CS 4/5789 - Programming Assignment 2

February 19, 2024

This assignment is due on **March 6, 2024** at 11:59pm.

Section 0: Requirements

This assignment uses Python 3.10+ and `ffmpeg`. It is recommended to use Anaconda to install Python as well as the required libraries.

Please install the corresponding `setup.py` file with `pip install -e .` in the directory of the `setup.py` file. `pyglet` 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`, `tqdm`, 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. **However note that we do not officially support not using the python package. Use as your own risk**

Section 1: Introduction

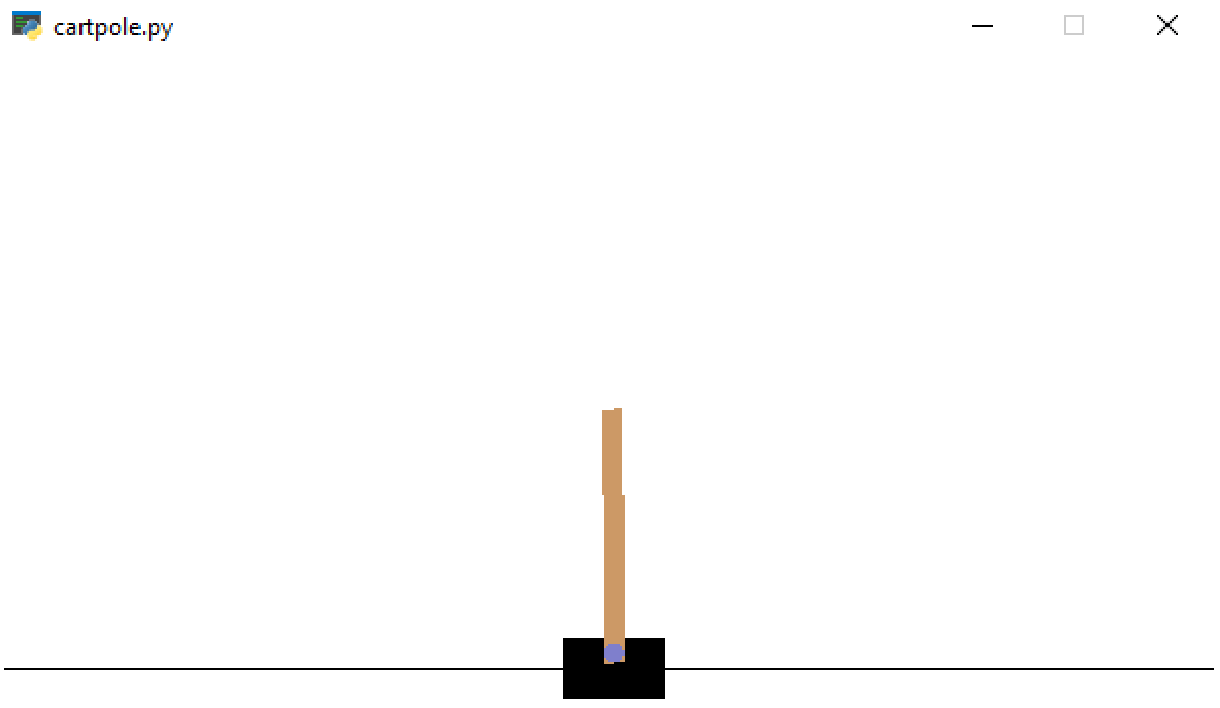
For this assignment, we will ask you to implement LQR (and DDP) in order to solve a cartpole environment. While we are using a custom cartpole environment, it would be useful to go through the following OpenAI Gym introduction. It may also be useful to look up how gym environments work in general. Wen-Ding, 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[1] 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.

There are several files in this project.

- `lqr.py`: This contains the code to find the optimal control policy for LQR given the dynamics and cost 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.
- `ddp.py`: This contains code which implements DDP.
- `local_controller.py`: This contains cartpole environment specific elements and `compute_local_expansion`.
- `cartpole.py` This contains the standard loop for interacting with the environment. One can use `--env DDP` to run DDP or `--env LQR` to run local LQR. You can use the `-h` flag for more information about usage.

[1]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.



Section 2: Deliverables

There are 3 parts to this programming assignment.

- Section 3 goes through the derivation of the generalized LQR formulation. 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**. **There is no coding in this section**.
- Section 4 describes a local linearization approach to stabilizing the cartpole. Section 4.3's task is to implement local linearization control for the cartpole to stay vertically upwards. You will need to complete the functions in
  - `finite_difference_method.py`
  - `local_controller.py`
  - `LQR.py`
- Section 5 goes through the derivation of DDP. After understanding the procedure, you are asked to complete functions in `DDP.py`.

  Once the programming tasks are done, you will need to test and visualize the cartpole. You will turn in a report containing the test output and images. Complete instructions can be found at the end of Section 6.

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

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

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

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:

$$\begin{bmatrix} Q & M/2 \\ M^\top/2 & R \end{bmatrix} \tag{3}$$

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 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

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

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.

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

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

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

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

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

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

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

Section 4: Programming: Local Linearization Approach for Controlling CartPole

4.1 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 `lqr.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.

4.1.1 TODO:

Please review the files for the programming assignment.

4.2 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 will later 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 gradients, Jacobians, and Hessians. At this end, we can use the provided Cartpole simulator which has black-box access to f and c and the goal balance point $s^\star, a^\star$, to compute the taylor expantion around the balancing point.

4.2.1 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 so that we can use them in the next section.

4.3 Local Linearization Approach for Nonlinear Control

Formally, consider our objective:

$$\min_{\pi_0,\ldots,\pi_{T-1}} \sum_{t=0}^{T-1} c(s_t, a_t) \tag{4}$$

$$\text{subject to: } s_{t+1} = f(s_t, a_t), a_t = \pi_t(s_t), s_0 \sim \mu_0; \tag{5}$$

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., $c(s, a) = \rho(s_t - s^\star) + \rho(a_t - a^\star)$, 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 $(s^\star, a^\star)$:

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

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

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

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 **Hessians** and **gradients** as follows:

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

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 force H to be convex as follows:

$$H \leftarrow \sum_{i=1}^{n_s+n_a} \mathbf{1}\{\sigma_i > 0\} \sigma_i v_i v_i^\top + \lambda I, \tag{6}$$

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 from 3. **You should not view these matrices as the same in Eq. 1 yet**. We still need to reformulate this problem in that form as shown in section 4.4

4.3.1 TODOs:

Review the concepts of gradient, Jacobian, Hessian, and positive definite matrices. With the previous implementation of finite difference methods, you can complete the function `compute_local_expansion` in `local_controller.py` to calculate the A, B, Q, R, M, q, r as we defined above.

4.4 Computing Locally Optimal Control

With A, B, Q, R, M, q, r computed, let us first check if $H = \begin{bmatrix} Q & M \\ M^\top & R \end{bmatrix}$ a positive definite matrix (if it's not PD, the LQR formulation may run into the case where matrix inverse does not exist and and numerically you will observe NAN as well.). If not, you must use the trick in Eq. 6 to modify H. We now are ready to solve the following linear quadratic system:

$$\min_{\pi_0,\dots,\pi_{T-1}} \sum_{t=0}^{T-1} \frac{1}{2} \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix}^\top H \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + \begin{bmatrix} q \\ r \end{bmatrix}^\top \begin{bmatrix} s_t - s^\star \\ a_t - a^\star \end{bmatrix} + c(s^\star, a^\star), \tag{7}$$

$$\text{subject to } s_{t+1} = As_t + Ba_t + m, a_t = \pi_t(s_t), \quad s_0 \sim \mu_0. \tag{8}$$

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 function `lqr` in `lqr.py` using the formulation derived above. You will need to compute the optimal policies for time steps $T-1, ..., 0$. At timestep t, we find find $\pi_t^\star$ using $Q_t^\star$.

Section 5: Differential Dynamic Programming

In the LQR setting of equation 1, where the cost is written with respect to some fixed matrices and vectors (i.e., Q, R, M, q, r, b), we are able to derive the closed-loop solutions of the policy. So far, we used this insight to approximate nonlinear control as LQR. However, recall that we originally derived the LQR policy using dynamic programming. What if we apply the idea of approximation to the dynamic programming steps directly? For our policy at time step i, we care about the *cost-to-go* as the accumulated sum of the costs in the future steps. The optimal *Value* at time i is

$$V_i^\star(s) \equiv \min_{\pi_i,\ldots,\pi_{T+1}} \sum_{t=i}^{T-1} c(s_t, a_t) \quad \text{s.t.} \quad s_i = s, s_{t+1} = f(s_t, a_t), a_t = \pi_t(s_t).$$

Using the finite horizon Bellman Optimality Equation (i.e. the principle behind Dynamic Programming), we can reduce this minimization over the entire sequence of controls to one action:

$$V_i^\star(s) = \min_a [c(s,a) + V_{i+1}^\star(f(s,a))] = \min_a Q_i^\star(s,a).$$

Let's try to do a second order approximation to $Q_i^\star(s,a)$ directly. **While you are responsible for understanding the motivation and basic ideas (quadratic approximation, dynamic programming) behind DDP, you are not required understand the technical details of the following derivation precisely.** We will approximate around a point s_0, a_0. Let $s = s_0 + \delta s$ and $a = a_0 + \delta a$ be small perturbations. Then,

$$Q_i^\star(s,a) - Q_i^\star(s_0,a_0) = c(s_0 + \delta s, a_0 + \delta a) - c(s_0,a_0) + V_{i+1}^\star(f(s_0 + \delta s, a_0 + \delta a)) - V_{i+1}^\star(f(s_0,a_0))\,.$$

Approximating around $\delta s = 0, \delta a = 0$ to second order results in

$$Q_i^\star(s,a) - Q_i^\star(s_0,a_0) \approx \frac{1}{2}\begin{bmatrix} 1 \\ \delta s \\ \delta a \end{bmatrix}^\top \begin{bmatrix} 0 & Q_s^\top & Q_a^\top \\ Q_s & Q_{ss} & Q_{sa} \\ Q_a & Q_{as} & Q_{aa} \end{bmatrix} \begin{bmatrix} 1 \\ \delta s \\ \delta a \end{bmatrix}. \tag{9}$$

The expansion coefficients are

$$\begin{aligned} Q_s &= c_s + f_s^\top V_s^{i+1}, \quad Q_s \in \mathbb{R}^{n_s} \\ Q_a &= c_a + f_a^\top V_s^{i+1}, \quad Q_a \in \mathbb{R}^{n_a} \\ Q_{ss} &= c_{ss} + f_s^\top V_{ss}^{i+1} f_s + V_s^{i+1} \cdot f_{ss}, \quad Q_{ss} \in \mathbb{R}^{n_s \times n_s} \\ Q_{aa} &= c_{aa} + f_a^\top V_{ss}^{i+1} f_a + V_s^{i+1} \cdot f_{aa}, \quad Q_{aa} \in \mathbb{R}^{n_a \times n_a} \\ Q_{as} &= c_{as} + f_a^\top V_{ss}^{i+1} f_s + V_s^{i+1} \cdot f_{as}, \quad Q_{as} \in \mathbb{R}^{n_a \times n_s} \end{aligned}$$

where we denote: c_s and c_a the gradients of the cost function; c_{ss}, c_{aa}, c_{as} the Hessians of the cost function; f_s and f_a the Jacobian of the transition function; f_{ss}, f_{aa} and f_{as} the Jacobian of the Jacobian of the transition function (this is a 3D tensor!); V_s^{i+1} and V_{ss}^{i+1} are the coefficients of the expanded value function at time step $i+1$, which will be defined later.

Our policy minimizes δa with respect to Q using the quadratic approximation in Eq. 9. With simple computation, we can get

$$\delta a^\star = -Q_{aa}^{-1}(Q_a + Q_{as}\delta s).$$

We define $\mathbf{k}_i = -Q_{aa}^{-1}Q_a$ and $\mathbf{K}_i = -Q_{aa}^{-1}Q_{as}$. Plug the policy into 9, we can derive (iteratively as using dynamic programming) the quadratic approximation of the value functions at timestep i:

$$\begin{aligned} V_i^\star(s) - V_i^\star(s_0) &\approx -\frac{1}{2} Q_a^\top Q_{aa}^{-1} Q_a \\ V_s^i &= Q_s - Q_{sa}^\top Q_{aa}^{-1} Q_a \\ V_{ss}^i &= Q_{ss} - Q_{sa}^\top Q_{aa}^{-1} Q_{as} \end{aligned}$$

The above procedure computes the control actions and values from $T-1$ to 0 in a "backwards pass." And once the backward pass is complete, we can use the policy to roll-out a new trajectory, which gives us new states and actions (s_0, a_0) to linearize around. Formally, the forward pass calculates:

$$\begin{aligned} \hat{s}_0 &= s_0 \\ \hat{a}_i &= a_i + \mathbf{k}_i + \mathbf{K}_i(\hat{s}_i - s_i) \\ \hat{s}_{i+1} &= f(\hat{s}_i, \hat{a}_i) \end{aligned}$$

The algorithm iterates until the forward pass and the backward pass converge.

5.1 DDP Implementation

We will implement DDP to complete a more challenging task—making the cartpole swing up.

5.1.1 TODO:

There are two main sections that need to be completed in `DDP.py`: `forward()`, which is the forward dynamics and the main driver code in `train()`. We have provided a small starter implementation for the number of timeststeps that it took for our solution to swing up the cartpole. But if yours takes less, feel free to change it.

Section 6: Testing the performance

6.1 LQR Stress Tests

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 $C(\tau) := \sum_{t=0}^{T-1} c(s_t, a_t)$.

In the main file `cartpole.py`, we provide several difference initialization states. These initial states 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.

6.1.1 TODO

Run `cartpole.py --env LQR` and take a screenshot of the output costs printed from running the LQR version. The costs 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.

6.2 DDP Tests

Likewise, we also provides different initial states for the swing-up task that are ordered such that their distance are increasing from the goal state. This time, since DDP no longer depends on the assumption of local linearization, we should expect our control to be able to complete the swing-up task even far away from the goal state.

6.2.1 TODO

Run `cartpole.py --env DDP --init_s far` and describe the behavior of the trained cartpole. Since the training process for DDP takes longer, we ask you to manually test the initial states one by one. You can simply exchange the `--init_s` to be {`far`,`close`,`exact`}.

Observe the generated videos, for each test case you run: how does DDP perform with the different initial states?

Compare the LQR policy and the DDP policy: how do they perform on different initial states? What's the reason behind their differences?

6.3 Submission

Please submit a zip file containing your implementation as follows:

```
YOUR_NET_ID/
 ├─ Answers.pdf
 ├─ README.md
 ├─ __init__.py
 ├─ cartpole.py
 ├─ local_controller.py
 ├─ finite_difference_method.py
 ├─ lqr.py
 ├─ ddp.py
 ├─ test.py
 └─ env/
     ├─ __init__.py
     ├─ cartpole_lqr_env.py
     └─ cartpole_ilqr_env.py
```

where `Answers.pdf` contains the screenshot of the cost of the cartpole simulations, the description of what happen in the three initial states in the cartpole swing-up task, and your explanations to the above highlighted three research questions.