6.832: Problem Set #2

Due on Wednesday, February 28, 2018 at 17:00. See course website for submission details. Use Drake release tag `drake-20180220`, i.e. use this notebook via `./docker_run_notebook.sh drake-20180220 .`, or whichever script you need for your platform.

About this problem set

This problem set will entirely live inside this jupyter notebook.

Grades will be assigned based on three components:

- **Manually graded free-response questions** -- the TAs will manually assign grades to your answers to short answer responses. You can write inline responses using Markdown (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet) with inline LaTeX -- double-click on any problem writeup to see some examples. Double-click response areas to edit them, and press Control-Enter to finish editing them.
- **Automated code testing** -- we will run automated tests against specific functions (see more details when we introduce the first coding test).
- **Quick code review** -- we will perform a quick manual check to make sure you have actually implemented the functions correctly (as opposed to hacked the unit tests to pass!).

The automated coding tests are pretty small in this problem set, but we are planning to move more towards this framework in the next three problem sets. We would love to hear feedback from you on how this testing setup works.

---

1. Cost Functions

In this problem we will explore how to design cost functions that make the robot exhibit the kind of behavior we want. For this, we will consider the Dubins car model, which is a very simple model of a vehicle given by the following equations:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix}, \qquad \dot{\mathbf{x}} = f(\mathbf{x}, u) = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} -\sin\psi \\ \cos\psi \\ u \end{bmatrix},$$

where $\mathbf{x}$ is the state of the system and consists of the states x (the x-position), y (the y-position) and ψ (the yaw angle of the vehicle).

The only control input is the steering angle u, which directly controls $\dot{\psi}$, while the car drives at constant velocity. (At $\psi = 0$, the vehicle drives in the $+y$ direction.) u has limits $[-u_{max}, +u_{max}]$.

Note that this is a very simple model -- it does not follow Newtonian physics, and can instead just instantaneously choose its yaw rate. It is however a simple model that has seen some use for actual robot research, for both UAVs and cars. For example, here's some relevant results on Google Scholar. (https://scholar.google.com/scholar?hl=en&as_sdt=0%2C22&q=dubins+path+planning&btnG=) To give us a little visual, here's a figure from a paper that uses a Dubins model for a 2D UAV in an environment with obstacles Barry et al., 2012 (http://groups.csail.mit.edu/robotics-center/public_papers/Barry12.pdf):

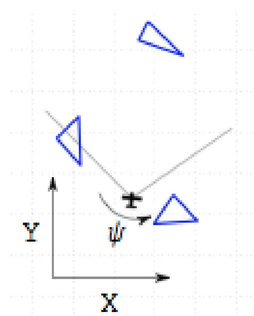


Figure: A dubins vehicle, navigating amonst polygonal obstacles.

Question 1.1 (2 points)

In general it is useful to have cost functions that penalize both the states along a trajectory, and also the final state:

$$J = \mathbf{x}(t_f)^T Q_f \mathbf{x}(t_f) + \int_0^{t_f} g(\mathbf{x}(t), \mathbf{u}(t))\, dt$$

In this question let's look only at cost functions that only involve the final state. A simple form for a cost function that penalizes only the final state of the robot is

$$J = \mathbf{x}(t_f)^T Q_f \mathbf{x}(t_f)$$

where t_f is the final time and Q_f is a symmetric positive semidefinite matrix of the appropriate size. Suppose we want the robot to end up with its yaw angle close to 0, but do not care about the final x and y positions. What should we choose Q_f to be (remember to make sure it is symmetric and positive semidefinite)?

In [2]:

```
'''
Autograded answer for 1.1

Please implement this function so that the return argument
satisfies the specification above.

Scroll to the end of the notebook to execute all tests for this notebook.
'''
import numpy as np

def get_Q_f_problem_1_1():
    Qf = np.zeros((3, 3))
    # YOUR CODE HERE TO SET Qf
    return Qf

print get_Q_f_problem_1_1()
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Short answer explanation for 1.1. Provide a brief justification of your choice of Q_f in this cell.

YOUR ANSWER HERE

Question 1.2 (2 points)

Now suppose we want the vehicle to end up close to the line $y = 0.5x$, but we do not care exactly where on this line and what yaw angle it ends up in. What should we choose Q_f to be (remember to make sure it is symmetric and positive semidefinite)?

In [2]:
```
'''
Autograded answer for 1.2

Please implement this function so that the return argument
satisfies the specification above.
'''

def get_Q_f_problem_1_2():
    Qf = np.zeros((3, 3))
    # YOUR CODE HERE TO SET Qf
    return Qf

print get_Q_f_problem_1_2()
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Short answer explanation for 1.2. Provide a brief justification of your choice of Q_f in this cell.

YOUR ANSWER HERE

Question 1.3 (3 points)

Now suppose we want to end up close to the curve $y = x^2$, and again do not care about the final yaw angle or where exactly on this curve we end up. Why is it not possible to set Q_f to achieve this?

Short answer explanation for 1.3. Please put your answer to this question in this cell.

YOUR ANSWER HERE

2. Optimal Control via HJB

Consider the scalar equation

$$\dot{x} = -4x + 2u,$$

and the infinite horizon cost function

$$J = \int_0^\infty [32x^2 + u^2] dt.$$

Question 2.1 (5 points)

Assume that the optimal cost-to-go function is of the form $J^\star = px^2$. What value of p satisfies the Hamilton-Jacobi-Bellman conditions for optimality?

Written answer explanation for 2.1. A proper answer to this question involves writing out more than a couple expressions. We recommend working through this problem on pencil and paper first. Please then write your answer below in this cell, using LaTeX-style math expressions.

YOUR ANSWER HERE

$$p =?$$

Question 2.2 (3 points)

Given that the optimal feedback controller associated with $J^\star$ is $u^\star = -Kx$, what is the value of K?

Written answer explanation for 2.2. We recommend working through this problem on pencil and paper first. Please then write your answer below in this cell, using LaTeX-style math expressions.

YOUR ANSWER HERE

$$K =?$$

3. Typing in the Optimal Controller for the Double Integrator

In this problem, we'll consider the optimal control problem for the 1-dimensional input-constrained double integrator described by

$$\ddot{q} = u, \; |u| \leq 1$$

Figure: Simple drawing of a double integrator

As you may remember, a simple way to think of the double integrator is as a system which has direct control over its acceleration, like a brick sliding around on ice, with a rocket booster attached for control. While a double integrator (especially in just 1 dimension) is a very simple system, double or triple integrators (which control jerk, the derivative of acceleration) can be for example used as simple models (see Fig. 2) to approximate the dynamics of quadrotors (http://groups.csail.mit.edu/robotics-center/public_papers/Florence16.pdf)) for performing (http://savefrom.r url=https%3A%2 obstacle avoidance and planning paths (https://www.youtube.com/watch?v=9a0eEscz1Cs&t=1s) chrome&utm_me

We won't often be able to simply type in an optimal controller, but for the double integrator, it's not too hard!

For this question, we want the optimal controller for a minimum-time cost function. Remember that for "minimum-time" problems, we just have the cost function:

$$J = \int_0^\infty g(x(t), u(t)) \, dt$$

for just:

$$g(x, u) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{otherwise} \end{cases}$$

Which gives us:

$$J = t_f$$

Where t_f is the time needed to reach the desired state, 0. So our cost function is just the time needed to reach 0.

Question 3.1 (4 points)

Implement the optimal controller, and the optimal cost-to-go function, for the double integrator in the space below. (May help to look in the course notes at Example 8.2)

The point of this question isn't to practice our ability to implement algorithms, but instead to give us some data and experience to later make some interesting comparisons and analyses.

Also you can rest well knowing that no-one will ever have a "more optimal" controller than you! (But actually, this maybe isn't the best controller. We'll discuss why.)

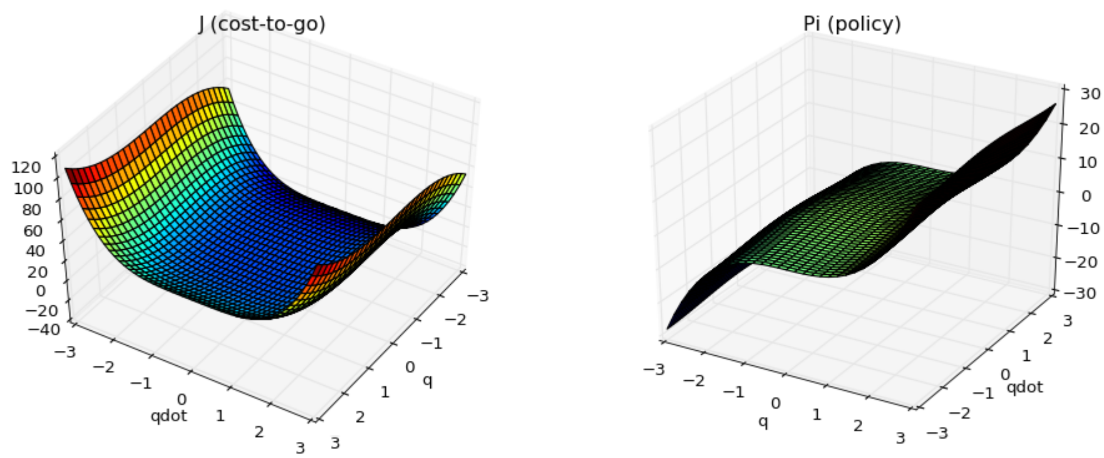
In [3]:
```
'''
Autograded answer for 3.1

Please implement this function so that the return argument
satisfies the specification above.
'''

def get_optimal_time_to_go_problem_3_1(q, qdot):
    t = q**3 + qdot**4 # bogus answer to make plots pretty
    # YOUR CODE HERE TO SET t
    return t

def get_optimal_control_problem_3_1(q, qdot):
    u = q**3+np.cos(qdot) # bogus answer to make plots pretty
    # YOUR CODE HERE TO SET u
    # Remember that the problem statement imposes constraints on u!
    return u
```

Plotting help for Question 3.1

We've written a little plotting code for you to visualize this policy:

In [11]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.animation as animation

num_q_bins = 31
qbins = np.linspace(-3., 3., num_q_bins)
num_qdot_bins = 51
qdotbins = np.linspace(-3., 3., num_qdot_bins)
state_grid = [set(qbins), set(qdotbins)]

[Q, Qdot] = np.meshgrid(qbins, qdotbins)

fig_optimal, (ax_J, ax_Pi) = plt.subplots(1, 2, figsize=(12,5))
ax_J.axis('off')
ax_Pi.axis('off')

ax_J = fig_optimal.add_subplot(121, projection='3d')
ax_J.set_title('J (cost-to-go)')
ax_J.set_xlabel("q")
ax_J.set_ylabel("qdot")

ax_Pi = fig_optimal.add_subplot(122, projection='3d')
ax_Pi.set_title('Pi (policy)')
ax_Pi.set_xlabel("q")
ax_Pi.set_ylabel("qdot")
plt.tight_layout()

J_optimal = np.zeros((num_qdot_bins, num_q_bins))
for i, row in enumerate(J_optimal):
    for j, val in enumerate(row):
        J_optimal[i,j] = get_optimal_time_to_go_problem_3_1(Q[i,j], Qdot[i,j])

J_surface = [ax_J.plot_surface(Q, Qdot, J_optimal, rstride=1, cstride=1,
                       cmap=cm.jet)]

Pi_optimal = np.zeros((num_qdot_bins, num_q_bins))
Pi_optimal = np.zeros((num_qdot_bins, num_q_bins))
for i, row in enumerate(Pi_optimal):
    for j, val in enumerate(row):
        Pi_optimal[i,j] = get_optimal_control_problem_3_1(Q[i,j], Qdot[i,j])

Pi_surface = [ax_Pi.plot_surface(Q, Qdot, Pi_optimal, rstride=1, cstride=1,
                        cmap=cm.jet)]
plt.show()
```



Note that the plotting code, for both J and Pi, is just based on a 3D mesh over a `np.meshgrid` of Q and Qdot.

We recommend printing out these inputs to the plots, and practice manipulating the data of the plots.

In [25]:
```
# Some examples of interacting with the data generated above
print qbins.shape
print qdotbins.shape
print Q.shape
print Qdot.shape
print J_optimal.shape
print Pi_optimal.shape

terrible_control_choice = -Pi_optimal
qbin_sample = 10
qdotbin_sample = 3
print terrible_control_choice[qdotbin_sample][qbin_sample]

q = qbins[qbin_sample]
qdot = qdotbins[qdotbin_sample]
print -get_optimal_control_problem_3_1(q,qdot)

# Recommend investigating more the shapes of some of these and how they relate
```

```
(31,)
(51,)
(51, 31)
(51, 31)
(51, 31)
(51, 31)
1.87681789039
1.87681789039
```

Question 3.2 (1 point)

For a real robot, why would the minimum-time solution not necessarily be the "best" thing to do?

Written answer explanation for 3.2.

YOUR ANSWER HERE

4. Value Iteration solution for the Double Integrator

Now let's approach the same problem, but solve it using the Value Iteration algorithm.

An implementation of that algorithm is available for you in Drake. This is a complete implementation of the algorithm with discrete actions and volumetric interpolation over state.

In [10]:
```
import math
import numpy as np

from pydrake.systems.framework import VectorSystem
from pydrake.systems.analysis import Simulator
from pydrake.systems.controllers import (
    DynamicProgrammingOptions, FittedValueIteration)

# This system block implements the dynamics
# of a double integrator. It takes one control
# input, has two states (1D position and velocity),
# and copies its current state as its output.
class DoubleIntegrator(VectorSystem):
    def __init__(self):
        # One input, one output, two state variables.
        VectorSystem.__init__(self, 1, 1)
        self._DeclareContinuousState(2)

    # qddot(t) = u(t)
    def _DoCalcVectorTimeDerivatives(self, context, u, x, xdot):
        xdot[0] = x[1]
        xdot[1] = u

    # y(t) = x(t)
    def _DoCalcVectorOutput(self, context, u, x, y):
        y[:] = x

# Set up a simulation of this system.
plant = DoubleIntegrator()
simulator = Simulator(plant)
options = DynamicProgrammingOptions()

# This function evaluates a minimum time
# running cost, given a context (which contains
# information about the current system state).
def min_time_cost(context):
    x = context.get_continuous_state_vector().CopyToVector()
    if x.dot(x) < .05:
        return 0.
    return 1.

# This function evaluates a running cost
# that penalizes distance from the origin,
# as well as control effort.
def quadratic_regulator_cost(context):
    x = context.get_continuous_state_vector().CopyToVector()
    u = plant.EvalVectorInput(context, 0).CopyToVector()
    return 2*x.dot(x) + 10*u.dot(u)

# Pick your cost here...
cost_function = min_time_cost
#cost_function = quadratic_regulator_cost

# This sets up the mesh of sample points over
# which we'll run the value iteration algorithm.
num_q_bins = 31
```

```
qbins = np.linspace(-3., 3., num_q_bins)
num_qdot_bins = 51
qdotbins = np.linspace(-3., 3., num_qdot_bins)
state_grid = [set(qbins), set(qdotbins)]

input_limit = 1.
input_grid = [set(np.linspace(-input_limit, input_limit, 9))]
timestep = 0.01

[Q, Qdot] = np.meshgrid(qbins, qdotbins)

# Recommend not increasing the max_iterations too much.  Another factor of 10
 should be okay, if you want to wait.
max_iterations = 10000
J_iterations  = np.zeros((num_qdot_bins, num_q_bins, max_iterations))
Pi_iterations = np.zeros((num_qdot_bins, num_q_bins, max_iterations))

num_iterations = 0
def add_iteration_solve(iteration, mesh, cost_to_go, policy):
    global num_iterations, J_iterations, Pi_iterations
    num_iterations += 1
    if num_iterations >= max_iterations:
        raise RuntimeError("Solution did not converge within "+str(max_iterati
ons)+" iterations.")

    J = np.reshape(cost_to_go, Q.shape)
    J_iterations[:,:,iteration-1] = J    # "first" iteration here is "1", so h
ere we 0-order it
    Pi = np.reshape(policy, Q.shape)
    Pi_iterations[:,:,iteration-1] = Pi

options.visualization_callback = add_iteration_solve

# Run value iteration on this mesh using
# the cost function chosen above.
policy, cost_to_go = FittedValueIteration(simulator, cost_function,
                                          state_grid, input_grid,
                                          timestep, options)

# trim array of iterations
J_iterations  = J_iterations[:,:,:num_iterations]
Pi_iterations = Pi_iterations[:,:,:num_iterations]

print "Done solving.  Converged in "+str(num_iterations)+" iterations."
```

```
Done solving.  Converged in 1067 iterations.
```

Similar to before, it's useful to manipulate the data used to generate the plots.

Unlike with the analytical optimal policy, where we only had one mesh over q and $\dot{q}$, this time we have saved the solution state from each solve. Let's print out the shape of `Pi_iterations`. Here we've set it up so that indexing into the last dimension gives us the solver at that iteration's state.

In [11]:
```
print Pi_iterations.shape

policy_on_10th_iteration = Pi_iterations[:,:,10]

final_computed_policy    = Pi_iterations[:,:,-1]
```

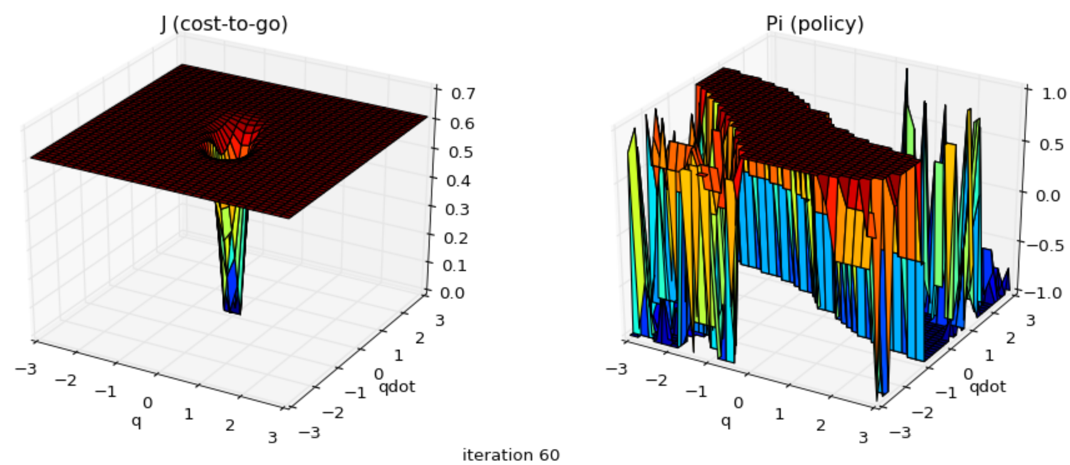
```
(51, 31, 1067)
```

Now plot the result of Value Iteration

The above code cell already solved the value iteration code, and saved its iterative solution on each iteration.

Now below, we plot this solution over time.

In [12]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.animation as animation

fig, (ax, ax2) = plt.subplots(1, 2, figsize=(12,5))
ax.axis('off')
ax2.axis('off')

ax = fig.add_subplot(121, projection='3d')
ax.set_title('J (cost-to-go)')
ax.set_xlabel("q")
ax.set_ylabel("qdot")

ax2 = fig.add_subplot(122, projection='3d')
ax2.set_title('Pi (policy)')
ax2.set_xlabel("q")
ax2.set_ylabel("qdot")
plt.tight_layout()

J_initial = J_iterations[:,:,0]
J_surface = [ax.plot_surface(Q, Qdot, J_initial, rstride=1, cstride=1,
                       cmap=cm.jet)]

Pi_initial = Pi_iterations[:,:,0]
Pi_surface = [ax2.plot_surface(Q, Qdot, Pi_initial, rstride=1, cstride=1,
                        cmap=cm.jet)]

def update_plots(iteration, J_iterations, J_surface, Pi_iterations, Pi_surface
):
    if iteration % 10 != 0:
        return # only plot every 10th
    J_surface[0].remove()
    J_surface[0] = ax.plot_surface(Q, Qdot, J_iterations[:,:,iteration], rstri
de=1, cstride=1,
                        cmap=cm.jet)

    for txt in fig.texts:
        txt.set_visible(False)
    fig.text(0.5, 0.04, 'iteration '+str(iteration), ha='center', va='center')

    Pi_surface[0].remove()
    Pi_surface[0] = ax2.plot_surface(Q, Qdot, Pi_iterations[:,:,iteration], rs
tride=1, cstride=1,
                        cmap=cm.jet)


animate = animation.FuncAnimation(fig, update_plots, num_iterations, interval=
1, fargs=(J_iterations, J_surface,
                                                                              Pi
_iterations, Pi_surface))
```



Tips

- while the animation is running, you can click + drag to change the 3D viewpoint
- pressing the "Power" button for the plot above will stop the animation, which is probably making your computer run pretty hard if you have it running
- restarting your jupyter notebook's kernel (the restart circle button up top in jupyter notebook) will also completely free up your system (but also lose all variable state)

Run the value iteration code for the double integrator to compute the optimal policy and optimal cost-to-go for the minimum-time problem.

Compare the result to the analytical solution we found in lecture, and programmed in Question 3.1 by answering the following questions.

Question 4.1 (5 points)

Find an initial condition of the form $(2, \dot{q}_0)$ such that the value iteration policy takes an action in exactly the wrong direction from the true optimal policy.

a) What value of $\dot{q}_0$ did you find that disagrees?

b) What is the true optimal time-to-go from this state (i.e., for the optimal bang-bang controller derived in class)?

c) What is the time-to-go estimated by value iteration?

d) Why are these values different?

Written answer explanation for 4.1. Bonus points if you generate plots that help explain your answer. (Feel free to add code cells, in addition to this markdown cell.)

a) YOUR ANSWER HERE

b) YOUR ANSWER HERE

c) YOUR ANSWER HERE

d) YOUR ANSWER HERE

Question 4.2 (3 points)

When implementing value iteration, one needs to be wary of several implementation details.

Find a setting of the discretization (i.e., the variables `num_q_bins` and `num_qdot_bins`) that causes the code to NOT converge in 10,000 iterations.

The underlying implementation (https://github.com/RobotLocomotion/drake/blob/master/systems/controllers/dynamic_programming.cc) of value iteration terminates after performing an update to J *(following the discrete value iteration update described in class) and finding that the update did not change J* by more than $\epsilon = 0.0001$ at any point.

Why doesn't the setting you discovered converge?

Note: The maximum distance between points in the q and $\dot{q}$ directions should still be at most 0.2, and the grid must still contain the square with sides of length 2 centered about the origin.

Short answer for 4.2.

YOUR ANSWER HERE

Question 4.3 (2 points)

We may have noticed some issues for the value iteration solution, compared to what we know is the optimal solution. But it's worth taking a moment to consider the pros and cons of each method we've tried so far:

a) What is at least one reason that the value iteration algorithm is powerful, compared to the analytical solution?

b) On the other hand, what is nice about the analytical solution?

Short answers for 4.3.

a) YOUR ANSWER HERE

b) YOUR ANSWER HERE

Question 4.4 (2 points)

Switch the cost function for the double integrator so that we use a quadratic cost on both: control input, and state. This has already been implemented for you as: `quadratic_regulator_cost()`.

How does the minimum-time solution of value iteration compare to the quadratic-regulator solution?

Short answer for 4.4.

YOUR ANSWERS HERE

Note: you can still receive full credit if you do not do the bonus questions. But you may be able to get extra credit if you do them.

5. Bonus Question: Value Iteration for the simple pendulum

Bonus 5.1 (2 points)

See if you run the value iteration example for the input-constrained pendulum. Either replicate the setup above we provided for the double integrator, or use the `bash` scripts provided for the docker image, i.e. run:

```
# or use the bash script for your system
./docker_run_bash_linux.sh drake-20180220 .
# now inside docker image
cd /underactuated/src
cd pendulum
python value_iteration.py
```

Question: how does the value-iteration solution compare to other control design methods we have explored for the input-constrained pendulum?

Short answer for 5.1.

YOUR ANSWER HERE

Test your own implementations

Running the cell below will run your implemented functions against unit tests.

Don't change the cell below, or the `test_set_2.py` file. We will grade your implementations against the original files.

Make sure to **SAVE** your notebook before running tests. (File --> Save and Checkpoint, or use the hotkey which should be ctrl+s on linux, cmd+s on osx, etc)

In [1]:
```
import os
# Run the test in a subprocess, to make sure it doesn't open any plots...
os.popen("python test_set_2.py set_2.ipynb test_results.json")

# Print the results json for review
import test_set_2
print test_set_2.pretty_format_json_results("test_results.json")
```

```
Test Problem 1_1: 0.00/2.00.
  * Test Failed: Q_f must be symmetric

Test Problem 1_2: 0.00/2.00.
  * Test Failed: We produced a ton of random samples, and the one that was
  * closest to the line y=0.5x didn't have the smallest cost according
  * to Q_f. Try again!

Test Problem 3_1 Optimal Control Input: 0.00/2.00.
  * Test Failed: The optimal control at [q=0.0,qd=0.0] should be 0.

Test Problem 3_1 Time (Cost) To Go: 0.00/2.00.
  * Test Failed: An optimal time-to-go test case for [q=5.00, qd=15.00]
  * failed.

TOTAL SCORE (automated tests only): 0.00/8.00
```

Note that many of the questions are not auto-graded, but will be graded manually! (Double-check you gave answers for each.)