UNIVERSITY OF PENNSYLVANIA

ESE 650: LEARNING IN ROBOTICS

SPRING 2023

[03/13] HOMEWORK 3

DUE: 04/10 MON 11.59 PM ET

Changelog: This space will be used to note down updates/errata to the homework problems.

- **Page 4: Line 21: Third bullet is removed. In short the robot can enter obstacles (which are all grey cells) but it has to stay there indefinitely and incur the penalty.**

Read the following instructions carefully before beginning to work on the homework.

- You will submit solutions typeset in LaTeX on Gradescope (strongly encouraged). You can use hw_template.tex on Canvas in the "Homeworks" folder to do so. If your handwriting is *unambiguously legible*, you can submit PDF scans/tablet-created PDFs.
- Please start a new problem on a fresh page and mark all the pages corresponding to each problem. Failure to do so may result in your work not graded completely.
- Clearly indicate the name and Penn email ID of all your collaborators on your submitted solutions.
- **For each problem in the homework, you should mention the total amount of time you spent on it. This helps us gauge the perceived difficulty of the problems.**
- You can be informal while typesetting the solutions, e.g., if you want to draw a picture feel free to draw it on paper clearly, click a picture and include it in your solution. Do not spend undue time on typesetting solutions.
- You will see an entry of the form "HW 3 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 3 Problem 1 Code" where you will upload your solution for the respective problems. **For each programming problem, you should create a fresh Python file**. This file should contain all the code to reproduce the results of the problem and you will upload the .py file to Gradescope. If we have installed Autograder

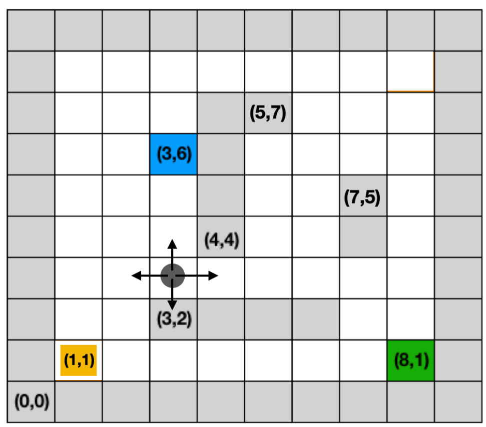
for a particular problem, you will use the Autograder. Name your file to be "pennkey_hw3_problem1.py", e.g., I will name my code for Problem 1 as "pratikac_hw3_problem1.py".

- **You should include all the relevant plots in the PDF, without doing so you will not get full credit. There is no auto-grader for this homework so this is particularly important.** You can, for instance, export your Jupyter notebook as a PDF (you can also use text cells to write your solutions) and export the same notebook as a Python file to upload your code.
- **Your PDF solutions should be completely self-contained. We will run the Python file to check if your solution reproduces the results in the PDF.**

Credit. The points for the problems add up to 110. You only need to solve for 100 points to get full credit, i.e., your final score will be $\min(\text{your total points}, 100)$.

Problem 1 (Policy Iteration, 20 points). Consider the following Markov Decision Process. The state-space is a 10×10 grid, cells that are obstacles are marked in gray. The initial state of the robot is in blue and our desired terminal state is in green. The robot gets a *reward* of 10 if it reaches the desired terminal state with a discount factor of 0.9. At each non-obstacle cell, the robot can attempt to move to any of the immediate neighboring cells using one of the four controls (North, East, West and South). The robot cannot diagonally. The move succeeds with probability 0.7 and with remainder probability 0.3 the robot can end up at some other cell as follows:

$$\begin{aligned} \mathrm{P}(\text{moves north} \mid \text{control is north}) &= 0.7, \\ \mathrm{P}(\text{moves west} \mid \text{control is north}) &= 0.1, \\ \mathrm{P}(\text{moves east} \mid \text{control is north}) &= 0.1, \\ \mathrm{P}(\text{does not move} \mid \text{control is north}) &= 0.1. \end{aligned}$$



Similarly, if the robot desired to go east, it may end up in the cells to its north, south, or stay put at the original cell with total probability 0.3 and actually move to the cell east with probability 0.7. The cost pays a cost of 1 (i.e., reward is -1) for each control input it takes, regardless of the outcome. If the robot ends up at a state marked as an obstacle (all grey cells are obstacles, i.e., cell marked (0,0), (0,1), (3,2) etc. are obstacles), it gets a reward of -10 for each time-step that it remains inside the obstacle cell. The robot is allowed to stay in the goal state indefinitely (i.e., take a special action to "not move") and this action gets no reward/cost.

We would like to implement policy iteration to find the best trajectory for the robot to go from the blue cell to the green cell.

(a) **(0 points)** Carefully code up the above environment to run policy iteration. You will need to think about how to code up the probability transition matrix $\mathbb{R}^{100 \times 100} \ni T_{x,x'}(u) = \mathrm{P}(x' \mid x, u)$, the run-time cost $q(x, u)$, and the terminal cost $q_f(x)$. Policy iteration is easy to implement if you represent all the above quantities as matrices and vectors. Plot the environment to check if it confirms to the above picture.

(b) **(10 points)** Initialize policy iteration with a feedback control $u^{(0)}(x)$ where the robot always goes east, this results in a policy $\pi^{(0)} = (u^{(0)}(\cdot), u^{(0)}(\cdot), \dots)$. Write the code for policy evaluation to obtain the cost-to-go from every cell in the above picture for this initial policy. Plot the value function $J^{\pi^{(0)}}(x)$ as a heatmap in the above picture.

(c) **(10 points)** Execute the policy iteration algorithm, you will iteratively perform policy evaluation and policy improvement steps. For the first 4 iterations, plot the feedback control $u^{(k)}(x)$ (using arrows as shown in the lecture notes (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.arrow.html, you can also write the control input in the cell). You should color the cell using the value function $J^{\pi^{(k)}}(x)$.

We have left the transition probabilities and the reward structure a bit vague to force you to think carefully of the nuances of this problem. But some clarification could be useful.

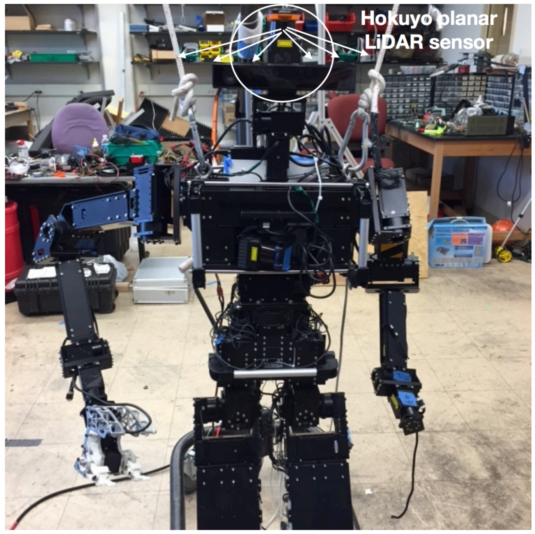
(1) You can code up what are called "sticky obstacles", i.e., if the robot enters an obstacle, then it stays there forever while incurring the obstacle cost at each time instant.

(2) It is easiest to think of the runtime cost in this problem as a function of three quantities $q(x, u, x')$ where x is the current state, u is the control and x' is the next state. The Bellman equation the becomes

$$J^*(x) = \min_{u \in U} \mathop{\mathrm{E}}_{x'} \left[q(x, u, x') + \gamma J^*(x') \right].$$

You will submit your own code for this problem; there is no auto-grader.

Problem 2 (Simultaneous Localization and Mapping (SLAM) with a particle filter, 60 points.). In this problem, we will implement mapping and localization in an indoor environment using information from an IMU and a LiDAR sensor. We have provided you data collected from a humanoid named THOR that was built at Penn and UCLA (https://www.youtube.com/watch?v=JhWYYuba1nE). You can read more about the hardware in this paper (https://ieeexplore.ieee.org/document/7057369.)



Hardware setup of Thor The humanoid has a Hokuyo LiDAR sensor (https://hokuyo-usa.com/products/lidar-obstacle-detection on its head (the final version of the robot had it in its chest but this is a different version); details of this are in the code (which will be explained shortly). This LiDAR is a planar LiDAR sensor and returns 1080 readings at each instant, each reading being the distance of some physical object along a ray that shoots off at an angle between (-135, 135) degrees with discretization of 0.25 degrees in an horizontal plane (shown as white rays in the picture). We will use the position and orientation of the head of the robot to calculate the orientation of the LiDAR in the body frame.

The second kind of observations we will use pertain to the location of the robot. However, in contrast to the previous homework were we used the raw accelerometer and gyroscope readings to get the orientation, we will directly use the (x, y, θ) pose of the robot in the world coordinates (θ denotes yaw). These poses were created presumably on the robot by running a filter on the IMU data (such estimates are called odometry estimates), and just as you saw some tracking errors in the previous homework, these poses will not be extremely accurate. However, we will treat them conceptually the same way as we treated Vicon in the previous homework, namely as a much more precise estimate of the pose of the robot that is used to check how well SLAM is working.

Coordinate frames The body frame is at the top of the head (X axis pointing forwards, Y axis pointing left and Z axis pointing upwards), the top of the head is at a height of 1.263m from the ground. The transformation from the body frame to the LiDAR frame depends upon the angle of the head (pitch) and the angle of the neck (yaw) and the height of the LiDAR above the head (which is 0.15m). The world coordinate frame where we want to build the map has its origin on the ground plane, i.e., the origin of the body frame is at a height of 1.263m with respect to the world frame at location (x, y, θ).

Data and code

(a) **(0 points)** We have provided you 4 datasets corresponding to 4 different trajectories of the robot in Towne Building at Penn. For example, dataset 0 consists of two files data/train/train_lidar0.mat and data/train/train_joint0.mat which contain the LiDAR readings and joint angles respectively. The functions load_lidar_data and load_joint_data inside load_data.py read the data. You can run the function show_lidar to see the LiDAR data. Each of the data reading functions returns a data-structure where t refers to the time-stamp (in seconds) of the data, xyth refers to (x, y, θ) *pose of the LiDAR* and rpy refers to Euler angles (roll, pitch, yaw). The joint data contains a number of fields, but we are only interested in the angle of the head and the neck at a particular time-stamp. The array slam_t.joint.head_angles contains the angles of neck and head respectively in the first two rows. This data is the same as the data inside the first two rows slam_t.joint.pos (that array contains the angles of all joints). You should read these functions carefully and check the values returned by them. The dicts joint_names and joint_names_to_index can be used to read off the data of a specific joint (we only need the head and the neck).

(b) **(0 points)** Next look at the slam.py file provided to you. Read the code for the class map_t and slam_t and the comments provided in the code very carefully. You are in charge of filling in the missing pieces marked as TODO: XXXXXX. A suggested order for studying this code is as follows: slam_t.read_data, slam_t.init_sensor_model, slam_t.init_particles, slam_t.rays2world, map_t.__init__, map_t.grid_cell_from_xy. Next, the file utils.py contains a few standard rigid-body transformations that you will need. You should pay attention to the functions smart_plus_2d and smart_minus_2d that will be used to code up the dynamics propagation step of the particle filter.

(c) **(10 points, dynamics step)** Next look at main.py which has two functions run_dynamics_step and run _observation_step which act as test functions to check if the particle filter and occupancy grid update has been updated correctly. The run_dynamics function plots the trajectory of the robot (as given by its IMU data in the LiDAR data-structure). It also initializes 3 particles and plots all particles at different time-steps while performing the dynamics step with a very small dynamics noise; this is a very neat way of checking if dynamics propagation in the particle filter is working correctly. This function will create two plots, one for the odometry trajectory and one more for the particle trajectories, both these trajectories should match after you code up the dynamics function slam_t.dynamics_step correctly.

(d) **(20 points, observation step)** The function run_observation_step is used to perform the observation step of the particle filter to get an estimate of the location of the robot and updates to the occupancy grid using observations from the LiDAR. First read the comments for the function slam_t.observation_step carefully.

We first discuss the particle filtering part.

(i) Compute the head and neck position for the time t. For each particle, assuming that that particle is indeed the true position of the robot, project the LiDAR scan slam_t.lidar[t]['scan'] into the world coordinates using the slam_t.ray2world function. The end points of each ray tell us which cells in the map are occupied, for each particle.

(ii) In order to compute the updated weights of the particle, we need to know the likelihood of LiDAR scans given the state (our current occupancy grid in the case of SLAM). We are going to use a simple model to do so

$$\log \mathrm{P}(\text{LiDAR scan as if the robot is at particle } p \mid m) = \sum_{ij \in O} m_{ij} \tag{1}$$

where O is the set of occupied cells as detected by the LiDAR scan assuming the robot is at particle p and m_{ij} is our current estimate of the binarized map (more on this below). In simple words, if the occupied cells as given by our LiDAR match the occupied cells in the binarized map created from the past observations, then we say the log-probability of particle p is large.

(iii) You will next implement the function slam_t.update_weights that takes the log-probability of each particle p, its previous weights, calculates the updated weights of the particles.

(iv) Typically, resampling step (slam_t.stratified_resampling) is performed only if the effective number of particles (as computed in slam_t.resample_particles) falls below a certain threshold (30% in the code). Implement resampling as we discussed in the lecture notes.

Mapping We have a number of particles $p^i = (x^i, y^i, \theta^i)$ that together give an estimate of the distribution of the location of the robot. For this homework, you will only use the particle with the largest weight to update the map although typically we update the map using all particles. Our goal is simple: we want to increase map_t.log_odds array at cells that are recorded as obstacles by the LiDAR and decrease the values in all other cells. You should add slam_t.log_odds_occ to all occupied cells and add slam_t.log_odds_free from all cells in the map. It is also a good idea to clip the log_odds to like between [-slam_t.map.log_odds_max, slam_t.map.log_odds_max] to prevent increasingly large values in the log_odds array. The array slam_t.map.cells is a binarized version of the map (which is used above to calculate the observation likelihood).

Check the run_observation_step function after you have implemented the observation step.

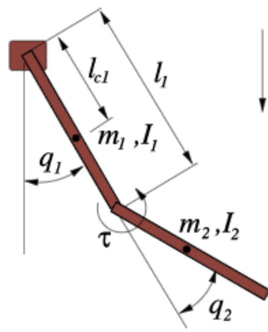
(e) Since the map is initialized to zero at the beginning of SLAM which results in all observation log-likelihoods to be zero in (1), we need to do something special for the first step. We will use the first entry in slam_t.lidar[0]['xyth'] to get an accurate pose for the robot and use its corresponding LiDAR readings to initialize the occupancy grid. You can do this easily by initializing the particle filter to have just one particle and simply calling the slam_t.observation_step as shown in main.py.

(f) **(30 points)** You will now run the full SLAM algorithm that performs one dynamics step and observation step at each iteration in the function run_slam in main.py. Make sure to start SLAM only after the time when you have both LiDAR scans and joint readings (the two arrays start at different times). For all 4 datasets, you will plot the final binarized version of the map, (x, y) location of the particle in the particle filter with the largest weight at each time-step and the odometry trajectory (x, y) (in a different color); this counts for 10 points each.

Some Notes This problem is much easier and shorter than it may seem. You should go through these steps carefully and in the suggested order. You should make sure that the results of the previous step are correct before proceeding. The two functions in main.py to check the dynamics and observation step are very important to find bugs. You do not need to implement more than 100 lines of code.

Problem 3 (Swinging up an Acrobot, 30 points). The Acrobot is a classical under-actuated dynamical system. It is a model for a gymnast swinging up on a bar.

The gymnast only applies control input at his waist. He begins from his initial position of hanging down from the bar and wants to end up at a terminal position where he is upright. Note that the vertical position is an equilibrium position, and therefore, the gymnast will stay there indefinitely without any control input (https://www.youtube.com/watch?v=O2b03YtMeRU). In this problem, you will write an LQR controller to take the Acrobot from the initial condition to the terminal condition. It is difficult to directly use LQR for such a highly nonlinear system. We will therefore use something called as an energy shaping controller to first swing up the Acrobot to the near vertical position and then switch on the LQR controller which will keep the robot in the upright position (which is the equilibrium point).



The equations of motion for an Acrobot are as follows. You can read http://underactuated.mit.edu/acrobot.html to see how they are derived by writing down the Lagrangian. Let $q(t) = [q_1(t), q_2(t)]$ be the joint angles.

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = Bu \tag{2}$$

where

$$M(q) = \begin{bmatrix} I_1 + I_2 + m_2 l_1^2 + m_2 l_1 l_2 \cos q_2 & \frac{1}{2} I_2 + m_2 l_1 l_2 \cos q_2 \\ I_2 + \frac{1}{2} m_2 l_1 l_2 \cos q_2 & I_2 \end{bmatrix}$$

$$C(q, \dot{q}) = \begin{bmatrix} -m_2 l_1 l_2 \sin q_2 \dot{q}_2 & -\frac{1}{2} m_2 l_1 l_2 \sin q_2 \dot{q}_2 \\ \frac{1}{2} m_2 l_1 l_2 \sin q_2 \dot{q}_1 & 0 \end{bmatrix}$$

$$G(q) = \begin{bmatrix} (\frac{1}{2} m_1 l_1 g + m_2 l_1 g) \sin q_1 + \frac{1}{2} m_2 l_2 g \sin (q_1 + q_2) \\ \frac{1}{2} m_2 l_2 g \sin (q_1 + q_2) \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

(a) Linearize the dynamics and implement LQR (10 points) For the second order dynamics, define the state $x \equiv [q, \dot{q}] \in \mathbb{R}^4$. We will first linearize (2) around the vertical position $x_f = [\pi, 0, 0, 0]^\top$. This involves defining a new state

$$\delta x = x - x_f$$

and writing down (2) in terms of the new state δx.

$$\delta \dot{x} = A \delta x + Bu.$$

Write down your A and B matrices and how you calculated them. You can use Mathematica or Matlab or Sympy (show the code if you use these softwares) to take derivatives if you'd like but doing it by hand is quicker and easier; just make sure to double check your calculations. You can verify your linearized dynamics using finite differences. That's:

$$\frac{\partial f}{\partial x_i}(x_f) \approx \frac{f(x_f + \epsilon \mathbf{1}_i) - f(x_f)}{\epsilon}$$

for some small ϵ.

You will now choose matrices Q and R to set the cost function to be

$$\frac{1}{2}\int_0^\infty \mathrm{d}t\, \delta x(t)^\top Q \delta x(t) + u(t)^\top R u(t).$$

We know that for infinite-horizon LQR, the optimal cost-to-go is quadratic in the state, i.e., $V(\delta x) = \delta x^\top P \delta x$. The matrix P can be obtained by solving a Riccati equation:

$$A^\top P + PA - PBR^{-1}B^\top P + Q = 0$$

and the corresponding optimal feedback control is given by

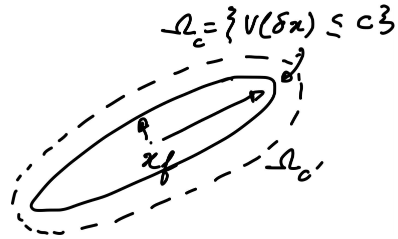
$$u = -K\delta x$$

where the Kalman gain is $K = R^{-1}B^\top P$.

In the code that we have provided, you will fill in the linearized dynamics in *get_lqr_term* and the LQR controller in *get_control_efforts*. You should use *scipy.linalg.solve_continuous_are* to solve the Riccati equation.

To check your controller, you should try a few initial conditions close to the vertical position, e.g., $x_0 = [\pi - 0.1, 0, 0.2, 0]$. Your controller should be able to balance the Acrobot to the equilibrium point.

(b) Check initial conditions for which LQR works (5 points) If you try a few others, you will notice that the LQR controller is not able to stabilize the robot. Argue why or why not. There is however a neat trick that we can use to calculate the set of initial conditions that LQR works for. Notice that the value function is $V(\delta x) = \delta x^\top P \delta x$. For write down the P matrix that you have calculated in the previous step and show the eigenvalues of this matrix. You will notice that the eigenvalues are not all equal, some are very large while some are rather small. This indicates that the level set, i.e., the set of states with small cost to go according to LQR looks like an ellipse with a large aspect ratio:



If we choose some small value of c, then we can think of the states in the set

$$\Omega_c = \left\{ x : \delta x^\top P \delta x \leq c \right\}$$

as the set of states that LQR works for. Use a small value of c (there is neat way to choose this if you think using the eigenvalues of P) and draw a plot of $q_1(t)$ vs $q_2(t)$ for at least two trajectories of the Acrobot, one that starts inside this set and one that starts outside this set.

(c) Energy shaping controller (10 points) LQR can balance the robot once it is close to the terminal condition. We will now use a different controller to swing up the robot from its starting state $x_0 = [0, 0, 0, 0]$, which is at the bottom.

If we have a fully-actuated system like $\ddot{q} = u$, it would be easy to stabilize it with feedback control, e.g., $\ddot{q} = u = k_p q + k_d \dot{q}$ with negative $k_p, k_d < 0$. The Acrobot is not a fully-actuated system: there is only one control input which is the torque at the waist and there are two degrees of freedom. But we can rewrite the system in a special way using a technique called partial feedback linearization. Here is how it works.

$$
\begin{aligned}
M_{11}\ddot{q}_1 + M_{12}\ddot{q}_2 &= \tau_1 \\
M_{21}\ddot{q}_1 + M_{22}\ddot{q}_2 &= \tau_2 + u
\end{aligned}
\tag{3}
$$

where $\tau = [\tau_1, \tau_2]^\top \equiv -C\dot{q} - G$ from (2). Observe that this is a linear system in the two variables $[\ddot{q}_1, \ddot{q}_2]$ and we can solve this system to get

$$
\begin{aligned}
\ddot{q}_1 &= \text{func}_1(q, \dot{q}, u) \\
\ddot{q}_2 &= \text{func}_2(q, \dot{q}, u).
\end{aligned}
$$

In our problem, the torque directly moves the second angle q_2 (the first angle q_1 is actuated only indirectly by the torque due to conversation of angular momentum). We therefore will use the second equation to obtain a relationship

$$
u = \text{func}_3(\ddot{q}_2, q, \dot{q}). \tag{4}
$$

Write down what your func_3 is and explain in detail how you found it.

This is pretty neat because (4) tells us what control u we should use if we want a particular $q, \dot{q}$ and $\ddot{q}_2$. We will now use this calculation to calculate a particular feedback control, the one that consistently adds energy to the robot and makes it swing up.

The Acrobot has the smallest energy (sum of potential energy and kinetic energy due to the angular velocity) when it is at the bottom. When it is stationary at the top, it only has potential energy which is larger than the energy at the bottom. If the Acrobot is to swing up, we need to add at least the difference between the two into the system. The basic idea of an energy shaping controller is to realize that just like you push down with your legs when you swing up on a swing in the park, i.e., you push towards the direction of motion instead of against it, we can calculate a control input that adds energy by applying a torque in the direction of motion.

$$
\begin{aligned}
\tilde{E} &= E(x) - E(x_f) \\
\bar{u} &= \dot{q}_1 \tilde{E} \\
\ddot{q}_2^d &= -k_1 q_2 - k_2 \dot{q}_2 + k_3 \bar{u} \\
u &= \text{func}_3(\ddot{q}_2^d, q, \dot{q})
\end{aligned}
\tag{5}
$$

In the above equation we have chosen a desired angular acceleration $\ddot{q}_2^d$ to calculate the actual control u, but this desired $\ddot{q}_2^d$ is calculated in such a way that as the gap between $E(x)$ and $E(x_f)$ goes to zero, the second angle q_2 forms a stable system by itself (for $k_1, k_3 > 0$), i.e., $q_2 \to 0$ if $\bar{u} = 0$ from any initial condition.

The most important point of the above equations is that $\bar{u} = \dot{q}_1 \tilde{E}$; this is the energy shaping part. In simple words, $\bar{u}$ is proportional to how much energy should be added/subtracted (depending upon $\tilde{E}$) at each time instant. This "effective control" leads to an angular acceleration $\ddot{q}_2^d$, which gives us the actual control u that we should take using the final equation. In the third equation, we have also added feedback in the form of $-k_1 q_2 - k_2 \dot{q}_2$ which makes sure that if q_2 is far from zero, we choose a slightly larger $\ddot{q}_2^d$ than what would be dictated by simply the effective control term $k_3 \bar{u}$. You should choose the values of $k_1, k_2, k_3 > 0$ yourself. Complete the *get_swingup_input* function in the code that we have provided. **When you implement all these formulae, do not forget to convert q back to $[-\pi, \pi]$ using numpy.unwrap every time you use it. This is very important.**

(d) Run the entire system (5 points) Plot the joint angles and the control input for when the Acrobot starts from the stationary state at the bottom, uses energy shaping to reach a state that lies within the ellipse of initial conditions that are suitable for LQR and then uses LQR to stabilize in the upright position. You find the code provided useful for animating the motion of the Acrobot for both debugging and final plotting.