function
    % [Note] use function {q_fun, Qv_fun, Qm_fun, Rv_fun, Rm_fun,
    % Pm_fun} provided above.
    %
    t0 = T0(:,k);
    % q = ...;
    % Qv = ...;
    % Qm = ...;
    % Rv = ...;
    % Rm = ...;
    % Pm = ...;
    q = q_fun(t0,x0,u0);
    Qv = Qv_fun(t0,x0,u0);
    Qm = Qm_fun(t0,x0,u0);
    Rv = Rv_fun(t0,x0,u0);
    Rm = Rm_fun(t0,x0,u0);
```

```
        Pm = Pm_fun(t0,x0,u0);
        % ======================================================================

        % ======================================================================
        % [Todo] control dependent terms of cost function (Problem 2.2 (f))
        %
        % g = ...;      % linear control dependent
        % G = ...;      % control and state dependent
        % H = ...;      % quadratic control dependent
        %
        g = Rv + transpose(B)*Sv(:,k+1);
        G = Pm + transpose(B)*Sm(:,:,k+1)*A ;
        H = Rm + transpose(B)*Sm(:,:,k+1)*B ;

        H = (H+H')/2; % ensuring H remains symmetric; do not delete!
        % ======================================================================

        % ======================================================================
        % [Todo] the optimal change of the input trajectory du = duff +
        % K*dx (Problem 2.2 (f))
        %
        duff(:,:,k) = -inv(H)*g;
        K(:,:,k) = -inv(H)*G;
        % ======================================================================

        % ======================================================================
        % [Todo] Solve Riccati-like equations for current time step n
        % (Problem 2.2 (g))
        %
        Sm(:,:,k) = Qm + transpose(A)*Sm(:,:,k+1)*A +
transpose(K(:,:,k))*H*K(:,:,k) + transpose(K(:,:,k))*G + transpose(G)*K(:,:,k);
        Sv(:,k) = Qv + transpose(A)*Sv(:,k+1) +
transpose(K(:,:,k))*H*duff(:,:,k) + transpose(K(:,:,k))*g +
transpose(G)*duff(:,:,k);
        s(k) = q + s(k+1) + 0.5*(transpose(duff(:,:,k)))*H*duff(:,:,k) +
transpose(duff(:,:,k))*g;
        % ======================================================================

    end % of backward pass for solving Riccati equation

    % define theta_ff in this function
    Controller.theta = Update_Controller(X0, U0, duff, K);

    i = i+1;
end

% simulating for the last update just to calculate the final cost
sim_out    = Simulator(Model,Task,Controller);
cost(i) = Calculate_Cost(sim_out, q_fun, qf_fun);
fprintf('Cost of Iteration %2d: %6.4f \n', i-1, cost(i));
end



function theta = Update_Controller(X0,U0,dUff,K)
% UPDATE_CONTROLLER Updates the controller after every ILQC iteration
%
%  X0  - state trajectory generated with controller from previous
%        ILQC iteration.
%  UO  - control input generated from previous ILQC iteration.
%  dUff- optimal update of feedforward input found in current iteration
%  K   - optimal state feedback gain found in current iteration
%
%  The updated control policy has the following form:
%  U1 = U0 + dUff + K(X - X0)
%     = U0 + dUff - K*X0 + K*X
```

```
%     =      Uff         + K*x
%
%  This must be brought into the form
%  U1 = theta' * [1,x']

%% Update Controller
% input and state dimensions
n  = size(X0,1); % dimension of state
m = size(U0,1); % dimension of control input
N  = size(X0,2); % number of time steps

% initialization
theta = init_theta();
theta_fb = zeros(n, m, N-1);
theta_ff = repmat(theta(1,:), 1, 1, N-1);

% ========================================================================
% [Todo] feedforward control input
%
U0 = permute(U0, [3, 1, 2]);
dUff = permute(dUff, [2, 1, 3]);
KX0 = zeros(1, m, N-1);
for i = 1:1:N-1
   KX0(:,:,i) = K(:,:,i)*X0(:,i);
end
%
theta_ff = U0 + dUff -KX0;
% ========================================================================

% feedback gain of control input (size: n * m * N-1)
theta_fb = permute(K,[2 1 3]);

% puts below (adds matrices along first(=row) dimension).
% (size: (n+1) * m * N-1)
theta = [theta_ff; theta_fb];

end


function cost = Calculate_Cost(sim_out, q_fun, qf_fun)
% CALCULATE_COST: calcules the cost of current state and input trajectory
% for the current ILQC cost function. Not neccessarily the same as the LQR
% cost function.

X0 = sim_out.x(:,1:end-1);
xf = sim_out.x(:,end);
U0 = sim_out.u;
T0 = sim_out.t(1:end-1);

cost = sum(q_fun(T0,X0,U0)) + qf_fun(xf);
end
```

2.2 e)

```
% Cost_Design: Definition of cost functions for reaching goal state and/or
% passing through via-point.
%
% Control for Robotics
% AER1517 Spring 2022
% Assignment 2
%
% --
% University of Toronto Institute for Aerospace Studies
% Dynamic Systems Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistant:
% SiQi Zhou
% siqi.zhou@robotics.utias.utoronto.ca
% Lukas Brunke
% lukas.brunke@robotics.utias.utoronto.ca
% Adam Hall
% adam.hall@robotics.utias.utoronto.ca
%
% This script is adapted from the course on Optimal & Learning Control for
% Autonomous Robots at the Swiss Federal Institute of Technology in Zurich
% (ETH Zurich). Course Instructor: Jonas Buchli. Course Webpage:
% http://www.adrlab.org/doku.php/adrl:education:lecture:fs2015
%
% --
% Revision history
% [20.01.31]    first version

function Cost = Cost_Design( m_quad, Task)
%COST_DESIGN Creates a cost function for LQR and for ILQC
%   .Q_lqr
%   .R_lqr
%
%   A ILQC cost J = h(x) + sum(l(x,u)) is defined by:
%   .h   - continuous-time terminal cost
%   .l - continous-time intermediate cost

% Quadcopter system state x
syms qxQ qyQ qzQ qph qth qps dqxQ dqyQ dqzQ dqph dqth dqps real
x_sym = [ qxQ qyQ qzQ ...    % position x,y,z
          qph qth qps ...    % roll, pitch, yaw
          dqxQ dqyQ dqzQ ... % velocity x,y,z
          dqph dqth dqps ]'; % angular velocity roll, pitch, yaw

% Quadcopter input u (Forces / Torques)
syms Fz Mx My Mz real; u_sym = [ Fz Mx My Mz ]';

% Time variable for time-varying cost
syms t_sym real;
Cost = struct;


%% LQR cost function
% LQR controller
% Q needs to be symmetric and positive semi-definite
Cost.Q_lqr = diag([  1   1    1   ...   % penalize positions
                     3   3    3   ...   % penalize orientations
```

```
                    0.1  0.1    2   ...   % penalize linear velocities
                    1   1    1 ]);     % penalize angular velocities
% R needs to be positive definite
Cost.R_lqr = 10*diag([1 1 1 1]);        % penalize control inputs

%% ILQC cost function
Cost.Qm  = Cost.Q_lqr;                  % it makes sense to redefine these
Cost.Rm  = Cost.R_lqr;
Cost.Qmf = Cost.Q_lqr;

% Reference states and input the controller is supposed to track
gravity = 9.81;
f_hover = m_quad*gravity; % keep input close to the one necessary for hovering
Cost.u_eq = [ f_hover ; zeros(3,1) ];
Cost.x_eq = Task.goal_x;

% alias for easier referencing
x = x_sym;
u = u_sym;
x_goal = Task.goal_x;

ilqc_type = 'via_point'; % Choose 'goal_state or 'via_point'
fprintf('ILQC cost function type: %s \n', ilqc_type);
switch ilqc_type
    case 'goal_state'
        Cost.h = simplify((x-x_goal)'*Cost.Qmf*(x-x_goal));
        Cost.l = simplify( (x-Cost.x_eq)'*Cost.Qm*(x-Cost.x_eq) ...
             + (u-Cost.u_eq)'*Cost.Rm*(u-Cost.u_eq));

    case 'via_point'
%% [Problem 1.2 (e)] Include via_point p1 in the ILQC cost function formulation
        p1 = Task.vp1;      % p1 = Task.vp2 also try this one
        t1 = Task.vp_time;

        % ====================================================================
        % [Todo] Define an appropriate weighting for way points (see
        % handout Eqn.(19)) Hint: Which weightings must be zero for the
        % algorithm to determine optimal values?
        %
        % Q_vp = ...;
        % ====================================================================

      Q_vp = diag([  1   1    1   ...   % penalize positions
                     0   0    0   ...   % penalize orientations
                     0   0    0  ...   % don't penalize linear velocities
                     0   0    0 ]); % don't penalize angular velocities
        % don't penalize position deviations, drive system with final cost
        Cost.Qm(1:3,1:3) = zeros(3);

        % ====================================================================
        % [Todo] Define symbolic cost function.
        % Note: Use function "viapoint(.)" below
        %
        viapoint_cost = viapoint(t1,p1,x,t_sym,Q_vp);
        Cost.h = simplify((x-x_goal)'*Cost.Qmf*(x-x_goal));
        Cost.l = simplify( (x-Cost.x_eq)'*Cost.Qm*(x-Cost.x_eq) + (u-
Cost.u_eq)'*Cost.Rm*(u-Cost.u_eq)) +  viapoint_cost;
        % ====================================================================

      otherwise
        error('Unknown ilqc cost function mode');
end

Cost.x  = x;
Cost.u  = u;
Cost.t  = t_sym;
```

```
end


function viapoint_cost = viapoint(vp_t,vp,x,t,Qm_vp)
% WAYPOINT Creates cost depending on deviation of the system state from a
% desired state vp at time t. Doesn't need to be modified.
% For more details see:
% http://www.adrlab.org/archive/p_14_mlpc_quadrotor_cameraready.pdf
%    vp_t  :   time at which quadrotor should be at viapoint wp
%    vp    :   position where quad should be at given time instance
%    x,t   :   symbolic references to state and time
%    Qm_vp :   The weighting matrix for deviation from the via-point

prec = 3; % how 'punctual' does the quad have to be? The bigger the
          % number, the harder the time constraint

viapoint_cost = (x-vp)'*Qm_vp*(x-vp) ...
                *exp(-0.5*prec*(t-vp_t)^2) /sqrt(2*pi/prec);

end
```