The first problem involves using value function approximation and linear programming for reinforcement learning. For this problem we will focus on a simple, benchmark problem called the inverted pendulum. If you’ve ever tried balancing a meter stick or a pen on your fingertip, then you are familiar with this problem. At first, the use of the term “pendulum” may seem backwards to you since we often think of the mass of the pendulum as hanging below the pivot point rather than above. This is why it’s called an inverted pendulum. It is also sometimes called the cart-pole problem because you can think of the task as balancing a pole that is upright on a cart.

For purposes of having an easy enough testbed, we will simplify the problem in several ways:

- We will consider only a 1-dimensional version of the problem. For example, you can think of the pendulum as constrained to move only forward and back, but not right and left.
- We will consider a noiseless version of the pendulum where your actions always have exactly the intended effect.
- We will limit the action space to just three discrete actions, doing nothing, adding a constant amount of forward force to the device to which the pendulum is attached, or adding a constant amount of backward force to the device. This is sometimes called a “bang-bang” controller.

Note that the inverted pendulum problem can be solved using a variety of techniques and that reinforcement learning is not the only or even best way to approach this. Even without the simplifying assumptions, it’s still a toy problem. Observe that without noise, the problem would be completely trivial if the pendulum starts in a balanced state since the optimal policy is (you guessed it) “do nothing”. To make things more interesting we will start in a random initial state, and the challenge will be to bring the pendulum into a balanced or nearly balanced state.

The state space for the pendulum is two dimensional, consisting of the angle of the pendulum with respect to upright and the angular velocity of the pendulum. States are represented in Matlab as a row vector, so \[0 \ 0\] represents a perfectly balanced state. For all states in which the pendulum is above horizontal, the reward is zero; a reward of -1 is obtained if the pendulum falls below horizontal, after which the simulation ends and the MDP can be considered to transition to an absorbing state with reward zero. An optimal policy then is one which achieves a perfect value of zero by never allowing the pendulum to fall.

A sample transition consists of a start state, an action (a number between 1 and 3), a reward, and a next state. A function `collect_samples` has been provided to collect sample transitions for you. This function obtains samples in which, at each step, a random action is selected. If the pendulum falls over, it is reset to a random point near the balanced state for the next transition. The actual return value from `collect_samples` is a Matlab structure with fields named states, actions, rewards, and nextstates. For example, if you collect 10 samples by calling `samples = collect_samples(10)`, the result is a Matlab structure named samples, where `samples.states` is a 10 × 2 matrix containing the 10 transition start states, `samples.actions` is a 10 × 1 array containing the actions taken in the 10 transitions, etc.
Features for your ALP are provided in the function `pendulum_basis_rbf`. To obtain a design matrix $\Phi$ containing features for all of your start states, for example, you would call $\Phi = \text{pendulum\_basis\_rbf}($\text{samples\_states}$)$. The features include nine radial basis functions and the constant function.

Finally, we provide some infrastructure for collecting data and testing your policy. The script collects `pendulumSampleTrainTest` 2000 samples, provides them to your ALP function, and then uses the resulting approximate value function in order to select actions by testing them against the pendulum simulator code. The script will control the pendulum for 3000 steps or until it falls over, after which it displays three plots showing the pendulum behavior during the test. A successful test will have plots showing 3000 steps of data, with each of the state dimensions oscillating over a small range.

You can download all of the files mentioned above from the root directory of the class web page in the file `pendulumALP.zip`. Note that this is not linked from the anywhere on the class web page; you’ll need to type it in.

1 Implementing ALP (10 points)

Implement approximate linear programming, as described in the class notes. Specifically, write a function

$$w = \text{ALP}(\text{samples, gamma})$$

which takes in a set of sample transitions (described above) and a discount factor, and returns a set of approximation weights. Your function should solve a linear program using Matlab’s built in function `linprog`. The remaining functions necessary to solving the problem are described above.

Since we are doing constraint sampling, there is positive probability your ALP will fail (your linear program will be unbounded) on some runs of the testing script. With 2000 samples, this should be rare, but it could happen. For debugging purposes, you may wish to reset the random number generator before each test run in order to ensure repeatable results (see the script for how to do this). You can avoid this difficulty entirely by adding regularization, but it is not required for this assignment.

Your answer for this question should be the ALP function submitted in your code directory.

A few hints:

- For an infinite state space, it’s not practical to sum over all of the state values for your objective function, so you should sum over all of the state values in your training set instead.

- If `linprog` is consistently complaining that the optimizer has stalled, try passing in an options parameter to reduce the tolerance of the optimizer to $10^{-6}$. Set this with

  $$\text{options} = \text{optimset}('\text{TolFun}', 1e-6)$$

  then pass this as an argument to `linprog` as described in the documentation.
• Our solution for this problem is 5 lines without any fancy Matlab tricks. Your solution doesn't need to be just 5 lines, but if you’re doing much more than this, it’s a good sign that you’re working too hard or on the wrong track.

2 Eliminating the Model (10 points)

Examine the function `pendulum_action`. Notice the for loop that iterates over all actions, and simulates the effect of taking each action. An alternative to doing this is to store something called a Q-function instead of a value function. Q-functions let you pick actions without using a simulator. The Bellman equation for a Q-function is:

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a')$$

This eliminates the need for a model when selecting actions because one can say $\pi(s) = \arg\max_a Q(s, a)$.

Recall that for the exact linear program solution to an MDP, we replaced the max in the Bellman equation with $|A|$ constraints per state, 1 for each action. Suppose you have a deterministic MDP, i.e, for all s and a, there exists an $s'$ such that $P(s'|s, a) = 1$. Show how to implement the Q-function formulation of the exact MDP solution using $|A|^2$ constraints for each state. Your solution should have $|S| \times |A|$ variables.

Your answer for this question should be a linear program submitted as text in your PDF. It’s OK to express your linear program in a generic manner since you won’t know the number of states in advance. For example, you can express a set of constraints over the entire state space by writing out one or more generic constraints with universally quantified variables for the states and actions, similar to what we provide in the slides. Don’t forget that a linear program should include an objective function too.

3 The Q-version of the linear program for Noisy Domains (10 points)

a) Suppose your MDP is not deterministic. Write out and justify a mathematical expression for the number of constraints needed for the Q-formulation of the linear programming approach using just $|S| \times |A|$ variables. Hint: The number of constraints will be very large.

b) Suppose you are allowed to use $|S|$ additional variables for a total of $|S| \times |A| + |S|$ variables. Show how to use the extra variables to reduce the number of constraints to a more reasonable number. Show this by writing out the form of the linear program as you did for question 2.
For the next questions, we recommend spending a few minutes with our favorite Java SVM Applet before you begin.

4 Kernelized SVMs I (10 points)

Consider the following code, which implements a (non-kernelized) dual SVM.

```matlab
function [w, svm_b] = dualsvm(X, T, C)
% DUALSVM Implementation of the dual. Should be API-compatible with svm.m,
% our solution for the primal.
T = T * 2 - 1;
numpoints = size(X, 1); % X has points rows and dims cols.
H = (T * T') .* (X * X');
f = -ones(numpoints, 1);
A = zeros(1, numpoints);
b = 0;
Aeq = T';
beq = 0;
lb = zeros(numpoints, 1);
ub = C * ones(numpoints, 1);
[a, v] = quadprog(H, f, A, b, Aeq, beq, lb, ub, [], ...
    optimset('Algorithm', 'interior-point-convex'));
% Compute the weights.
w = sum(repmat(a .* T, 1, size(X,2)) .* X, 1)';
% b from text
svm_b = sum((T - X*w).*((a < C) & (a > sqrt(eps))),1) / sum((a < C) & (a > sqrt(eps)));
end
```

**Note:** The above contains a change from what we gave you earlier. In the earlier version, we forgot to filter the Lagrange multipliers so as to use only those that were positive in the calculation of b. For robustness, we also discard those with very small but positive Lagrange multipliers. No points were removed in grading if students made the same mistake(s).

Extend this code to the kernelized case. You should supply the kernel function using a Matlab `function handle`. Function handles make it possible to pass functions as arguments to other functions; in this case, it will allow you to write a single SVM function, and then pass it arbitrary kernels. See `doc function_handle` for details. Your code should have the following function signature:

```matlab
function [w, svm_b] = kernelsvm(X, T, C, handle)
    % code goes here.
end
```

Here `X`, `T`, and `C` are as before, while `handle` is the function handle argument for the kernel function that will be used by the SVM. Note that in the kernelized case, your `w` should just be the Lagrange
multipliers ($\mu$ in the slides) since you can’t compute a single weight vector that is used for all test points.

Be sure to test your code carefully! Submit your code in a file named `kernelsvm.m` in your code directory.

5 Kernelized SVMs II (10 points)

Consider the (non-linearly-separable) two-dimensional data below:

\[
\begin{bmatrix}
-1 & 1 & + \\
1 & 1 & + \\
1 & -1 & + \\
-1 & -1 & + \\
-2 & 2 & - \\
2 & 2 & - \\
2 & -2 & - \\
-2 & -2 & - \\
\end{bmatrix}
\]

Implement a quadratic kernel for your `kernelsvm`, and apply it to these data. Demonstrate that these data can be separated by your quadratic SVM by creating a figure that includes the data themselves, as well as an illustration of the returned decision boundary. Note that your decision boundary will be a more-than-two-dimensional hyperplane; plot its projection onto two dimensions. The easiest way to illustrate the decision boundary is to overlay a dense grid on your figure, and then color each grid element according to which side of the decision boundary it lands on. See our favorite SVM applet for an example of what this might look like.

Repeat the above, but for degree-one (linear), degree-five and degree-nine kernels. Generate equivalent figures in each case. Then, spend a few sentences comparing the decision boundaries for your various kernels. How do they compare? What interesting thing has happened?

Submit your figures and any code you used to produce them in your PDF.