CS 4/5789 - Programming Assignment 2

February 15, 2023

This assignment is due on **March 1, 2023** at 11:59pm.

Section 0: Requirements

This assignment uses Python 3+. It is recommended to use Anaconda to install Python as well as the required libraries. If using Anaconda, the required libraries are

- `conda install -c conda-forge gym`
- `conda install numpy`
- `conda install -c conda-forge pyglet`

This last library is important for the visualization and also installs and sets up ffmpeg which is why we prefer Anaconda for use as a package manager.

If not using Anaconda, please use pip to install gym, numpy and pyglet. Please also install ffmpeg. For Mac and Linux (Ubuntu distro) users, using `brew install ffmpeg` or `sudo apt-get install ffmpeg`, respectively, should be sufficient. For Windows users, please follow the instructions here. The download link on that page is broken so please download `ffmpeg-git-full.7z` found here.

If the visualization doesn't work (which is required for submission) locally, then it may be useful to port your code over to a colab notebook and visualize things there. The tutorial on OpenAI Gym in the next section has code for visualization.

Section 1: Introduction

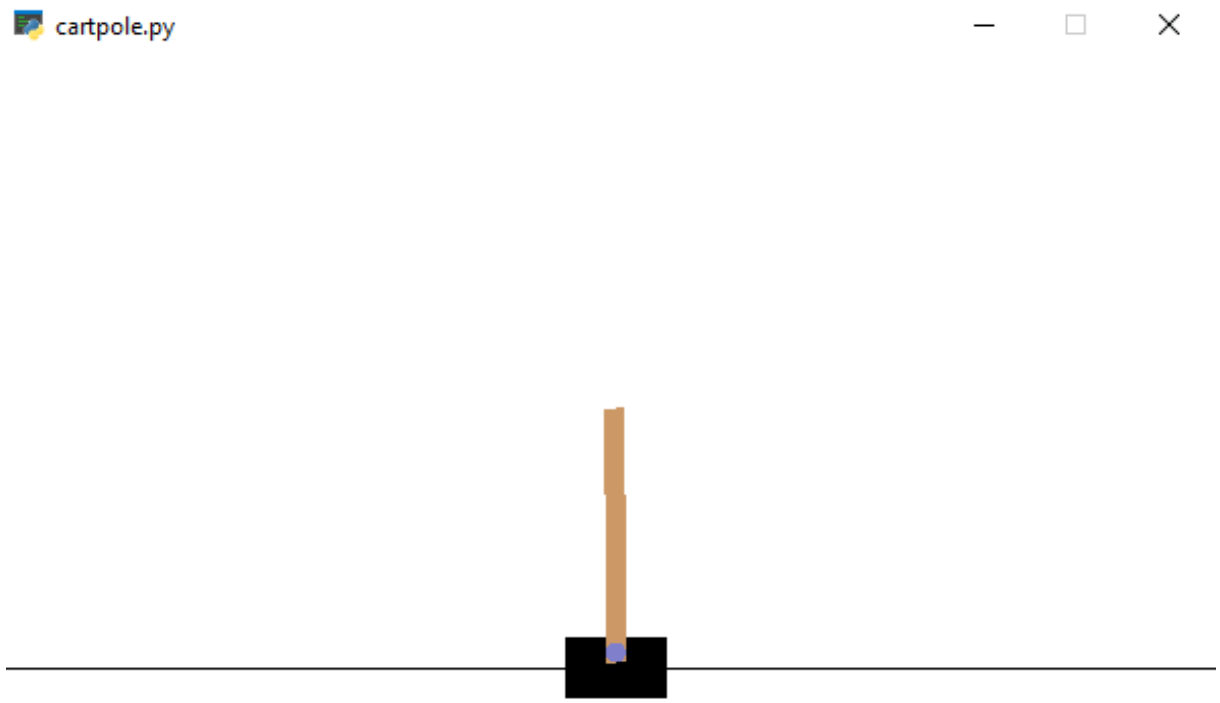
For this assignment, we will ask you to implement LQR in order to solve a cartpole environment. While we are using a custom cartpole environment, it would be useful to go through the following OpenAI Gym introduction. It may also be useful to look up how gym environments work in general. Wen-Ding, a graduate TA of CS 4789 from SP21, created a tutorial that may be useful.

OpenAI Gym contains many environments that can be used for testing RL algorithms. Additionally, it allows users to create their own custom environments. All these environments share a standard API, making it easy to test different algorithms on different environments. As mentioned earlier, we are using a custom version of the cartpole environment. The goal here is to keep the pole on the cartpole upright by applying a variable force either left or right on the cartpole as shown below.

In the standard cartpole environments, the only possible actions are to apply a force left or right. The amount of force applied is constant. In our environment, we allow variable force to be applied.

There are several files in this project.

- `LQR.py`: This contains the code to find the optimal control policy for an LQR given the respective matrices.
- `finite_difference_method.py`: This file is used to finite difference approximations. These will be used to help compute the reward and transition matrices from the environment which we then pass to the LQR.
- `test.py`: Test case files. Run this with `python test.py`. These will contains tests for different parts of the code.
- `cartpole.py` This contains the standard loop for interacting with the environment.



Section 2: Deliverables

There are 2 parts to this programming assignment.

- Section 3's task goes through the derivation of the generalized LQR formulation. **There is no coding in this section**. We break the proof down into intermediate steps and show the final results. You should try the proofs yourself and verify the results yourself. **You are responsible for understanding the proof for exams**.
- Section 4's task is to implement local linearization control for the cartpole non-linear setting. Section 4 depends on section 3 so make sure to understand the derivation done in section 3. You will need to complete the functions in
  - `finite_difference_method.py`
  - `cartpole_controller.py`
  - `LQR.py`

  In `cartpole_controller.py`, we also include a class called `PIDController`. It is an optional task for people who are interested in learning. You are welcomed to play with it and tune the parameters for a lower cost.

  Once these are done, you will need to run some code for testing and visualizing the cartpole. You will turn in the test output and video. More instructions can be found at the bottom of this handout.

Section 3: LQR

In the class, we introduced the most basic formulation of LQR. In this section, we are going to slightly make the model more general. We are interested in solving the following problem

$$\min_{\pi_0,\cdots,\pi_{T-1}} \mathbf{E}\left[\sum_{t=0}^{T-1} s_t^\top Q s_t + a_t^\top R a_t + s_t^\top M a_t + q^\top s_t + r^\top a_t + b\right] \tag{1}$$

$$\text{subject to } s_{t+1} = As_t + Ba_t + m, a_t = \pi_h(s_t) \qquad s_0 \sim \mu_0 \tag{2}$$

Here we have $s \in \mathbb{R}^{n_s}, a \in \mathbb{R}^{n_a}, Q \in \mathbb{R}^{n_s \times n_s}$ (Q is positive definite), $M \in \mathbb{R}^{n_s \times n_a}, q \in \mathbb{R}^{n_s}, R \in \mathbb{R}^{n_a \times n_a}$ (R is positive definite), $r \in \mathbb{R}^{n_a}, b \in \mathbb{R}, A \in \mathbb{R}^{n_s \times n_s}, B \in \mathbb{R}^{n_s \times n_a}$, and $m \in \mathbb{R}^{n_s}$. We also always have the following matrix being positive definite:

$$\begin{bmatrix} Q & M/2 \\ M^\top/2 & R \end{bmatrix}$$

The difference between the above formulation and the formulation we had in class is that the cost function contains an additional second order term $s_t^\top M a_t$, first-order terms $q^\top s_t, r^\top a_t$ and zeroth order terms b, and the dynamics contain zeroth order term m. The transitions here are deterministic.

In this problem, we will be asking you to derive the expressions for $Q_t^\star, \pi_t^\star$, and $V_t^\star$ for the base and inductive cases like in class. The difference this time is $V_t^\star$ and $\pi_t^\star$ can be thought of as complete quadratic and complete linear functions as follows

$$V_t^\star(s) = s^\top P_t s + y_t s + p_t$$
$$\pi_t^\star(x) = K_t^\star s + k_t^\star$$

where $P_t \in \mathbb{R}^{n_s \times n_s}$ (P_t is PSD), $y_t \in \mathbb{R}^{n_s}, p_t \in \mathbb{R}$ and $K_t^\star \in \mathbb{R}^{n_a \times n_s}, k_t^\star \in \mathbb{R}^{n_a}$. The following sections outline the derivation but do not go through all the steps. These are left as an exercise to students.

3.1 Base Case

3.1.1 $Q_{T-1}^\star$

$$Q_{T-1}^\star(s,a) = s^\top Q s + a^\top R a + s^\top M a + q^\top s + r^\top a + b$$

3.1.2 Extracting the policy

We can derive the expression from $\pi_{T-1}^\star$ as done in class. Then, we can write out the expression for $K_{T-1}^\star, k_{T-1}^\star$. If you've done things correctly, you should get the following

$$\pi_{T-1}^\star(s) = -\frac{1}{2}R^{-1}M^\top s - \frac{1}{2}R^{-1}r$$
$$\therefore K_{T-1}^\star = -\frac{1}{2}R^{-1}M^\top$$
$$\therefore k_{T-1}^\star = -\frac{1}{2}R^{-1}r$$

3.1.3 $V_{T-1}^\star$

Recall from class that $V_t^\star(s) = Q_t^\star(s, \pi_t^\star(s))$.

Using $Q_{T-1}^\star$ and $\pi_{T-1}^\star$, we can derive the expression for $V_{T-1}^\star(s)$. We'll keep $K_{T-1}^\star, k_{T-1}^\star$ in the expression and then $P_{T-1}, y_{T-1}, p_{T-1}$. If you've done everything correctly, you should get the following

$$P_{T-1} = Q + {K_{T-1}^\star}^\top R K_{T-1}^\star + M K_{T-1}^\star$$
$$y_{T-1}^\top = q^\top + 2(k_{T-1}^\star)^\top R K_{T-1}^\star + (k_{T-1}^\star)^\top M^\top + r^\top K_{T-1}^\star$$
$$p_{T-1} = (k_{T-1}^\star)^\top R k_{T-1}^\star + r^\top k_{T-1}^\star + b$$
$$V_{T-1}^\star(s) = s^\top P_{T-1} s + y_{T-1}^\top s + p_{T-1}$$

These expressions can be simplified as shown below but the above expressions will be useful when deriving the inductive case.

$$P_{T-1} = Q - \frac{1}{4} M R^{-1} M^\top$$
$$y_{T-1}^\top = q^\top - \frac{1}{2} r^\top R^{-1} M^\top$$
$$p_{T-1} = b - \frac{1}{4} r^\top R^{-1} r$$

Note above, we technically need to show that P_{T-1} is PSD before moving on. You can try to show this.

3.2 Inductive step

For the inductive step, assume $V_{t+1}^\star(s) = s^\top P_{t+1} s + y_{h+1}^\top s + p_{h+1}$ where P_{t+1} is PSD.

3.2.1 $Q_t^\star$

We can derive the expression for $Q_t^\star(s, a)$ following the steps done in class. If done correctly, you should get the following

$$Q_t^\star(s,a) = s^\top C s + a^\top D a + s^\top E a + f^\top s + g^\top a + h$$

where

$$C = Q + A^\top P_{t+1} A$$
$$D = R + B^\top P_{t+1} B$$
$$E = M + 2A^\top P_{t+1} B$$
$$f^\top = q^\top + 2m^\top P_{t+1} A + y_{t+1}^\top A$$
$$g^\top = r^\top + 2m^\top P_{t+1} B + y_{t+1}^\top B$$
$$h = b + m^\top P_{t+1} m + y_{t+1}^\top m + p_{t+1}$$

You should notice that $Q_t^\star(s, a)$ is similar to the expression for $Q_{T-1}^\star(s, a)$. Although we are not asking you to prove this, you should verify for yourself that C and D are positive definite matrices. Thus, we can use the exact same steps as in 3.1.2 and 3.1.3 to find $\pi_t^\star$ and $V_t^\star$.

3.2.2 Extracting the policy

Following the same steps in 3.1.2, we get that

$$\pi_t^\star(s) = -\frac{1}{2} D^{-1} E^\top s - \frac{1}{2} D^{-1} g$$
$$K_t^\star = -\frac{1}{2} D^{-1} E^\top$$
$$k_t^\star = -\frac{1}{2} D^{-1} g$$

Please verify these for yourself.

3.2.3 $V_t^\star$

Using the same steps as in 3.1.3, we can derive $V_t^\star(s)$. Since we know

$$Q_t^\star(s,a) = s^\top C s + a^\top D a + s^\top E a + f^\top s + g^\top a + h$$

then we have

$$P_t = C + {K_t^\star}^\top D K_t^\star + E K_t^\star$$
$$y_t^\top = f^\top + 2(k_t^\star)^\top D K_t^\star + (k_t^\star)^\top E^\top + g^\top K_t^\star$$
$$p_t = (k_t^\star)^\top D k_t^\star + g^\top k_t^\star + h$$
$$V_t^\star = x^\top P_t x + y_t^\top x + p_t$$

Section 4: Programming: Local Linearization Approach for Controlling CartPole

4.1 Background on Local Linearization Approach for Nonlinear Control

Consider the following general nonlinear control problem:

$$\min_{\pi_0,\ldots,\pi_{T-1}} \sum_{t=0}^{H-1} c(s_t, a_t)$$
$$\text{subject to:} s_{t+1} = f(s_t, a_t), a_t = \pi_t(s_t), s_0 \sim \mu_0;$$

where $f : \mathbb{R}^{n_s} \times \mathbb{R}^{n_a} \mapsto \mathbb{R}^{n_s}$.

In general the cost $c(s, a)$ could be anything. Here we focus on a special instance where we try to keep the system stable around some stable point $(x^\star, a^\star)$, i.e., our cost function $c(s, a)$ penalizes the deviation to $(s^\star, a^\star)$, i.e., $c(s, a) = \rho\left(s_t - s^\star\right) + \rho(a_t - a^\star)$, where ρ could be some distance metric such as ℓ_1 or ℓ_2 distance.

To deal with nonlinear f and non-quadratic c, we use the linearization approach here. Since the goal is to control the robot to stay at the pre-specified stable point $(s^\star, a^\star)$, it is reasonable to assume that the system is approximately linear around $(s^\star, a^\star)$, and the cost is approximately quadratic around $(s^\star, a^\star)$. Namely, we perform first-order taylor expansion of f around $(s^\star, a^\star)$, and we perform second-order taylor expansion of c around $(s^\star, a^\star)$:

$$f(s, a) \approx A(s - s^\star) + B(a - a^\star) + f(s^\star, a^\star),$$
$$c(s, a) \approx \frac{1}{2} \begin{bmatrix} s - s^\star \\ a - a^\star \end{bmatrix}^\top \begin{bmatrix} Q & M \\ M^\top & R \end{bmatrix} \begin{bmatrix} s - s^\star \\ a - a^\star \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} s - s^\star \\ a - a^\star \end{bmatrix} + c(s^\star, a^\star);$$

Here A and B are Jacobians, i.e.,

$$A \in \mathbb{R}^{n_s \times n_s} : \; A[i, j] = \frac{\partial f[i]}{\partial s[j]}[s^\star, a^\star] : \qquad B \in \mathbb{R}^{n_s \times n_a}, B[i, j] = \frac{\partial f[i]}{\partial a[j]}[s^\star, a^\star],$$

where $f[i](s, a)$ stands for the i-th entry of $f(s, a)$, and $s[i]$ stands for the i-th entry of the vector s. Similarly, for cost function, we will have Hessian and gradients as follows:

$$Q \in \mathbb{R}^{n_s \times n_s} : Q[i, j] = \frac{\partial^2 c}{\partial s[i] \partial s[j]}[s^\star, a^\star], \quad R \in \mathbb{R}^{n_a \times n_a} : R[i, j] = \frac{\partial^2 c}{\partial a[i] \partial a[j]}[s^\star, a^\star]$$
$$M \in \mathbb{R}^{n_s \times n_a} : M[i, j] = \frac{\partial^2 c}{\partial s[i] \partial a[j]}[s^\star, a^\star], \quad q \in \mathbb{R}^{n_s} : q[i] = \frac{\partial c}{\partial s[i]}[s^\star, a^\star]$$
$$r \in \mathbb{R}^{n_a} : r[i] = \frac{\partial c}{\partial a[i]}[s^\star, a^\star].$$

We are almost ready compute a control policy using A, B, Q, R, M, q, r together with the optimal control we derived for the system in Eq. 1. One potential issue here is that the original cost function $c(s, a)$ may not be even convex. Thus the Hessian matrix $H := \begin{bmatrix} Q & M \\ M^\top & R \end{bmatrix}$ may not be a positive definite matrix. We will apply further approximation to make it a positive definite matrix. Denote the eigen-decomposition of H as $H = \sum_{i=1}^{n_s+n_a} \sigma_i v_i v_i^\top$ where σ_i are eigenvalues and v_i are corresponding eigenvectors. We approximate H as follows:

$$H \approx \sum_{i=1}^{n_s+n_a} \mathbf{1}\{\sigma_i > 0\} \sigma_i v_i v_i^\top + \lambda I, \tag{3}$$

where $\lambda \in \mathbb{R}^+$ is some small positive real number for regularization which ensures that after approximation, we get an H that is positive definite with minimum eigenvalue lower bounded by λ.

Note this Hessian matrix is slightly different than the one defined in section 3 but this is not important for the problem. **Additionally, you should not view these matrices as the same in Eq. 1**. We still need to reformulate this problem in that form as shown in section 4.4.

4.1.1 TODOs:

Before moving on, please review the concepts of gradient, hessians, and positive definite matrices above.

4.2 Setup of Simulated CartPole

The simulated CartPole has the following nonlinear deterministic dynamics $s_{t+1} = f(s_t, a_t)$, and potentially non-quadratic cost function $c(s, a)$ that penalizes the deviation of the state from the balance point $(s^\star, a^\star)$ where $a^\star = 0$, and $s^\star$ represents the state of CartPole where the pole is straight and the cart is in a pre-specified position. A state s_t is a 4-dimension vector defined as

$$s_t = \begin{bmatrix} x_t \\ v_t \\ \theta_t \\ \omega_t \end{bmatrix}$$

It consists of the position of the cart, the speed of the cart, the angle of the pole in radians, and the angular velocity of the pole. The action a_t is a 1-dimensional vector correspond to the force applied on the cart.

Through this section, we assume that we have black-box access to c and f, i.e., we can feed any (s, a) to f and c, we will get two scalar returns which are $f(s, a)$ and $c(s, a)$ respectively. Namely, we do not know the analytical math formulation of f and c (e.g., imagine that we are trying to control some complicated simulated humanoid robot. The simulator is the black-box f).

In this assignment we will use our customized OpenAI gym CartPole environment provided in the following repository. The environment is under `env` directory. The goal is to finish the implementation of `cartpole_controller.py` which contains a class to compute the locally linearized optimal policy of our customized CartPole environment. We also provide other files to help you get started.

4.2.1 TODO:

Please review the files for the programming assignment.

4.3 Using Finite Difference for Taylor Expansion

Since we do not know the analytical form of f and c, we cannot directly compute the analytical formulations for Jacobians, Hessians, and gradients. However, given the black-box access to f and c, we can use *finite difference* to approximately compute these quantities.

Below we first explain the finite differencing approach for approximately computing derivatives. Your task is to use finite differencing methods to compute A, B, Q, R, M, q, r.

To illustrate finite differencing, assume that we are given a function $g : \mathbb{R} \mapsto \mathbb{R}$. Given any $\alpha_0 \in \mathbb{R}$, to compute $g'(\alpha_0)$, we can perform the following process:

$$\textbf{Finite Difference for derivative: } \widehat{g'}(\alpha_0) := \frac{g(\alpha_0 + \delta) - g(\alpha_0 - \delta)}{2\delta},$$

for some $\delta \in \mathbb{R}^+$. Note that by the definition of derivative, we know that when $\delta \to 0^+$, the approximation approaches to $g'(\alpha_0)$. In practice, δ is a tuning parameter: we do not want to set it to 0^+ due to potential numerical issue. We also do not want to set it too large, as it will give a bad approximation of $g'(\alpha_0)$.

With $\widehat{g'}(\alpha)$ as a black-box function, we can compute the second-derivate using Finite differencing on top of it:

$$\textbf{Finite Difference for Second Deriviative: } \widehat{g''}(\alpha_0) := \frac{\widehat{g'}(\alpha_0 + \delta) - \widehat{g'}(\alpha_0 - \delta)}{2\delta}$$

Note that to implement the above second derivative approximator $\widehat{g''}(\alpha)$, we need to first implement the function $\widehat{g'}(\alpha)$ and treat it as a black-box function inside the implementation of $\widehat{g''}(\alpha)$. You can see that we need to query black-box f twice for computing $g'(\alpha_0)$ and we need to query black-box f four times for computing $g''(\alpha_0)$.

Similar ideas can be used to approximate Jacobians, gradients, and Hessians.

At this end, using the provided Cartpole simulator which has black-box access to f and c, and the goal balance point $s^\star, a^\star$, to compute A, B, Q, R, q, r, with the provided value for δ.

4.3.1 TODO:

We provide minimum barebone functions and some tests for finite difference method in the file `finite_difference_method.py`. Complete the implementation of gradient, Jacobian and Hessian functions as we can use them to compute A, B, Q, R, M, q, r which will be used in the next section.

4.4 Computing Locally Optimal Control

With A, B, Q, R, M, q, r computed from finite differencing, let us first check if $\begin{bmatrix} Q & M \\ M^\top & R \end{bmatrix}$ a positive definite matrix (if it's not PD, your LQR formulation may run into the case where matrix inverse does not exist and and numerically you will observe NAN as well.). If not, let's use the trick in Eq. 3. Denote $H := \begin{bmatrix} Q & M \\ M^\top & R \end{bmatrix}$. We now are ready to solve the following linear quadratic system:

$$\min_{\pi_0,\ldots,\pi_{T-1}} \sum_{t=0}^{T-1} \frac{1}{2} \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix}^\top H \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + c(s^\star, a^\star), \tag{4}$$

$$\text{subject to } s_{t+1} = As_t + Ba_t + m, a_t = \pi_t(s_t), \quad s_0 \sim \mu_0. \tag{5}$$

With some rearranging terms, we can re-write the above program in the format of Eq. 1. This is shown below.

We can expand the cost function as

$$\begin{aligned}
&\frac{1}{2} \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix}^\top H \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + c(s^\star, a^\star) \\
&= \frac{1}{2} \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix}^\top \begin{bmatrix} Q & M \\ M^\top & R \end{bmatrix} \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + c(s^\star, a^\star) \\
&= \frac{1}{2}(s_t - s^\star)^\top Q(s_t - s^\star) + \frac{1}{2}2(s_t - s^\star)^\top M(a_t - a^\star) + \frac{1}{2}(a_t - a^\star)^\top R(a_t - a^\star) + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + c(s^\star, a^\star) \\
&= \frac{1}{2}(s_t^\top Q s_t - 2s^{\star\top} Q s_t + s^{\star\top} Q s^\star) + (s_t^\top M a_t - a^{\star\top} M^\top s_t - s^{\star\top} M a_t + s^{\star\top} M a^\star) \\
&\quad + \frac{1}{2}(a_t^\top R a_t - 2a^{\star\top} R a_t + a^{\star\top} R a^\star) + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + c(s^\star, a^\star) \\
&= \frac{1}{2} s_t^\top Q s_t - s^{\star\top} Q s_t + \frac{1}{2} s^{\star\top} Q s^\star + s_t^\top M a_t - a^{\star\top} M^\top s_t - s^{\star\top} M a_t + s^{\star\top} M a^\star + \frac{1}{2} a_t^\top R a_t - a^{\star\top} R a_t + \frac{1}{2} a^{\star\top} R a^\star \\
&\quad + q^\top s_t - q^\top s^\star + r^\top a_t - r^\top a^\star + c(s^\star, a^\star) \\
&= s_t^\top (\frac{Q}{2}) s_t + a_t^\top (\frac{R}{2}) a_t + s_t^\top M a_t + (q^\top - s^{\star\top} Q - a^{\star\top} M^\top) s_t + (r^\top - a^{\star\top} R - s^{\star\top} M) a_t + \\
&\quad (c(s^\star, a^\star) + \frac{1}{2} s^{\star\top} Q s^\star + \frac{1}{2} a^{\star\top} R a^\star + s^{\star\top} M a^\star - q^\top s^\star - r^\top a^\star)
\end{aligned}$$

Next, let us expand out the transition function, f. As in section 3.1, we perform first-order taylor expansion of f around $(s^\star, a^\star)$ so our transition is defined by

$$\begin{aligned}
s_{t+1} &= A(s - s^\star) + B(a - a^\star) + f(s^\star, a^\star) \\
&= As - As^\star + Ba - Ba^\star + f(s^\star, a^\star) \\
&= As + Ba + (f(s^\star, a^\star) - As^\star - Ba^\star)
\end{aligned}$$

Thus, let us define the following variables

$$
\begin{aligned}
Q_2 &= \frac{Q}{2} \\
R_2 &= \frac{R}{2} \\
q_2^\top &= q^\top - s^{\star\top} Q - a^{\star\top} M^\top \\
r_2^\top &= r^\top - a^{\star\top} R - s^{\star\top} M \\
b &= c(s^\star, a^\star) + \frac{1}{2} s^{\star\top} \frac{Q}{2} s^\star + \frac{1}{2} a^{\star\top} R a^\star + s^{\star\top} M a^\star - q^\top s^\star - r^\top a^\star \\
m &= f(s^\star, a^\star) - A s^\star - B a^\star
\end{aligned}
$$

Thus, we can rewrite our formulation as

$$
\begin{aligned}
\min_{\pi_0, \cdots, \pi_{T-1}} \sum_{t=0}^{T-1} & s_t^\top Q_2 s_t + a_t^\top R_2 a_t + s_t^\top M a_t + q_2^\top s_t + r_2^\top a_t + b \\
\text{subject to } & s_{t+1} = A s_t + B a_t + m, a_t = \pi_h(s_t), s_0 \sim \mu_0
\end{aligned}
$$

This is exactly the same formulation as in section 3. Thus, we use the formulation there to derive the optimal policies.

4.4.1 TODO:

Please complete the functions in `lqr.py`. using the formulation derived above. We recommend implementing the functions in order. For the LQR function specifically, you will need to compute the optimal policies for time steps $T-1, \cdots, 0$. As you've done in section 3, the general pipeline at timestep t is to derive $Q_t^\star$, find $\pi_t^\star$, then finally derive $V_t^\star$ before moving on to the next time-step.

Please complete the function `compute_local_policy` in `cartpole_controller.py`. The function computes the linear locally optimal control policy of the CartPole environment. You should use the finite difference function as well as the lqr functions to implement this function.

There are a few tests in `test.py` for testing your functions.

Note: It may be useful to occasionally check that cast the numpy arrays you work with are of type np.float64. While you shouldn't need to explicitely cast things as all our tests initialize arrays to be of type np.float64, occasionally checking for it can avoid type issues. The gym environment is built around np.float64 and using other types can cause incorrect results.

4.5 PID Controller

A proportional–integral–derivative (PID) controller is a control loop mechanism employing feedback. The overall control function

$$
u_t = K_\mathrm{p} e_t + K_\mathrm{i} \sum_{k=0}^{t} e_k + K_\mathrm{d} (e_t - e_{t-1}),
$$

where K_p, K_i, and K_d, all non-negative, denote the coefficients for the proportional, integral, and derivative terms respectively. A PID controller continuously calculate an error value e_t as the difference between a desired setpoint SP $= r_t$ and a measured process variable PV $= y_t$: $e_t = r_t - y_t$, and applies a correction based on proportional, integral, and derivative terms. The controller attempts to minimize the error over time by adjustment of a control variable u_t, such as the opening of a control valve, to a new value determined by a weighted sum of the control terms.

The balance of PID's effects is achieved by loop tuning to produce the optimal control function. The tuning constants must be derived for each control application, as they depend on the response characteristics of the complete loop external to the controller. Approximate values of constants can usually be initially entered knowing the type of application, but they are normally refined, or tuned, by "bumping" the process in practice by introducing a setpoint change and observing the system response.

4.5.1 TODO

This section is optional. In `cartpole_controller.py`, the class `PIDController`, you can tune the `P`, `I`, `D` variables in order to achieve low cost.

4.6 Test the performance

Given policies $\pi_0, \ldots, \pi_{T-1}$ that computed from previous sections, we will evaluate its performance by executing it on the real system f and real cost c. To generate a T-step trajectory, we first sample $s_0 \sim \mu_0$, and then take $a_t = \pi_t(s_t)$, and then call the black-box f to compute $s_{t+1} = f(s_t, a_t)$ till $t = T-1$. This gives us a trajectory $\tau = \{s_0, a_0, s_1, a_1, \ldots, s_{T-1}, a_{T-1}\}$. The total cost of the trajectory is $C(\tau) := \sum_{t=0}^{T-1} c(s_t, a_t)$. Since we have randomness in μ_0, we need to draw N such i.i.d trajectories, $\tau^1 \ldots \tau^N$, and compute the average, i.e., $\sum_{i=1}^{N} C(\tau^i)/N$. As N approaches to $+\infty$, we have (by law of large numbers):

$$\sum_{i=1}^{N} C(\tau^i)/N \to \mathbb{E}_{s_0 \sim \mu_0} \left[\sum_{t=0}^{T-1} c(s_t, a_t) | a_t = \pi_t(s_t), s_{t+1} = f(s_t, a_t) \right].$$

In the test file `test.py`, we provide several difference initialization distributions. These distributions are ordered based on the distance between the means to the goal $(s^\star, a^\star)$. Intuitively, we should expect that our control perform worse when the initial states are far away from the taylor-expansion point $(s^\star, a^\star)$, as our linear and quadratic approximation become less accurate. Testing your computed policies on these difference initializations and report the performances for all initializations by running `test.py`.

This file also contains tests for testing your finite difference methods and the LQR functions.

4.6.1 TODO

Run `test.py` and take a screenshot of the output costs printed from `test_policy`. The costs should be as it can vary depending on the version of packages used (even with the environment being seeded). However, your costs should fall in the following ranges

- Case $0 < 10$
- Case $1 < 200$
- Case $2 < 1000$
- Case $3 < 2000$
- Case $4 < 5000$
- Case $5 = \infty$
- Case $6 = \infty$
- Case $7 = \infty$

Please know that these are loose upper bounds and the actual costs for a correct implementation may be much lower than the upper bound listed (apart from Case 5-7).

Please briefly explain why you think the costs are different for each case.

4.7 Visualization

We can now see the controller in action. Using the visualization tool provided by Open AI Gym, provide a video demonstration of your policy with the given initial distribution by running `cartpole.py`. The generated mp4 video file should be under the directory `./gym-results`. Save this as `cartpole.mp4`

4.7.1 TODO

Please check that the mp4 player can work. There have been known issues in the past with saving the mp4 so if this occurs, please reach out to a TA. Additionally, save the cost of the cartpole and include it in your writeup.

4.8 Comparison

You can compare the LQR policy to a very simple PID policy by changing `flag='LQR'` to `flag='PID'` in `cartpole.py`. You can also investigate how the two different policies perform when you change the initial condition `s_init`. Which works better? There are no deliverables for this section.

4.9 Submission

Please submit a zip file containing your implementation along with your video file name `cartpole.mp4` organized as follows.

```
YOUR_NET_ID/
├── Answers.pdf
├── README.md
├── __init__.py
├── cartpole.mp4
├── cartpole.py
├── cartpole_controller.py
├── finite_difference_method.py
├── lqr.py
├── test.py
└── env/
    ├── __init__.py
    └── cartpole_control_env.py
```

where `Answers.pdf` contains the screenshot of the cost of the cartpole simulations and your explanation from Section 4.6.1 as well as the cost from 4.7.1.