# AER 1517: Assignment 1

# Model-Based Iterative Linear Quadratic Control

Abhinav Grover (groverab)

University of Toronto

Febraury 24th, 2020

## 1.0 Mobile Robot

### 1.1 Linear Quadratic Regulator

The mobile robot in question is assumed to follow a bicycle model. The system model is shown below:

$$\dot{y}(t) = v(t)sin(h(t))$$
$$\dot{h}(t) = \omega(t)$$

It is given that $v(t) = \bar{v}$ is a constant. At equilibrium, the conditions $\dot{y}_{eq} = 0$ and $\dot{h}_{eq} = 0$ are true. Thus, the equilibrium point is caculated to be $\omega_{eq} = 0, h_{eq} = 0$ and $y_{eq} = y_{goal}$. The linearization steps are shown below.

$$\mathbf{x}(t) := \begin{bmatrix} y(t) \\ h(y) \end{bmatrix} \qquad \mathbf{u}(t) := [\omega(t)]$$
$$\Rightarrow x_{eq} = x_{goal} = \begin{bmatrix} y_{goal} \\ 0 \end{bmatrix} \text{ and } u_{eq} = [0]$$

Thus the system motion model is:

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} \bar{v}sin(h(t)) \\ \omega(t) \end{bmatrix} := \mathbf{f}(x(t), u(t))$$

#### 1.1.2 Linearization

Lets define $\delta\mathbf{x} = \mathbf{x}(t) - x_{eq}$ and $\delta\mathbf{u} = \mathbf{u}(t) - u_{eq}$. LInearizing the system at $x = x_{eq}$ and $u = u_{eq}$ gives the following approximate system model.

$$\delta\dot{\mathbf{x}} = \frac{\partial\mathbf{f}}{\partial\mathbf{x}}\bigg|_{\mathbf{x}=x_{eq},\mathbf{u}=u_{eq}} \cdot \delta\mathbf{x} \quad + \quad \frac{\partial\mathbf{f}}{\partial\mathbf{u}}\bigg|_{\mathbf{x}=x_{eq},\mathbf{u}=u_{eq}} \cdot \delta\mathbf{u}$$

where,

$$\frac{\partial\mathbf{f}}{\partial\mathbf{x}} = \begin{bmatrix} 0 & \bar{v}cos(h_{eq}) \\ 0 & 0 \end{bmatrix} \qquad \frac{\partial\mathbf{f}}{\partial\mathbf{u}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Plugging in the Quilibrium values and comparing the equation to: $\delta\dot{\mathbf{x}} = \mathbf{A}\delta\mathbf{x} + \mathbf{B}\delta\mathbf{u}$

$$\mathbf{A} = \begin{bmatrix} 0 & \bar{v} \\ 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

### 1.1.2 LQR Implementation

It is given that the control cost function for the above control problem is:

$$J = \frac{1}{2}(\mathbf{x}(T) - x_{goal})^T \mathbf{Q}_t (\mathbf{x}(T) - x_{goal}) + \int_0^T \frac{1}{2}\left((\mathbf{x}(t) - x_{goal})^T \mathbf{Q}_s (\mathbf{x}(t) - x_{goal}) + \mathbf{u}(t)^T \mathbf{R}_s \mathbf{u}(t)\right) dt \qquad (1)$$

The cost fucntion is Quadratic in which implies that the matlab function *lqr* can be used to solve this problem for a linearized dynamics equation. Since the system has been linearized about $x_{eq} = x_{goal}$ and $u_{eq} = [0]$, the cost function can be written as:

$$J = \frac{1}{2}\left(\delta\mathbf{x}(T)^T \mathbf{Q}_t\, \delta\mathbf{x}(T)\right) + \int_0^T \frac{1}{2}\left(\delta\mathbf{x}(t)^T \mathbf{Q}_s\, \delta\mathbf{x}(t) + \delta\mathbf{u}(t)^T \mathbf{R}_s\, \delta\mathbf{u}(t)\right) dt$$

where, $\mathbf{Q}_s = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{R}_s = [20]$ are given

The *lqr* function provides the corresponding optimal state gain ($K^*$), such that the control equation can be written as $\delta\mathbf{u} = -{K^*}^T \delta\mathbf{x}$. Here $\delta\mathbf{x} = \mathbf{x}(t) - x_{eq}$ and $\delta\mathbf{u} = \mathbf{u}(t) - u_{eq}$ can be substituted in to give:

$$\mathbf{u}(t) = \left(u_{eq} + {K^*}^T x_{eq}\right) - {K^*}^T \mathbf{x}(t)$$

$$\mathbf{u}(t) = \begin{bmatrix} u_{eq} + {K^*}^T x_{eq} \\ -{K^*}^T \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \mathbf{x}(t) \end{bmatrix}$$

### 1.1.3 Hyper-parameter Analysis

It is imperitive to compare the effect of the chosen hyperparameters, ie, the values of $\mathbf{Q}_s$ and $\mathbf{R}_s$. It is important to note that, only the ratio of the hyperparameters matter rather than their actual values. Figure 1 below shows the response of the system as we vary the control ratio ($R_s/Q_y$). There is a clear relationship between this ratio and the control effort applied to get to the final state; as the ratio increases, the control effort $\omega$ is reduces and the vehicle reaches the goal state at a later point of time. It is important to note that, at $R_s = 0.1 \cdot Q_y$, the penalization for control effort is too low such that the controller produces a high control effort which leads to a vehicle angle lower than $-\pi$, which is outside the domain of the linearized motion model. This implies that the above designed control ratio has to be carefully tuned to make sure that the linearization is still valid.

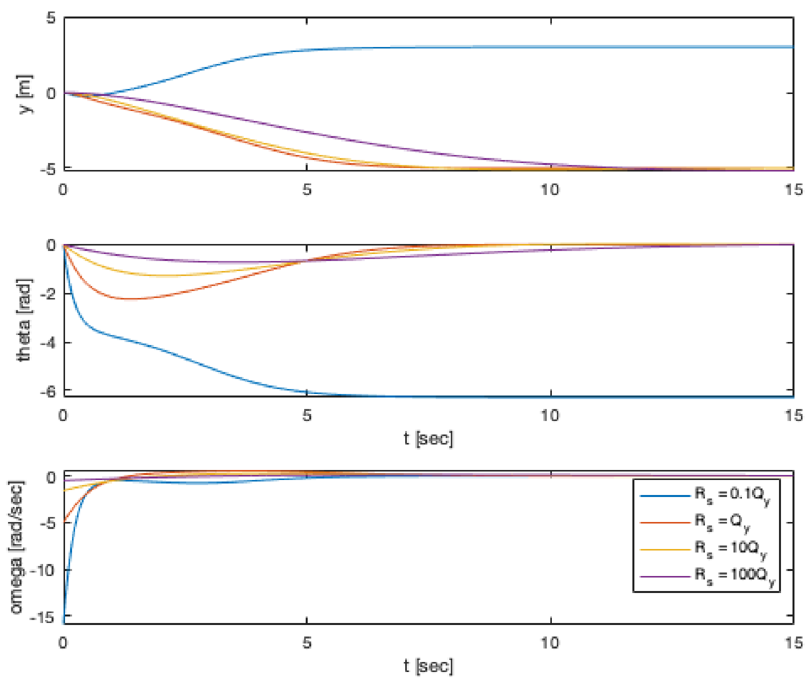


**Figure 1**: Mobile robot trajectory plots with varying ratio of state cost ($\mathbf{Q}_s = diag([Q_y = 1, Q_h = 1])$) and input cost ($R_s$) weights

Figure 2 shows the system response as the heading ratio ($Q_h/Q_y$) is varied. A higher ratio would penalize devaition of heading angles from the equilibrium value of 0 radians, which was consistant with the plots. It is interesting to note that, heading ratio of 1 appears to perform the best.

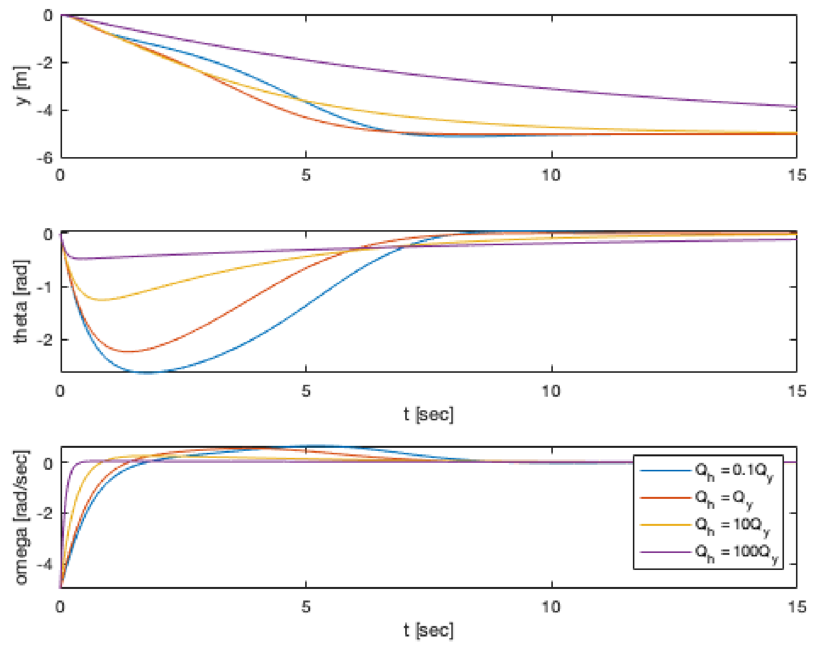


**Figure 2**: ratio of heading cost ($Q_h$) and vehicle height ($Q_y$) weights, while control cost weight ($R_s = 20$) is kep constant

A change in initial state $\mathbf{x}_0$ can have a change in the performance of the system. Figure 3 shows the effect of this change. The LQR controller is run at the default values for $\mathbf{Q}_s$ and $R_s$, while $y_{start}$ is varied from 0 - 9 meters. It can be seen that, the controllers produces an unstable trajectory at $y_{goal} = 9m$ due to a similar reason as before: vehicle heading exceeding $-\pi$.

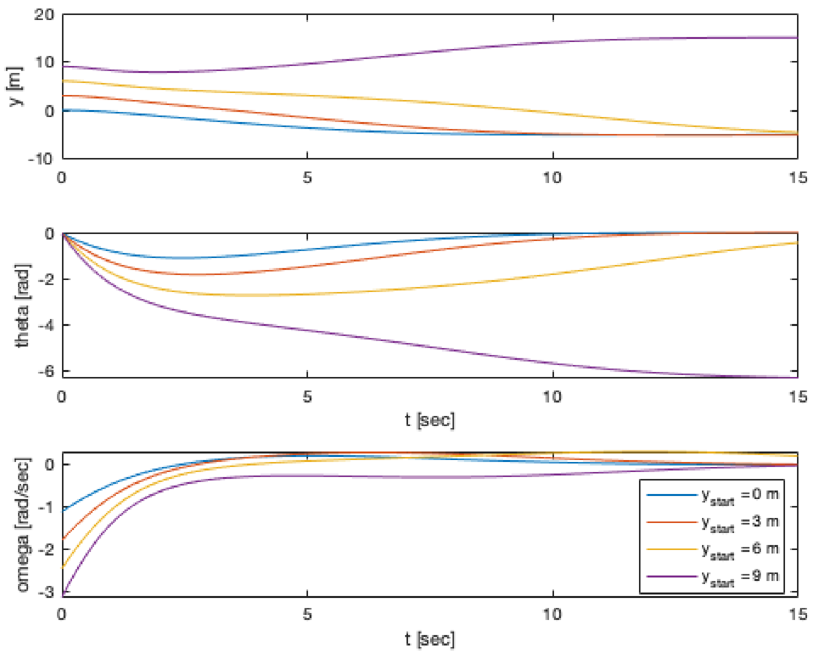


**Figure 3:** Mobile robot trajectory plots with varying initial position ($\mathbf{x}_0 = [y_{start}, 0]$)

## 1.2 Iterative Linear Quadratic Controller

### 1.2.1 Linearization

The system is descried by the following equation where $\delta t$ is the time step.

$$\mathbf{x}_{k+1} = \begin{bmatrix} y_{k+1} \\ h_{k+1} \end{bmatrix} = \begin{bmatrix} y_k \\ h_k \end{bmatrix} + \delta t \begin{bmatrix} \bar{v} sin(h_k) \\ \omega_k \end{bmatrix}$$

Linearizing this system, about an operating point $\bar{\mathbf{x}}_k$ and $\bar{\mathbf{u}}_k$, requires the definition of state and input perturbation: $\delta \mathbf{x}_k = \mathbf{x}_k - \bar{\mathbf{x}}_k$ and $\delta \mathbf{u}_k = \mathbf{u}_k - \bar{\mathbf{u}}_k$. Thus, the linearized kinematics model of the state error is shown below.

$$\delta \mathbf{x}_{k+1} = \mathbf{A}_k \delta \mathbf{x}_k + \mathbf{B}_k \delta \mathbf{u}_k \qquad (2)$$

where,

$$\mathbf{A}_k = \begin{bmatrix} 1 & \bar{v}\delta t\, cos(\bar{h}_k) \\ 0 & 1 \end{bmatrix} \quad \mathbf{B}_k = \begin{bmatrix} 0 \\ \delta t \end{bmatrix}$$

### 1.2.2 Cost Function

The cost given in Equation 1 also needs to be discretized as follows:

$$J = g_N(\mathbf{x}_N) + \delta t \sum_{0}^{N-1} g_k(\mathbf{x}_k, \mathbf{u}_k)$$

where,

$$g_N(\mathbf{x}_N) = \frac{1}{2}(\mathbf{x}_N - x_{goal,N})^T \mathbf{Q}_t (\mathbf{x}_N - x_{goal,N})$$

$$g_k(\mathbf{x}_k, \mathbf{u}_k) = \frac{1}{2}\left((\mathbf{x}_k - x_{goal,k})^T \mathbf{Q}_s (\mathbf{x}_k - x_{goal,k}) + \mathbf{u}_k^T \mathbf{R}_s \mathbf{u}_k\right)$$

Expressing the states and inputs in terms of the operating point and perturbation; $\mathbf{x}_k = \bar{\mathbf{x}}_k + \delta\mathbf{x}_k$ and $\mathbf{u}_k = \bar{\mathbf{u}}_k + \delta\mathbf{u}_k$, and plugging this into $g_N(\mathbf{x}_N)$ and $g_k(\mathbf{x}_k, \mathbf{u}_k)$, the above mentioned parts of cost function can be written as:

$$g_N(\mathbf{x}_N) = q_N + \mathbf{q}_N^T \delta\mathbf{x}_N + \frac{1}{2}\delta\mathbf{x}_N^T \mathbf{Q}_N \delta\mathbf{x}_N$$

$$g_k(\mathbf{x}_k, \mathbf{u}_k) = q_k + \mathbf{q}_k^T \delta\mathbf{x}_k + \frac{1}{2}\delta\mathbf{x}_k^T \mathbf{Q}_k \delta\mathbf{x}_k + \mathbf{r}_k^T \delta\mathbf{u}_k + \frac{1}{2}\delta\mathbf{u}_k^T \mathbf{R}_k \delta\mathbf{u}_k + \frac{1}{2}\delta\mathbf{u}_k^T \mathbf{P}_k \delta\mathbf{x}_k \tag{3}$$

where,

$$q_N = \frac{1}{2}(\bar{\mathbf{x}}_N - x_{goal,N})^T \mathbf{Q}_t (\bar{\mathbf{x}}_N - x_{goal,N}), \quad \mathbf{q}_N = \frac{1}{2}\left(\mathbf{Q}_t + \mathbf{Q}_t^T\right)(\bar{\mathbf{x}}_N - x_{goal,N}), \quad \mathbf{Q}_N = \frac{1}{2}\left(\mathbf{Q}_t + \mathbf{Q}_t^T\right),$$

$$q_k = \frac{1}{2}\left((\bar{\mathbf{x}}_k - x_{goal,k})^T \mathbf{Q}_s (\bar{\mathbf{x}}_k - x_{goal,k}) + \bar{\mathbf{u}}_k^T \mathbf{R}_s \bar{\mathbf{u}}_k\right), \quad \mathbf{q}_k = \frac{1}{2}\left(\mathbf{Q}_s + \mathbf{Q}_s^T\right)(\bar{\mathbf{x}}_k - x_{goal,k}), \quad \mathbf{Q}_k = \frac{1}{2}\left(\mathbf{Q}_s + \mathbf{Q}_s^T\right),$$

$$\mathbf{r}_k = \frac{1}{2}\left(\mathbf{R}_s + \mathbf{R}_s^T\right)\bar{\mathbf{u}}_k, \quad \mathbf{R}_k = \frac{1}{2}\left(\mathbf{R}_s + \mathbf{R}_s^T\right), \quad \mathbf{P}_k = \mathbf{0}$$

Note the the cost shown here is the exact discretized cost and not an approximation.

### 1.2.3 Input Perturbation for Optimal Policy

According to the bellman equation, the optimal policy must follow the equation:

$$V^*(k, \mathbf{x}_k) = \min_{\mathbf{u}_k} \left(g_k(\mathbf{x}_k, \mathbf{u}_k) + V^*(k+1, \mathbf{x}_{k+1})\right) \tag{4}$$

We can approximate the optimal value function $V^*$ as a second order linearization about the linearization point $\bar{\mathbf{x}}_{k+1}$ for the $(k+1)^{th}$ time-step as follows:

$$V^*(k+1, \mathbf{x}_{k+1}) = s_{k+1} + \delta\mathbf{x}_{k+1}^T \mathbf{s}_{k+1} + \frac{1}{2}\delta\mathbf{x}_{k+1}^T \mathbf{S}_{k+1} \delta\mathbf{x}_{k+1} \tag{5}$$

Plugging in Equation 2, 3, 5 into Equation 4, we get the RHS of Equation 4 to be a minimization with respect to $\mathbf{u}_k$:

$$V^*(k, \mathbf{x}_k) = \min_{\mathbf{u}_k} \Big( q_k + \mathbf{q}_k^T \delta\mathbf{x}_k + \frac{1}{2}\delta\mathbf{x}_k^T \mathbf{Q}_k \delta\mathbf{x}_k + \mathbf{r}_k^T \delta\mathbf{u}_k + \frac{1}{2}\delta\mathbf{u}_k^T \mathbf{R}_k \delta\mathbf{u}_k + \\ s_{k+1} + (\mathbf{A}_k\delta\mathbf{x}_k + \mathbf{B}_k\delta\mathbf{u}_k)^T \mathbf{s}_{k+1} + \frac{1}{2}\delta(\mathbf{A}_k\delta\mathbf{x}_k + \mathbf{B}_k\delta\mathbf{u}_k)^T \mathbf{S}_{k+1}(\mathbf{A}_k\delta\mathbf{x}_k + \mathbf{B}_k\delta\mathbf{u}_k) \Big) \tag{6}$$

The given optimization can be solved by setting the derivative of RHS to 0 and computing the optimal perturbation in the input at the $k^{th}$ time-step:

$$\delta\mathbf{u}_k^* = \delta\mathbf{u}_k^{ff} + \mathbf{K}_k\delta\mathbf{x}_k \tag{7}$$

where,

$$\delta\mathbf{u}_k^{ff} = -\left(\mathbf{R}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{B}_k\right)^{-1}\left(\mathbf{r}_k + \mathbf{B}_k^T\mathbf{s}_{k+1}\right)$$

$$\mathbf{K}_k = -\left(\mathbf{R}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{B}_k\right)^{-1}\left(\mathbf{P}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{A}_k\right)$$

Plugging in $\mathbf{x}_k = \bar{\mathbf{x}}_k + \delta\mathbf{x}_k$ and $\mathbf{u}_k = \bar{\mathbf{u}}_k + \delta\mathbf{u}_k$, we get

$$\mathbf{u}_k = \boldsymbol{\theta}_{k,ff} + \boldsymbol{\theta}_{k,bf}\mathbf{x}_k \tag{8}$$

where,

$$\boldsymbol{\theta}_{k,ff} = \bar{\mathbf{u}}_k + \delta\mathbf{u}_k^{ff} - \mathbf{K}_k\bar{\mathbf{x}}_k$$

$$\boldsymbol{\theta}_{k,bf} = \mathbf{K}_k$$

Note that the value of $s_{k+1}$, $\mathbf{s}_{k+1}$ and $\mathbf{S}_{k+1}$ are still unknown. In order to find their values, $\delta\mathbf{u}_k^*$ from equation 7 is subsituted back into equation 6 and $V^*(k, \mathbf{x}_k)$ is replaced with in approximate value. This step completely eliminates $\delta\mathbf{u}_k$ from the equation and leaves behind terms that are only dependent on $\delta\mathbf{x}_k$. A simple comparison of constants and terms containing $\delta\mathbf{x}_k$, given us the following propogation relation between $s_{k+1}$, $\mathbf{s}_{k+1}$, $\mathbf{S}_{k+1}$ and $s_k$, $\mathbf{s}_k$ and $\mathbf{S}_k$.

$$\mathbf{S}_k = \mathbf{Q}_k + \mathbf{A}_k^T\mathbf{S}_{k+1}\mathbf{A}_k + \mathbf{K}_k^T(\mathbf{R}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{B}_k)\mathbf{K}_k + \mathbf{K}_k^T(\mathbf{P}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{A}_k) + (\mathbf{P}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{A}_k)^T\mathbf{K}_k$$

$$\mathbf{s}_k = \mathbf{q}_k + \mathbf{A}_k^T\mathbf{s}_{k+1} + \mathbf{K}_k^T(\mathbf{R}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{B}_k)\delta\mathbf{u}_k^{ff} + \mathbf{K}_k^T(\mathbf{P}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{A}_k) + (\mathbf{r}_k + \mathbf{B}_k^T\mathbf{s}_{k+1})^T\mathbf{u}_k^{ff}$$

$$s_k = q_k + s_{k+1} + \frac{1}{2}{\mathbf{u}_k^{ff}}^T(\mathbf{R}_k + \mathbf{B}_k^T\mathbf{S}_{k+1}\mathbf{B}_k)\mathbf{u}_k^{ff} + {\mathbf{u}_k^{ff}}^T(\mathbf{r}_k + \mathbf{B}_k^T\mathbf{s}_{k+1})$$

Matlab fucntions that implement the above algorithm are given in Appendix A. Below is the pseudocode for the iterative linear quadratic controller:

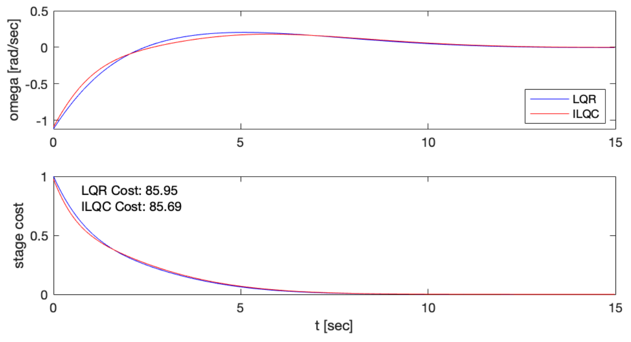
```
controller_lqr = run_LQR_control
controller_ilqc = controller_lqr

while iteration < max_iteration, loop
```

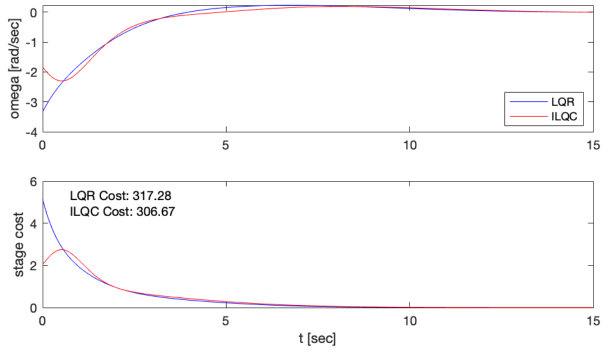
```
        prev_control = controller_ilqc

        output = run_simulation(controller_ilqc)
        x = output.x
        u = output.u
        teminal_x = x(end)

        [q(N), q_v(N), Q(N)] = terminal_cost_quad(Q_terminal, goal_x, terminal_x)
        s(N) = q_N
        s_v(N) = q_v_N
        S(N) = Q_N

        for k = N-1 to 1
            [ Ak, Bk ] = mobile_robot_lin(x(k), u(k), dt, v_constant)
            [ q(k), q_v(k), Q(k), r(k), R(k), P(k) ] = stage_cost_quad( Q_s, R_s, goal_x,
                                                                        dt, x(k), u(k))
            [U_ff, U_fb, s(k), s_v(k), S(k)] = update_policy(Ak, Bk,
                                                             q(k), q_v(k), Q(k),
                                                             r(k), R(k), P(k),
                                                             s(k+1), s_v(k+1), S(k+!),
                                                             x(k), u(k))
            controller_ilqc(k) = [U_ff; U_fb];
        end

        if norm(control_ilqc - prev_control) < threshold
            break while loop
        end

    end
```

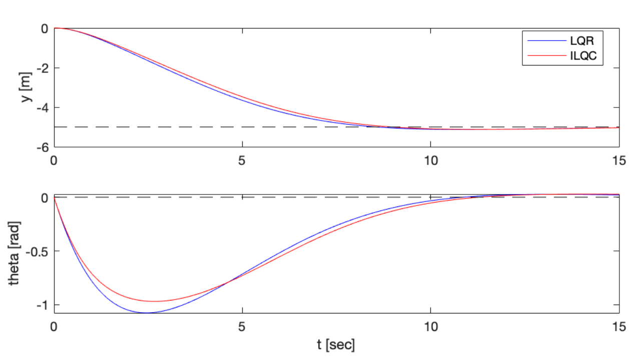
## 1.3 Comparing LQR and ILQC

Figure 4 shows a comparison between the Linear Quadratic Regulator (LQR) and Iterative Linear Quadratic Controller (ILQC) for two different starting positions. The response of the LQR and the ILQC controllers are very similar for the case where the starting pose is the origin. The is not true for a starting pose of $[0, -\pi]$ where the ILQC has managed to use lesser control effort to produce a very similar control trajectory.

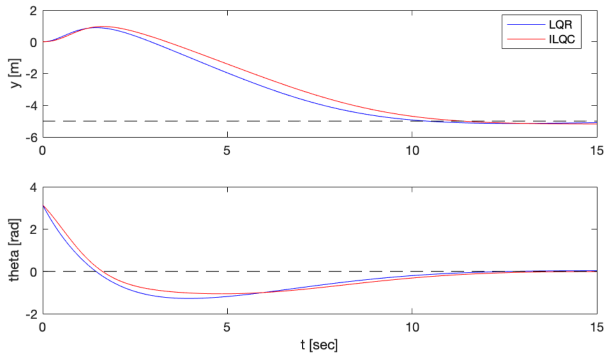
If there was a mismatch between the model and the real system, the LQR is expected to diverge quickly owing to the compounding linearization errors. The ILQC controller, on the other hand, would be able to tolerate small error in the model and the real system but would diverge if errors are high and time-steps are large.







(a)



(b)

**Figure 4**: Plots showing a comparison between LQR and ILQC methodologies in terms of state, cost and inputs for an initial state of (a) [0, 0] and (b) [0,$\pi$]

## 2.0 Quadrotor Control

### 2.1 LQR Implementation

In order to use an LQR controller for a quadrotor, the states and inputs of the quadrotor have to be augmented such that the goal state lies at the origin ($\mathbf{x}_N = \mathbf{0}$). Secondly, the kinematics model of the vehicle has to be linear while the cost function has to be quadratic. Thus, the kinematics model is chosen to be linearized about the goal state ($\bar{\mathbf{x}} = \mathbf{x}_{goal}$) and the corresponding input to be an equilibruim input ($\bar{\mathbf{u}} = \mathbf{u}_{eq}$), which implies that the goal of the quadrotor is to be stationary (hovering) at the goal state. Figure 5, shows the resultant trajectory quadrotor. It can seen that the quadrotor descended while moving towards the goal and regained the height after getting sufficiently close to the goal in the horizontal plane.

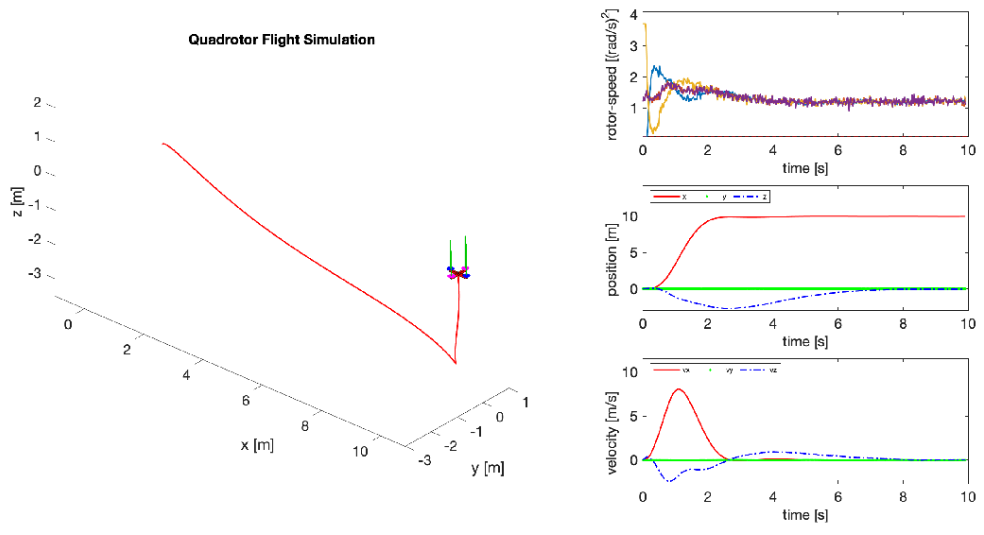


**Figure 5**: Resultant quadrotor trajectory using an LQR controller with plots of the corresponding rotor speed, position and velocity vs time

After some experimentation, it is found that the LQR controller has limitations with regards to the distant between the goal and the start state. Empercically, it is found out that the LQR controller diverges (doesn't reach a feasible solution) if the goal state is 13 or more meters away from the start position. Figure 6 shows an example of an unstable LQR controller policy. This instability of the policy can be attributed to the compounding error due to a linear kinematics assumption. The linearized motion model of the quadrotor is thus not a good enough approximation when the difference between the linearization state and current state is large enough.

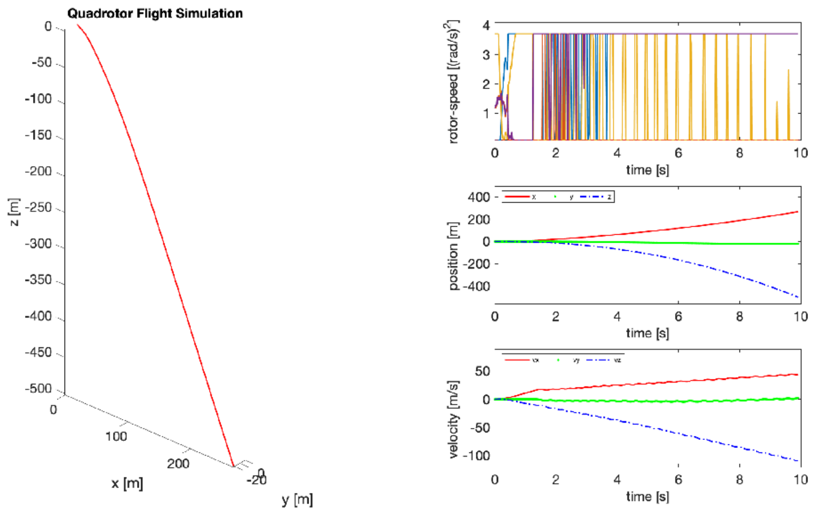


**Figure 6**: Resultant trajectory of the quadrotor with $\mathbf{x}_{goal} = [13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$

### 2.1.1 LQR with Via-point

One way to extend the range of the LQR controller is to use via-points in the trajectory of the quadrotor. This implies the goal state of the quadrotor, which is the linearization state of the kinematics model, changes with time. The duration for which the quadrotor is moving from via-point $i-1$ to $i$, the state at via-point $i$ is treated as the goa state in the LQR frame ($\bar{\mathbf{x}} = \mathbf{x}_{goal} = \mathbf{x}_{vp,1}$). Figure 7 shows the result of using one via-point in the same quadrotor problem formulation as before. The varying performence lies in the implementation details where the kinematic model of the quadrotor is first linearized at the via-point and used till $t_{vp}$. Subsequently, the model linearized at the goal pose is used for designing the controller. The resultant trajectory uses less control effort but reaches the goal state later than the previous case.

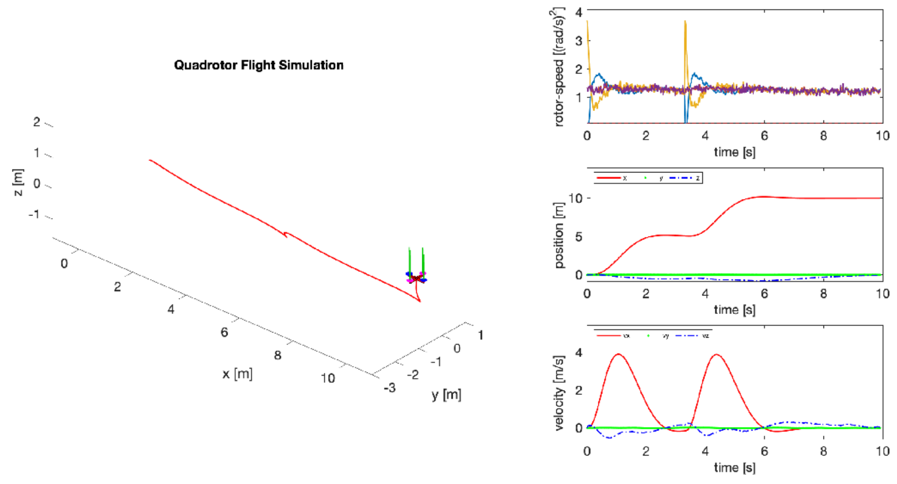


**Figure 7**: Quadrotor trajectory produced by an LQR controller with the $\mathbf{x}_{goal} = [10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$, $\mathbf{x}_{vp,1} = [5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$ and $t_{vp,1} = 3.33s$

Through some emperical experimentation it is confirmed that adding an intermediate via-point extends the stable range of the quadrotor to upto 17 m away from the start state. Figure 8 shows the response of the system when an LQR controller is used with an intermediate via-point. The 17 meters range is undoubtedly more than the 12 meters range with the via-point. From the previous section, it is clear that if the difference between the model linearization pose and the start pose is more than a certain threshold, the linear assumption for the kinematics model would be deemed invalid and the system would diverege. When a via-point is added in the trajectory, it reduces the maximum distance a linearization pose and the vehicle pose, thus entending the stablity range of LQR control framework. This implies that using additional via-points in the trajectory of the quadrotor would improve the stability of the controller and, at the same time, extend its stability range. But this also means that it would take the quadrotor much more time to reach the goal pose, and the net control effort would be much higher due to the frequent acceleration and deceleration of the quadrotor.

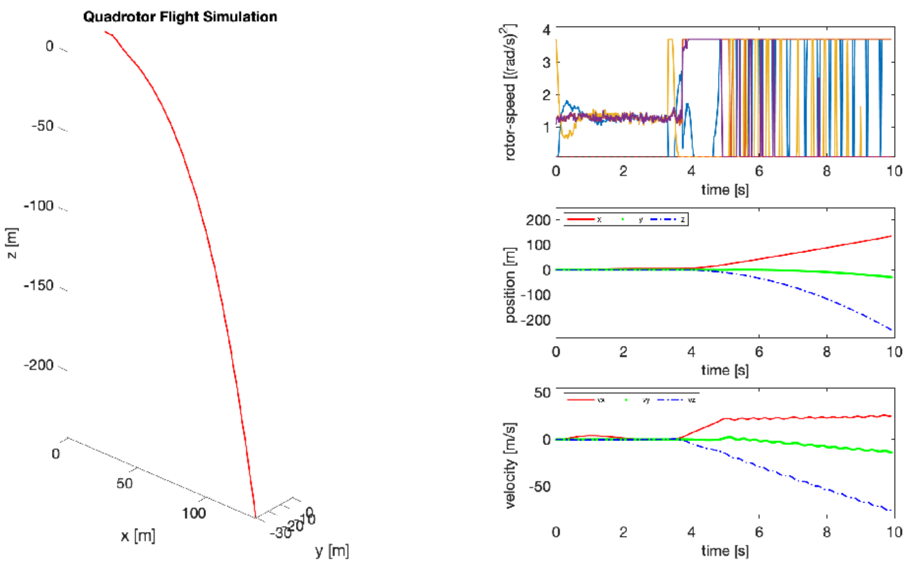


**Figure 8**: Quadrotor trajectory produced by an LQR controller with the $\mathbf{x}_{goal} = [18,0,0,0,0,0,0,0,0,0,0,0]^T$, $\mathbf{x}_{vp,1} = [5,0,0,0,0,0,0,0,0,0,0,0]^T$ and $t_{vp,1} = 3.33s$

## 2.2 ILQC Results and Comparison

The ILQC controller has been implemented according to the guidelines provided in the assignment document and Figure 9 shows the resultant trajectory. Compared to the LQR controller trajectory in figure 5, the ILQC controller trajectory is clearly better since the quadrotor does not have to do a steep climb close to the goal coordinates. Moreover, the trajectory is more smooth and the maneuvers are more streamlined. The final cost of the two controller relay the same precedence such that there is $\sim$40% reduction in the cost when switching from LQR to ILQC. Thus, there is a significant visible improvement in the resultant trajectory as well as the controller cost metric.

It is important to note that the ILQC implementation involves a kinematic model linearization at every time step, contrary to the LQR imlplementation, which uses the same linearized model throughout. There are clues from the via-point result of from the previous that explain this improvement. By making every trajectory point a via-point, and running the optimization iteratively through the output trajectories, the ILQC significantly reduces the effect of linearization errors and produces a highly optimal open-loop trajectory. This means the the ILQC would be able to work well regardless of the difference between the start pose and the goal pose.

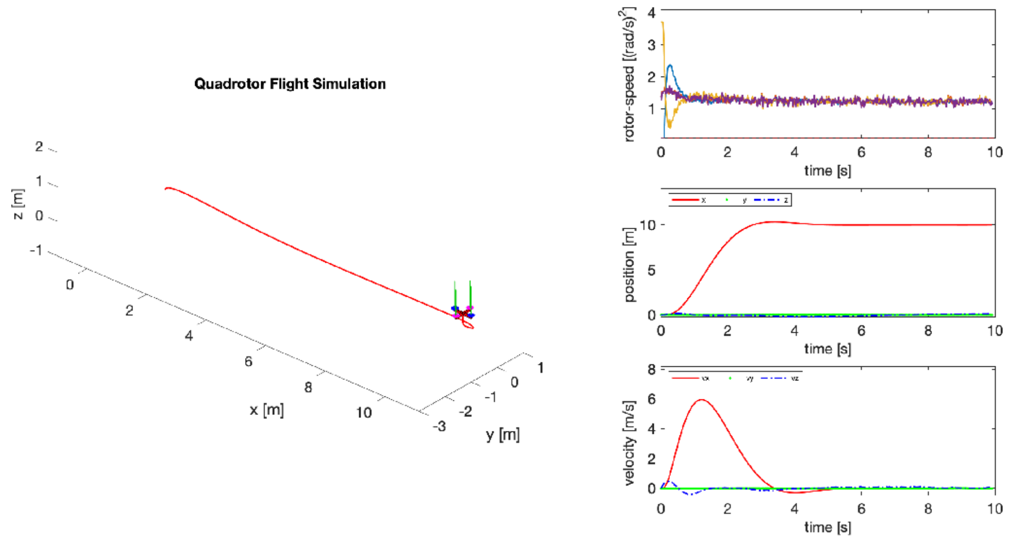


**Figure 9**: Quadrotor trajectory produced by an ILQC controller with the $\mathbf{x}_{goal} = [10,0,0,0,0,0,0,0,0,0,0,0]^T$

### 2.2.1 ILQC with Via-point

The via-point in an ILQC framework is implemented using a time dependent cost function added to the intermediate state cost. The cost function is shown in the equation below:

$$g_{vp}(t) = (\mathbf{x}-\mathbf{x}_{vp})^T\mathbf{Q}_{vp}(\mathbf{x}-\mathbf{x}_{vp})\sqrt{\frac{\rho}{2\pi}}e^{-\frac{\rho}{2}(t-t_{vp})^2} \tag{9}$$

It would be beneficial to have the elements corresponding to the non-positional state variables in the via-point weighting matrix ($\mathbf{Q}_{vp}$) to zero. This basically means that only the positional error close to the time $t_{vp}$ should penalized while the optimization should have the liberty to choose the other state varaibles. Moreover, the elements corresponding to the positional state varaibales in the intermediate state weighting matrix ($\mathbf{Q}_k$) should be set to zero and only let the terminal positional cost drive the quadrotor towards the final goal. Figure 10 shows the resultant plot for using a simple intermediate via-point while Figure 11 shows the resultant plot for an out-of-the-way via-point.

The current implementation treats the via-point as a relatively soft constraint since it is not garunteed that the quadrotor will pass through the via-point, as shown in figure 11. If the intent is to garuntee that the quadrotor passes through the via-point, it can be implemented by using a timed cost function which switches goal state from via-point to the actual goal state after time $t_{vp}$. Another way would be to increase the values $\mathbf{Q}_{vp}$ to be multiple orders to magnitude greater than $\mathbf{Q}_m$ in order to force the vehicle to reach the via-point with a tighter bound.

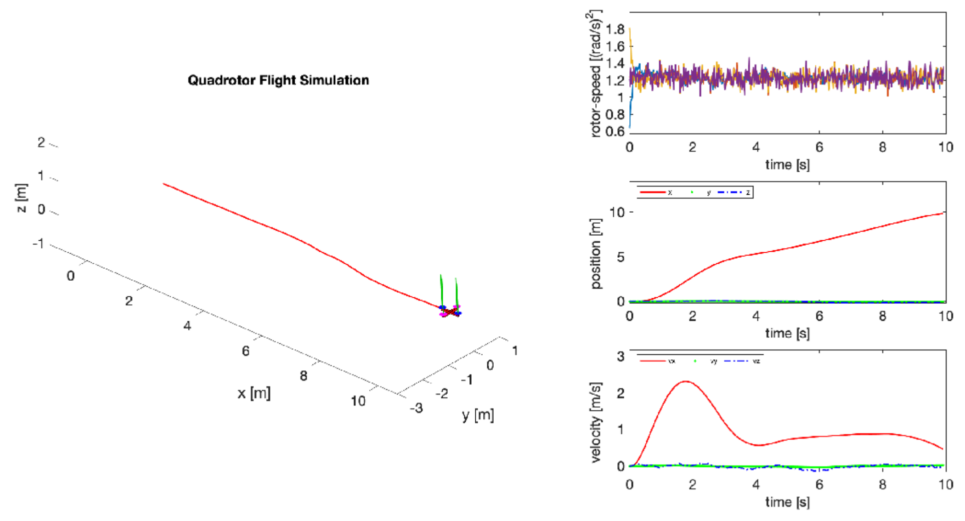


**Figure 10**: Quadrotor trajectory produced by an ILQC controller with the $\mathbf{x}_{goal} = [10,0,0,0,0,0,0,0,0,0,0,0]^T$, $\mathbf{x}_{vp,1} = [5,0,0,0,0,0,0,0,0,0,0,0]^T$ and $t_{vp,1} = 3.33s$

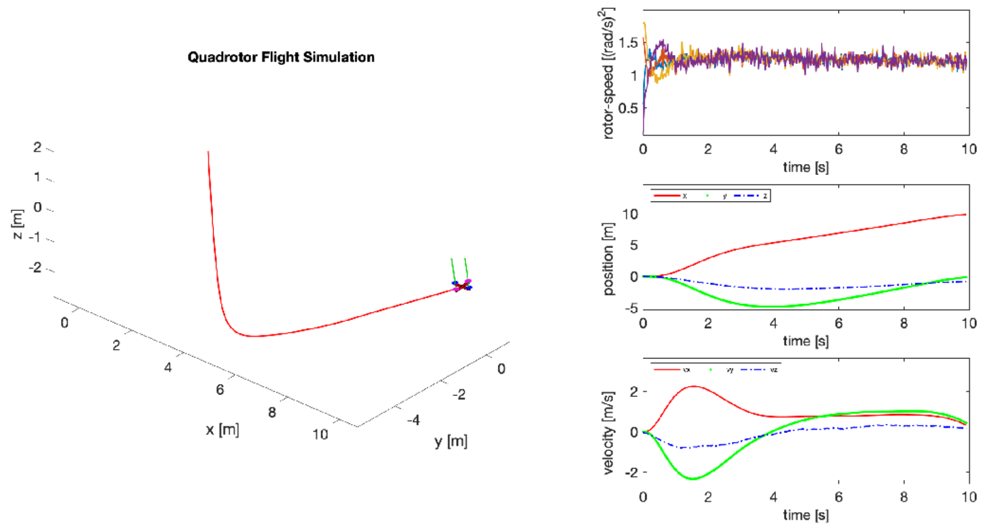


**Figure 11**: Quadrotor trajectory produced by an ILQC controller with the $\mathbf{x}_{goal} = [10,0,0,0,0,0,0,0,0,0,0,0]^T$, $\mathbf{x}_{vp,2} = [5,-5,-5,0,0,0,0,0,0,0,0,0]^T$ and $t_{vp,1} = 3.33s$

## Appendix A

### Solution code

MAIN_P1_LQR

```
%% 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 1.1 (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()'.
%
%
% =========================================================================

% task_lqr.start_x = [0, pi]';

ratio = [0.1, 1, 10, 100];
% Mode 0 is Default
% Mode 1 is compare R vs Q_y
% Mode 2 is compare Q_h vs Q_y
% Mode 3 is Check effects of X_0
mode = 0;
Q = task_lqr.cost.params.Q_s;
R = task_lqr.cost.params.R_s;

if mode == 0
    ratio = [1];
end

for i = 1:length(ratio)
    y_target = 1;
    A = [ 0, const_vel;
          0, 0];
    B = [ 0;
          1];

    if mode == 0
        % Do nothing
    elseif mode == 1
        R = ratio(i)*Q(1,1);
    elseif mode == 2
        Q = diag([1,ratio(i)]);
        R = [1];
    elseif mode == 3
        task_lqr.start_x(1) = 3*(1 + log10(ratio(i)));
    end

    [K, S, E] = lqr(A, B, Q, R);

    theta_ff = 0 + K*task_lqr.goal_x;
    theta_fb = -K';
    for j = 1:N-1
        controller_lqr(1:3, j) = [theta_ff; theta_fb];
    end

    %% 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); hold on;
    end
end
if mode == 1
    legend( 'R_s = 0.1Q_{y}', 'R_s = Q_{y}', 'R_s = 10Q_{y}', 'R_s = 100Q_{y}' )
elseif mode == 2
    legend( 'Q_{h} = 0.1Q_{y}', 'Q_{h} = Q_{y}', 'Q_{h} = 10Q_{y}', 'Q_{h} = 100Q_{y}' )
elseif mode == 3
    legend( "y_{start} = " + int2str(3*(1 + log10(ratio(1)))) + " m",...
            "y_{start} = " + int2str(3*(1 + log10(ratio(2)))) + " m",...
            "y_{start} = " + int2str(3*(1 + log10(ratio(3)))) + " m",...
            "y_{start} = " + int2str(3*(1 + log10(ratio(4)))) + " m" );
end
hold off;

%% 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
```

## MOBILE_ROBOT_LIN

```
function [A_k, B_k] = mobile_robot_lin(x_lin, u_lin, dt, v_bar)

    add_error = false;

    A_k = [];
    B_k = [];
    if dt <= 0
       disp('mobile_robot_lin: Invalid dt')
       return
    end
    A_k = [ 1, dt*v_bar*cos(x_lin(2));
            0 , 1];
    B_k = [ 0; dt];

    if add_error
        A_k = A_k * (1 + 0.2);
        B_k = B_k * (1 + 0.2);
    end
end
```

## TERMINAL_COST_QUAD

```
function [q_N, q_v_N, Q_N] = terminal_cost_quad(Q_t, x_goal, x_lin)

    x = x_lin - x_goal;
    x = reshape(x, numel(x), []);

    q_N   = 1/2 * x' * Q_t * x;
    q_v_N = 1/2 * (Q_t' + Q_t) * x;
    Q_N   = 1/2 * (Q_t' + Q_t);
end
```

## STAGE_COST_QUAD

```
function [ q_k, q_v_k, Q_k, r_k, R_k, P_k ] = stage_cost_quad(Q_s, R_s, x_goal, dt, x_lin, u_lin)

    if dt <= 0
       disp('stage_cost_quad: Invalid dt')
       return
    end

    x = x_lin - x_goal;
    x = reshape(x, numel(x), []);
    u = u_lin;
    u = reshape(u, numel(u), []);

    q_k   = 1/2 * x' * Q_s * x + 1/2 * u' + R_s * u * dt;
    q_v_k = 1/2 * (Q_s' + Q_s) * x * dt;
    Q_k   = 1/2 * (Q_s' + Q_s) * dt;

    r_k = 1/2 * (R_s' + R_s) * u * dt;
    R_k = 1/2 * (R_s' + R_s) * dt;

    P_k = zeros(numel(u), numel(x)) * dt;

end
```

## UPDATE_POLICY

```
function [U_ff, U_fb, s_k, s_v_k, S_k] = update_policy(A_k, B_k, ...
                                                      q_k, q_v_k, Q_k, ...
                                                      r_k, R_k, P_k, ...
                                                      s_kp, s_v_kp, S_kp, ...
                                                      x_lin, u_lin)

    x_lin = reshape(x_lin, numel(x_lin), []);
    u_lin = reshape(u_lin, numel(u_lin), []);

    g_k = r_k + B_k' * s_v_kp;
    G_k = P_k + B_k' * S_kp * A_k;
    H_k = R_k + B_k' * S_kp * B_k;

    H_k = 1/2*(H_k + H_k');

    u_ff = -inv(H_k) * (g_k);
    K_k  = -inv(H_k) * (G_k);

    U_ff = u_lin + u_ff - K_k * x_lin;
    U_fb = K_k';
    S_k   = Q_k + A_k' * S_kp * A_k + K_k' * H_k * K_k + ...
                                      K_k' * G_k + G_k' * K_k;
    s_v_k = q_v_k + A_k' * s_v_kp + K_k' * H_k * u_ff + ...
                                      K_k' * g_k + G_k' * u_ff;
    s_k   = q_k + s_kp + 1/2 * u_ff' * H_k * u_ff + u_ff' * g_k;

end
```

## MAIN_P1_ILQC

```
%% 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 1.1 (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()'.
%
%
% =========================================================================

Q_s = task_ilqc.cost.params.Q_s;
R_s = task_ilqc.cost.params.R_s;
Q_t = task_ilqc.cost.params.Q_t;

controller_ilqc_last = zeros(size(controller_ilqc));
iter = 0;

% task_ilqc.start_x = [0, pi]';

while norm(controller_ilqc - controller_ilqc_last, 2) > 10^-5  && iter < task_ilqc.max_iteration
    % Something
    controller_ilqc_last = controller_ilqc;
    iter = iter+1;

    % Init policy as controller_ilqc(:, k) = 0 for all k
    lin_out = mobile_robot_sim(model, task_ilqc, controller_ilqc);
    x_lin = lin_out.x;
    u_lin = lin_out.u;

    [q_N, q_v_N, Q_N] = terminal_cost_quad(Q_t, task_ilqc.goal_x, x_lin(:, end));
    s_kp = q_N; s_v_kp = q_v_N; S_kp = Q_N;
    for k = N-1:-1:1
        [A_k, B_k] = mobile_robot_lin(x_lin(:, k), u_lin(:, k), ...
                                           task_ilqc.dt, const_vel);
        [ q_k, q_v_k, Q_k, r_k, R_k, P_k ] = ...
                        stage_cost_quad( Q_s, R_s, ...
                                         task_ilqc.goal_x, task_ilqc.dt, ...
                                         x_lin(:, k), u_lin(:, k) );
        [U_ff, U_fb, s_k, s_v_k, S_k] = update_policy(A_k, B_k, ...
                                                    q_k, q_v_k, Q_k, ...
                                                    r_k, R_k, P_k, ...
                                                    s_kp, s_v_kp, S_kp, ...
                                                    x_lin(:, k), u_lin(:, k));
        controller_ilqc(:, k) = [U_ff; U_fb];
        S_kp   = S_k;
        s_v_kp = s_v_k;
        s_kp   = s_k;
    end
end


%% 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
```

## LQR_DESIGN

```
% LQR_Design: Implementation of the LQR controller.
%
% Control for Robotics
% AER1517 Spring 2020
% Programming Exercise 1
%
% --
% 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
%
```

```
% 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 1.2 (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 = Task.goal_x;
u_lin = Task.cost.u_eq;


% [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 = 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, temp, temp_] = lqr(A_lin, B_lin, Task.cost.Q_lqr,  Task.cost.R_lqr);

% ========================================================================

%% 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

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 1.2 (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)
        % =================================================================

        theta = [ Task.cost.u_eq + K * Task.goal_x , -K]';

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

    case 'via_point'
%% [Problem 1.2 (c)] Drive system to Task.goal_x through via-point p1.
        t1 = Task.vp_time;
        p1 = Task.vp1; % steer towards this input until t1

        x_lin_wp = p1;
        u_lin_wp = Task.cost.u_eq;

        A_lin_wp = Model.Alin{1}(x_lin_wp, u_lin_wp, Model.param.syspar_vec);
        B_lin_wp = Model.Blin{1}(x_lin_wp, u_lin_wp, Model.param.syspar_vec);

        [K_wp, temp, temp_] = lqr(A_lin_wp, B_lin_wp, Task.cost.Q_lqr,  Task.cost.R_lqr);

        for t=1:Nt-1
           % ==============================================================
           % [Todo] x_ref at current time
           % ...
           %
           % [Todo] time-varying theta matrices for every time step Nt
           % (see handout Eqn.(12)) (size: 13 x 4)
           % theta_k = ...;
           % ==============================================================

            if t * Task.dt <= t1
                theta_k = [ u_lin_wp + K_wp * x_lin_wp , -K_wp]';
            else
                theta_k = [ u_lin + K * x_lin , -K]';
            end

           % 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
```

## ILQC_DESIGN

```
% ILQC_Design: Implementation of the ILQC controller.
%
% Control for Robotics
% AER1517 Spring 2020
% Programming Exercise 1
%
% --
% 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
%
% 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 notes Sec. 1.6 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 1.2 (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
    % ==================================================================
    % Rollout states and inputs
    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
 % ==================================================================
    % 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;
    U0 = sim_out.u; % u_bar
    T0 = sim_out.t; % t
    % ==================================================================

    % ==================================================================
    % Initialize the value function elements starting at final time
    % step (Problem 1.1 (g))

    xf = Task.goal_x; % final state when using current controller
    s(N) = qf_fun(xf);
    Sv(:,N) = Qvf_fun(xf);
    Sm(:,:,N) = Qmf_fun(xf);
    % ==================================================================

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

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

        % ==============================================================
        % Discretize and linearize continuous system dynamics Alin
        % around specific pair (xO,uO). See exercise sheet Eqn. (18) for
        % details.
        Alin = Model_Alin(x0, u0, Model_Param);
        Blin = Model_Blin(x0, u0, Model_Param);
        A = eye(size(Alin)) + Alin * Task.dt;
        B = Blin * Task.dt;
        % ==============================================================


        % ==============================================================
        % Quadratize cost function
        % [Note] use function {q_fun, Qv_fun, Qm_fun, Rv_fun, Rm_fun,
        % Pm_fun} provided above.
        %
        t0 = T0(:,k);
        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);
        % =================================================================

        % =================================================================
        % control dependent terms of cost function (Problem 1.1 (f))

        g = Rv + B' * Sv(:,k+1);       % linear control dependent
        G = Pm + B' * Sm(:,:,k+1) * A; % control and state dependent
        H = Rm + B' * Sm(:,:,k+1) * B; % quadratic control dependent

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

        % =================================================================
        % the optimal change of the input trajectory du = duff +
        % K*dx (Problem 1.1 (f))

        duff(:,:,k) = -H \ g;
        K(:,:,k) = -H \ G;
        % =================================================================

        % =================================================================
        % Solve Riccati-like equations for current time step n
        % (Problem 1.1 (g))

        Sm(:,:,k) = Qm + A' * Sm(:,:,k+1) * A + K(:,:,k)' * H * K(:,:,k) + ...
                                         K(:,:,k)' * G + G' * K(:,:,k);
        Sv(:,k) = Qv + A' * Sv(:,k+1) + K(:,:,k)' * H * duff(:,:,k) + ...
                                         K(:,:,k)' * g + G' * duff(:,:,k);
        s(k) = q + s(k+1) + 1/2 * duff(:,:,k)' * H * duff(:,:,k) + 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);

% ============================================================================
% 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
```

## COST_DESIGN

```
% Cost_Design: Definition of cost functions for reaching goal state and/or
% passing through via-point.
%
% --
% Control for Robotics
% AER1517 Spring 2020
% Programming Exercise 1
%
```

```
% --
% 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
%
% 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 = 'goal_state'; % 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 = Cost.Q_lqr;
        Q_vp(4:end, 4:end) = zeros(9, 9);
        % =================================================================

        % don't penalize position deviations, drive system with final cost
        Cost.Qm(1:3,1:3) = zeros(3);

        % Additional penalization of final cost
        Cost.Qmf([1:3, 7:9], [1:3, 7:9]) = Cost.Qmf([1:3, 7:9], [1:3, 7:9]) * 20;

        % =================================================================
        % [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
```