

CS 221, Autumn 2009

Problem Set #3 Programming Assignment

Due by 9:30am on Tuesday, November 10. Please see the course information page on the class website for late homework submission instructions. SCPD students can also fax their solutions to (650) 725-1449. We will not accept solutions by email or courier.

Programming part (30 points)

Overview

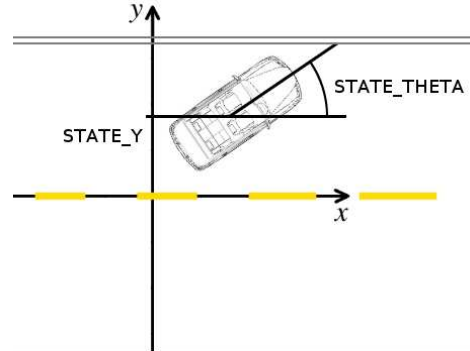
In this programming assignment, you'll use the value iteration algorithm to find a policy for driving a car on a loose-surface road. The car will begin facing down the road in one direction, traveling at a fixed speed. Your policy will need to learn to spin the car around and then drive off in the opposite direction as quickly as possible.

You are provided with a simple simulator of the car that, given a (real-valued) state vector, will simulate forward in time using a specified (real-valued) action vector. Since the simulator operates on continuous-valued states and actions, we've discretized the state and action spaces for you. In these discretized spaces, you'll use the simulator to collect data and build up a probabilistic model of the car's dynamics. Once you have such a model, you'll compute the value function (using value iteration) for the discrete MDP, and finally compute the optimal policy from the value function. This policy can be used by the provided graphical display function, so that you can see how the car performs using your policy.

Please keep all of your code confined to `learnTransitionModel.m` and `solveMDP.m`. The code necessary to complete this assignment is not very long – you shouldn't need to modify any other files.

The Continuous State Model

The road on which the car is driving defines the x -axis of the coordinate system (which you can think of as east). The y -axis points to the left side of the road (north) if the car is facing in the positive x direction. Angles are measured counter-clockwise from the x axis, so if the car's heading is 0, then it is facing directly down the road, and if its heading is π , then it is facing down the road in the opposite direction.



The state of the car, s , can be described by an array of 4 real numbers: $s = [y, v_x, v_y, \theta]$. y is the y position of the car relative to the road. v_x and v_y are the car's velocity along the x and y axes, and θ is the car's heading.

In each state, one may choose an action for the car to execute. An action is an array of 2 real numbers: $a = [\alpha, \omega]$. α is the steering angle (i.e., the angle of the tires relative to the car's center-line – positive angles cause a left turn, negative angles cause a right turn), and ω is the “velocity” of the car's wheels. As an example, if $a = [0, 10]$, then the car will drive straight, with the wheels spinning at a rate such that they propel the car at 10m/s (i.e., they spin at a rate of $\omega = 10/r$ radians/sec, where r is the radius of the wheel). The car has an infinite amount of torque, and will immediately drive the wheels at whatever velocity is commanded. Choosing $\omega \leq 0$ will result in applying the car's brakes – you cannot drive backward.

While the car has infinite torque, it does not have infinite traction. The car is driving on a loose surface so that it does not have the best traction. If the wheel velocity is set too high or if the car attempts to turn too abruptly, the wheels will lose traction and the car will not be able to steer effectively. A learned policy will thus have to balance the desire to accelerate and turn around with the need to avoid skidding out of control.

Some simple car simulator functions and a definition of the continuous state and action spaces are provided for you. A state is just an array of size `STATE_SIZE`, and an action is an array of size `ACTION_SIZE` (where `STATE_SIZE = 4`, `ACTION_SIZE = 2`). The elements of a state vector are indexed by the constants `STATE_Y`, `STATE_VX`, `STATE_VY`, `STATE_THETA`, and the elements of an action vector are indexed by `ACTION_STEERING` and `ACTION_WHEEL_VEL`.

When your policy is evaluated, the car will start out in the “initial state.” You can set a state vector to the initial state using the `setStateToInitial()` function.

The `simulate()` function will compute the next state for a given state and action pair, according to the simulated vehicle dynamics. This is the function you will use to build your model.

The Discrete State Model

Solving the complete continuous MDP is not easy, so we will discretize the state and action spaces. Code is provided for managing a discrete model of the system. You can look at the code for details, (see also the last section of this handout), but roughly speaking, it provides two basic facilities: discretization of continuous states and actions, and a data structure to hold your transition probability model (P_{sa}).

Discrete States and Actions

A discrete state or action is represented by a single integral value. You can convert a continuous state (represented by an array of numbers) into a discrete state (a single integer) using the `discretizeState()` function. Likewise, `discretizeAction()` will convert a continuous action into one of the `DISCRETE_ACTION_COUNT` available discrete actions (there are 7 possible steering angles, and 5 possible wheel speeds, for a total of 35 discrete actions).

For as long as you're dealing with discrete states and actions, you can represent your value functions and policies as arrays. For instance, your value function and optimal policy will be represented as `DISCRETE_STATE_COUNT` size arrays. For example, if V is the value function and π^* the optimal policy, then $V(i)$ would be the value of discrete state i , and $\pi^*(i)$ would be the corresponding (discrete) action to be taken when in that state. (`DISCRETE_STATE_COUNT` is the number of different discrete states, as defined in `setVars.m`.)

The Transition Model

In addition to providing utilities for converting between continuous (simulator) states and discrete (MDP) states, you are also given a special data structure that can be used to build a transition model for the system. The entire transition model can't be stored as an array because it would be too large, so instead we store it in a "sparse" structure that doesn't store all of the 0 probabilities.

When you first create a `TransitionModel`, it is in a degenerate state (all states have probability 0). You can add "observations" of transitions using the `transitionModelAddCount()` function and, once you've gathered up enough observations, functionality is provided to compute a transition probability.

Rewards

You're given some reward functions in `reward.m` and `rewardRubber.m` with which to try your algorithm. Each reward function takes in a discrete state index and returns the reward, $R(s)$, for being in that state. Note: $R(s)$ may be negative in this assignment. **Your code should not assume a non-negative value function!**

The task

1. [6 points] Since we can't run the value iteration algorithm directly on the simulator, your first task is to build a transition model that represents the simulator's behavior. As mentioned in lecture, this can be done empirically by having the car experiment in the simulator with different actions in different states. Your learned transition model will not only account for the dynamics of the simulator, but also the uncertainty introduced by mapping continuous states and actions to a small number of discrete states and actions.

In `learnTransitionModel.m`, implement the following data collection strategy:

- (a) Choose a random (continuous) state for the car, s_0 . You can use the `getRandomState()` function, already written for you.
- (b) For *each* of the possible choices of discrete action a , simulate the car from state s_0 under action a to obtain the next state, s_1 . *Tip: use `undiscretizeAction()` to get the continuous action for each discrete action.*

- (c) Add this observation into your discrete transition model after discretizing your states (use `transitionModelAddCount()`). This will update the probability $P_{s_0a}(s_1)$.
- (d) Repeat the above steps at least 1,000,000 times.

This procedure will roughly cover all of the states in the discrete MDP, and will ensure that each possible action in each state is tried the same number of times. Normally, we would choose the actions randomly, but because we're using a relatively small number of examples to build our model it is often hard to ensure we cover all of the actions – and this can lead to trouble when you run value iteration.

Note: running all of these simulations can take quite a while (~ 4 hours on the pod machines). Be sure to allow enough time for this to run to completion. Once this has been done (correctly) once, you can use this model for the rest of the assignment. To help with debugging this portion of the assignment (without having you create the whole transition model and then find out it is wrong) we have provided the test function `testFirstTen()` in `testFirstTen.m`. This will test a transitionModel that is created with only 10 iterations, to ensure that you are doing it correctly. Once this test passes, you can then run for all 1,000,000 iterations.

2. **[10 points]** Now that you have a state transition model, implement the Value Iteration algorithm in `solveMDP.m`, with a discount factor of $\gamma = 0.8$ and using the reward function given by `reward()` in `reward.m`, to find the value function for the MDP. You should represent your value function as a one-dimensional array of length `DISCRETE_STATE_COUNT`. For this assignment, please implement synchronous updates of your value function, i.e. update the value function for all states at the same time.

The Bellman update includes a summation that must be maximized over the possible choices of action:

$$\max_a \sum_{s'} P_{sa}(s')V(s')$$

The summation is the average reward that one would expect to achieve (the “expected value”) after choosing action a in state s . Computing the summation in the Bellman update will be too slow if you just call the `transitionModelProbability()` function on your model and sum over all possible states. We've provided the `expectedValue()` function in `expectedValue.m` that will take in your model, a discrete state and action value, and a value-function array to compute the sum for you. (Why? Because we like you.) The function skips all of the states with 0 probability, making the summation much faster.

Once your value iteration has converged, what is the value of the initial state? (Use the `setStateToInitial()` function to get the initial state, convert it to a discrete state s_0 , and then print out $V(s_0)$). This is the expected sum of rewards that your policy would achieve if it were run from the starting state in the simulator.

You can either run a fixed number of iterations of the Bellman update (e.g., 60 iterations for $\gamma = 0.8$), or use the termination condition given in Russell & Norvig; just be sure that your value function has converged before you move on!

3. **[5 points]** Compute the optimal policy using your converged value function. Represent your policy as an array of `DISCRETE_STATE_COUNT` integer values as described above. If your array is called `piStar`, and your value function is `V`, then you need to compute, for all i :

$$\text{piStar}(i) = \arg \max_a \sum_{i'} P_{ia}(i')V(i');$$

Remember that the summation can be computed efficiently by using `expectedValue()` in `expectedValue.m`. If more than one action yields the same expected value, break ties in favor of the action with the lowest discrete index.

Once you're done, `solveMDP()` will return the optimal policy you found, and also save the policy and value function to file for later use.

4. [3 points] Run your policy in the simulator. You can get your policy either by saving the return value of the `solveMDP()` function or by typing 'load OptimalPolicy' at the command line in matlab, where 'OptimalPolicy' is the name of the file that the policy was saved to in `solveMDP()`. To run your policy, use the `runPolicy()` function, passing it the policy as input. Your car should successfully complete the task of turning around and driving off in the opposite direction as quickly as possible. (If not, you should debug your code and keep trying!) Save the final plot showing the entire path of the car and turn it in.
5. [3 points] Now change your discount factor to $\gamma = 0.03$ and re-run your algorithm (if you are using a constant number of iterations you can run 10 iterations in this case). Don't forget to change the output file name in `solveMDP.m` if you don't want to overwrite your previous results. What is the value of the initial state now? Run this new policy in the simulator (`runPolicy(policy)`). What happens? Why? Turn in a short answer and a plot showing this behaviour.
6. [3 points] Set $\gamma = 0.8$ again. Change your code to use the reward function `rewardRubber()` instead of `reward()` (see `rewardRubber.m`). Re-run `solveMDP()`. Run the new policy in the simulator. What does the car do now? Does this policy make sense for the new reward function? (See the comment or the code in `rewardRubber.m`). Once again, turn in a short answer and a plot of the car's behaviour.

Deliverables

The programming assignment can be done in groups of up to 3 students. Please turn in just one copy per group. Remember, the only files you should modify to complete this assignment are the ones listed below. Your code should be clear and commented. You should print out and turn in the following:

1. The 2 functions to which you added your code: `learnTransitionModel.m` and `solveMDP.m`
2. Any other code (e.g. helper functions) that you wrote
3. The 3 plots generated by `runPolicy` for each your learned policies. You should have a plots for your $\gamma = 0.8$ policy, your $\gamma = 0.03$ policy, and your policy with $\gamma = 0.8$ and the `rewardRubber()` reward function.
4. Your answers to all of the questions above.

Code

Code Infrastructure

The Matlab files for this assignment are available at cs221.stanford.edu/code/pa3.zip. It will unpack into a folder called `cs221_pa3`. (You can unpack this on unix machines using the

`unzip` command). Be sure to read all of the comments in the code.

The data structures are explained in the code. In particular, be sure to read over the comments in `transitionModelCreate.m` for a description of the data structure used to represent the transition model.

Finally, remember to write as efficient code as possible, and to allow time for your model learning to occur. The value iteration portion of the assignment should take considerably less time to run.

We give here short descriptions of the functions that have been provided for you (each is in its own `.m` file). More detailed descriptions can be found in the files themselves.

- `discretizeAction` – Returns the discrete action corresponding to the input continuous action.
- `discretizeState` – Returns the discrete state corresponding to the input continuous state.
- `undiscretizeAction` – Returns a continuous action corresponding to the input discrete action.
- `undiscretizeState` – Returns a continuous state corresponding to the input discrete state.
- `expectedValue` – Returns the expected value of taking the input discrete action in the input discrete state according to the input value function.
- `getRandomState` – Returns a random continuous state.
- `reward`, `rewardRubber` – Returns the reward corresponding to the input discrete state.
- `runPolicy` – Shows a simple display of the car running the input policy from the initial state.
- `setStateToInitial` – Returns the initial continuous state.
- `setVars` – Initializes all of the global parameters used in the code.
- `simulate` – Returns the continuous state which results from taking the input continuous action in the input continuous state.
- `testFirstTen` – Tests a transition model created from just 10 iterations of `learnTransitionModel()` to ensure that you are doing things correctly. This assumes that the random seed has not been changed in `learnTransitionModel.m`.
- `transitionModelAddCount` – Returns the input transition model, modified to reflect an observed input transition (initial discrete state and action, resulting discrete state).
- `transitionModelCreate` – Returns an initial transition model with all 0 probabilities.
- `transitionModelGetSampleCount` – Returns the number of times that an input discrete action has been taken in an input discrete state.
- `transitionModelProbability` – Returns the probability of reaching an input discrete state from another input discrete state, given that the input discrete action was taken.