CS 287, Fall 2009
Problem Set #1 : Optimal Control

Deliverable: 3-6 page pdf write-up by Monday October 26, 23:59pm.

NOTE: Please refer to the class webpage for the homework policy.

For all questions, the starter files provide a code skeleton. Dumb placeholders have been put in place where your code is supposed to go. The "main" files should run without errors (if not, please let me know so I can fix it!), the results will just be meaningless until you did your part.

When making your write-up, make sure to include and discuss plots (and if helpful, snippets of code) which are helpful in demonstrating that your system works and in answering the questions. However, there is no need to include all plots being generated, similarly for code you write. (In fact, including all on both accounts would make for a pretty poor write-up.) Your writing should be self-contained, though it can reference other material (just like a paper would).

For most problems possible extensions are listed. You are free to choose which ones you would like to complete. The extensions have asterisks next to them indicating my estimate of their difficulty/time-consumingness. Aside from solving the basic problems, you should complete a total of 6 stars over the assignment as a whole.

1. Discretization and Value Iteration on the Double Integrator

In this problem you will investigate various properties of discretization methods for solving continuous optimal control problems. To enable convenient visualization (and to limit computational power required) we consider a 2D problem: the double integrator (aka brick of unit mass on ice), which has the following dynamics:

$$\ddot{x} = u$$

To eliminate the possibility of having to attribute certain performance characteristics to numerical integration inaccuracies, we consider the above dynamical equations integrated with Euler integration at 100Hz as our "ground-truth" dynamical system on which all our control algorithms have to act. I.e., we consider the system:

$$\begin{aligned} x_{k+1} &= x_k + \dot{x}_k \ \delta t \\ \dot{x}_{k+1} &= \dot{x}_k + u_k \ \delta t \end{aligned}$$

with $\delta t = .01$.

We use a discrete time quadratic cost function which favors the brick resting at the origin:

$$g(x, u) = (x^\top Q x + u^\top R u)$$

and the optimal control objective is to minimize:

$$\sum_{k=0}^{\infty} \gamma^k g(x_k, u_k).$$

In your code, you will use some very large horizon (rather than ∞).

You will compare the performance of three optimal control algorithms: (a) LQR, which provides the optimal solution and is our basis to evaluate the other two algorithms, (b) Value iteration based upon a nearest neighbor discretization, (c) Value iteration based upon a piecewise linear approximation over a triangulation.

Your first task is to implement LQR, value iteration, the execution of LQR, and the execution of the greedy policy w.r.t. a value function. Study the resulting plots, discuss your observations.

If your implementation is similar to mine, you might notice the nearest neighbor model is doing particularly poorly. Can you find an explanation? [Hint: inspect the actions it chooses and inspect its transition models.]

Do you see any effects of the finite action resolution? Refine the set of actions being chosen from at execution time. [This can be done without increasing the resolution at value iteration time.]

Each of the following improvements are possible extensions:

1. (**) Implement a fix for the nearest neighbor model performing so poorly? [Hint: only allow discrete transitions to happen when a new discretized state is reached to avoid the stickiness property of discretization through nearest neighbors. This means transitions will take a varying number of discrete time steps, i.e., we obtain a semi-MDP. The semi-MDP can be transformed into an MDP—which enables you to keep using the same code for value iteration. Make sure to make the according change when acting greedily w.r.t. the value function!] To what extent does the same change affect the triangulation based discretization results?

2. (**) Implement roll-out. In roll-out, one assumes one already has a policy π available—in our case the policy π is the greedy policy w.r.t. the value function which you have been using in the discretization schemes so far. One is guaranteed to do at least as well as the policy π, and often better, by following the roll-out procedure: Every time one decides upon which action to execute at that particular time, one scores each possible action using the following procedure: evaluate the sum of costs obtained when taking that action at the current time followed by executing the policy π for all future times.

3. (****) Develop a sufficiently fast implementation that enables solving 4D problems such as cartpole and acrobot. In particular, consider the problem of minimum time swing-up with bounds on the control inputs and bounds on the space being used (in case of cartpole). Note that minimum time problems tend to have bang-bang solutions (optimality can be achieved by only using the max and the min control input)—this simplifies the discretization of the control inputs. [Unlikely to work within matlab.]

You might want to augment the policy evaluation plot (in main.m) to include additional graphs corresponding to your extensions.

2. Energy Pumping Swing-up and LQR Stabilization of the Cartpole

Implement LQR for stabilizing the cartpole around the upright position of the pole and around the zero x coordinate. Lyapunov's linearization method suggests that the LQR controller will also (locally) stabilize the non-linear system. Try various different starting conditions and empirically find a region from which it seems plausible LQR would work for stabilization around the top. Try a couple of choices of Q and R and see how/if it affects the stability region (or at least stability for the gridpoints you sampled within the region you consider).

If the LQR controller can stabilize the cartpole from being 10m off to the right—then it could do so from any distance, as long as it would "imagine" the error was only 10m. Incorporate this idea. (10m might not be the right choice though!) Think about whether something similar is the case for other state variables? Describe what property of the dynamics model makes this hold true.

Implement energy pumping for swing-up (Tedrake lecture notes Section 3.6.3.) and make the control policy transition from energy pumping to LQR once an appropriate point is reached. You should use the provided cartpole simulator—this means you will have to re-work through the energy equations because the notes assume all physical constants are equal to one.

3. Trajectory Stabilization for Helicopter Flight

1. Implement LQR to find a linear feedback controller that can stabilize the helicopter around hover. Test how far you can push it in various axes (position, orientation) before it fails, i.e., leaves the regime where this LQR controller could do the job. Are there any state variables for which a similar trick can be played as in the cartpole stabilization of the previous question?
2. Implement LQR for trajectory following to find a time varying linear controller that can make the helicopter follow the provided target trajectory—which consists of hover, forward flight, hover.

Note: I updated the posted slides for Lecture 7 with a slide which works through some of the details of casting iterative LQR for trajectory following into the standard linear quadratic format. For the particular trajectory you are asked to design the controller for, it should suffice to run a single iteration only—where your linearization is around the target at each time t.

Some of your extension choices are:

1. (*) Investigate through simulation how the performance of your controller degrades with the introduction of: (a) latency, (b) noise on the measurements, (c) noise on the dynamics, (d) wind (=you can roughly model this with a constant force in a certain direction), (e) mismatches between the dynamics used for control design and the dynamics used for test flying (e.g., mass properties, scaling of the effect of the controls, drag properties). You would probably want to do this for the hover setting.
2. (*) Through considering the closed-loop poles investigate how the stability properties of the closed-loop system are affected by some of the forementioned influences. How about your choice of Q and R? You would probably want to do this for the hover setting.

4. Realtime Recurrent Learning (RTRL) for Policy Optimization

In this problem you will implement RTRL to solve for a policy (parameterized by some vector α) that will achieve the cart-pole swing-up by minimizing a cost function of the form:

$$\int_{t=0}^{T} g(x(t), u(t))dt + h(x(T)),$$

with the specifics of g and h defined in the starter code.

You will parameterize your policy strictly by time, i.e.,

$$u[i] = \alpha_i$$

The Tedrake lecture notes, Section 9.3, provide a detailed description of RTRL.

You should be able to converge to a solution that gets the pendulum sufficiently close to the top (to within 5 degrees). No need to incorporate catching the pendulum with an LQR stabilizing controller.

1. (**) In your current setup it merely tries to swing the pole within some vicinity of the target state. Include stabilization at the top and make the swing-up optimization account for the stabilization at the top. You could do this as follows: incorporate an LQR based controller for stabilization (make sure it works for $-\pi$ and $+\pi$) into the control function control_utape_zoh.m. Make the control function then return a control input which is a weighted combination of (a) The open loop control for that time (which it does in the form you received it), (b) The control input computed with LQR. You could have the weight be such that the influence of the LQR controller quickly fades with distance from the target. [Note: you will also have to change the derivatives control_utape_zoh.m outputs.]

2. (*) The current scheme tends to find solutions with very large control inputs. Can you have it find a swing-up sequence with significantly smaller control inputs? [You could try to incorporate constraints on the controls, or more heavily penalize for the control inputs. This might require more swings and hence might be more susceptible to good initialization.]

3. (*) Change your code to use Matlab's "ode45" differential equation solver instead of Euler integration. Does the solution found differ? Why? Does it work to use the control tape found with the Euler integration based optimization with "ode45" simulation and vice versa?

Other possible extensions:

1. (**) Application of one of the foredescribed exercises to the acrobot or to a dynamical system of your choice that has my approval of being sufficiently interesting.

2. (**) Prove or disprove the input-state linearizability of the cartpole or acrobot. (You might want to consult Theorem 6.2, p.238 in Slotine and Li.) If input-state linearizable, design a controller using linear control design techniques for an appropriate task of your choice and discuss its performance.

5. Feedback (Voluntary, no credit involved)

Towards improving this assignment for next year: if you had one or two constructive suggestions, that would be helpful. E.g., in question X it would have been nice had you provided the code for Y b/c it took a really long time compared to the amount of learning involved; in question Z it would have been a great learning experience if we had to do W ourselves instead of being given the code; it would be great if the lecture slides had a bit more detail on V.