using the new experience. Specifically, if we keep around the counts for both the numerator and denominator terms of (4), then as we observe more trials, we can simply keep accumulating those counts. Computing the ratio of these counts then given our estimate of P_{sa}.

Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward $R(s)$ in state s to be the average reward observed in state s.

Having learned a model for the MDP, we can then use either value iteration or policy iteration to solve the MDP using the estimated transition probabilities and rewards. For example, putting together model learning and value iteration, here is one possible algorithm for learning in an MDP with unknown state transition probabilities:

```
1. Initialize $\pi$ randomly.
2. Repeat {
    (a) Execute $\pi$ in the MDP for some number of trials.
    (b) Using the accumulated experience in the MDP, update our estimates for $P_{sa}$ (and $R$, if applicable).
    (c) Apply value iteration with the estimated state transition probabilities and rewards to get a new estimated value function $V$.
    (d) Update $\pi$ to be the greedy policy with respect to $V$.
   }
```

We note that, for this particular algorithm, there is one simple optimization that can make it run much more quickly. Specifically, in the inner loop of the algorithm where we apply value iteration, if instead of initializing value iteration with $V = 0$, we initialize it with the solution found during the previous iteration of our algorithm, then that will provide value iteration with a much better initial starting point and make it converge more quickly.

4 Continuous state MDPs

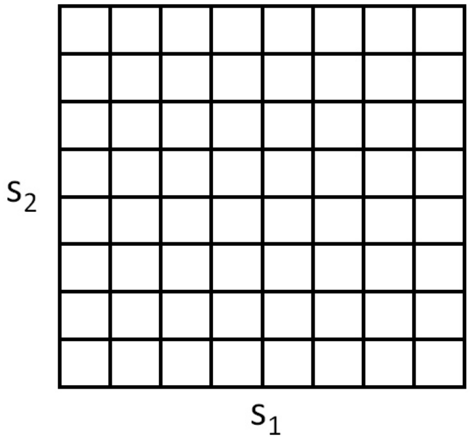
So far, we've focused our attention on MDPs with a finite number of states. We now discuss algorithms for MDPs that may have an infinite number of states. For example, for a car, we might represent the state as $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$, comprising its position (x, y); orientation θ; velocity in the x and y directions $\dot{x}$ and $\dot{y}$; and angular velocity $\dot{\theta}$. Hence, $S = \mathbb{R}^6$ is an infinite set of states, because there is an infinite number of possible positions and orientations for the car.[2] Similarly, the inverted pendulum you saw in PS4 has states $(x, \theta, \dot{x}, \dot{\theta})$, where θ is the angle of the pole. And, a helicopter flying in 3d space has states of the form $(x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi})$, where here the roll ϕ, pitch θ, and yaw ψ angles specify the 3d orientation of the helicopter.

In this section, we will consider settings where the state space is $S = \mathbb{R}^n$, and describe ways for solving such MDPs.

4.1 Discretization

Perhaps the simplest way to solve a continuous-state MDP is to discretize the state space, and then to use an algorithm like value iteration or policy iteration, as described previously.

For example, if we have 2d states (s_1, s_2), we can use a grid to discretize the state space:



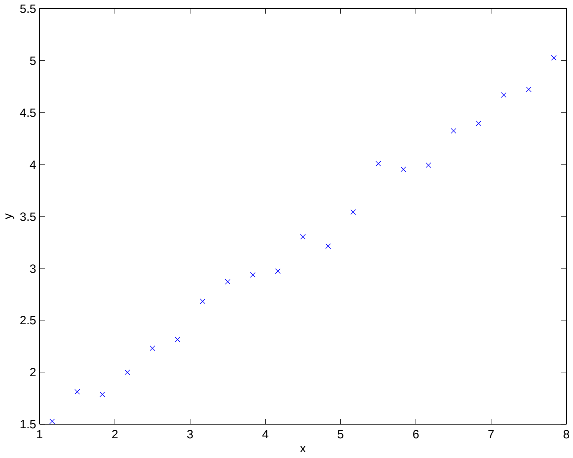
Here, each grid cell represents a separate discrete state $\bar{s}$. We can then approximate the continuous-state MDP via a discrete-state one $(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)$, where $\bar{S}$ is the set of discrete states, $\{P_{\bar{s}a}\}$ are our state transition probabilities over the discrete states, and so on. We can then use value iteration or policy iteration to solve for the $V^*(\bar{s})$ and $\pi^*(\bar{s})$ in the discrete state MDP $(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)$. When our actual system is in some continuous-valued state $s \in S$ and we need to pick an action to execute, we compute the corresponding discretized state $\bar{s}$, and execute action $\pi^*(\bar{s})$.

This discretization approach can work well for many problems. However, there are two downsides. First, it uses a fairly naive representation for V^*

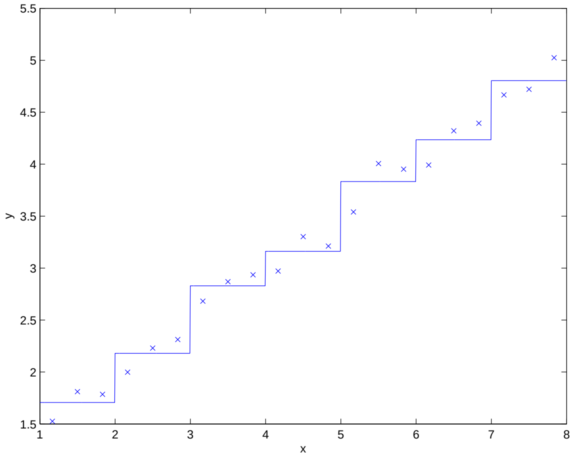
[2] Technically, θ is an orientation and so the range of θ is better written $\theta \in [-\pi, \pi)$ than $\theta \in \mathbb{R}$; but for our purposes, this distinction is not important.

(and π^*). Specifically, it assumes that the value function is takes a constant value over each of the discretization intervals (i.e., that the value function is piecewise constant in each of the gridcells).

To better understand the limitations of such a representation, consider a *supervised learning* problem of fitting a function to this dataset:



Clearly, linear regression would do fine on this problem. However, if we instead discretize the x-axis, and then use a representation that is piecewise constant in each of the discretization intervals, then our fit to the data would look like this:



This piecewise constant representation just isn't a good representation for many smooth functions. It results in little smoothing over the inputs, and no generalization over the different grid cells. Using this sort of representation, we would also need a very fine discretization (very small grid cells) to get a good approximation.

A second downside of this representation is called the **curse of dimensionality**. Suppose $S = \mathbb{R}^n$, and we discretize each of the n dimensions of the state into k values. Then the total number of discrete states we have is k^n. This grows exponentially quickly in the dimension of the state space n, and thus does not scale well to large problems. For example, with a 10d state, if we discretize each state variable into 100 values, we would have $100^{10} = 10^{20}$ discrete states, which is far too many to represent even on a modern desktop computer.

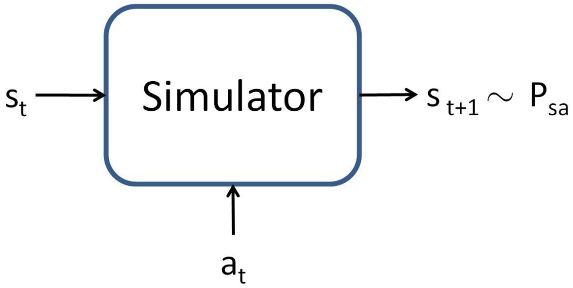
As a rule of thumb, discretization usually works extremely well for 1d and 2d problems (and has the advantage of being simple and quick to implement). Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If you're extremely clever, and somewhat lucky, you may even get it to work for some 6d problems. But it very rarely works for problems any higher dimensional than that.

4.2 Value function approximation

We now describe an alternative method for finding policies in continuous-state MDPs, in which we approximate V^* directly, without resorting to discretization. This approach, caled value function approximation, has been successfully applied to many RL problems.

4.2.1 Using a model or simulator

To develop a value function approximation algorithm, we will assume that we have a **model**, or **simulator**, for the MDP. Informally, a simulator is a black-box that takes as input any (continuous-valued) state s_t and action a_t, and outputs a next-state s_{t+1} sampled according to the state transition probabilities $P_{s_t a_t}$:



There're several ways that one can get such a model. One is to use physics simulation. For example, the simulator for the inverted pendulum in PS4 was obtained by using the laws of physics to calculate what position and orientation the cart/pole will be in at time $t+1$, given the current state at time t and the action a taken, assuming that we know all the parameters of the system such as the length of the pole, the mass of the pole, and so on. Alternatively, one can also use an off-the-shelf physics simulation software package which takes as input a complete physical description of a mechanical system, the current state s_t and action a_t, and computes the state s_{t+1} of the system a small fraction of a second into the future.[3]

An alternative way to get a model is to learn one from data collected in the MDP. For example, suppose we execute m **trials** in which we repeatedly take actions in an MDP, each trial for T timesteps. This can be done picking actions at random, executing some specific policy, or via some other way of choosing actions. We would then observe m state sequences like the following:

$$\begin{array}{l}
s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} \cdots \xrightarrow{a_{T-1}^{(1)}} s_T^{(1)} \\
s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} \cdots \xrightarrow{a_{T-1}^{(2)}} s_T^{(2)} \\
\cdots \\
s_0^{(m)} \xrightarrow{a_0^{(m)}} s_1^{(m)} \xrightarrow{a_1^{(m)}} s_2^{(m)} \xrightarrow{a_2^{(m)}} \cdots \xrightarrow{a_{T-1}^{(m)}} s_T^{(m)}
\end{array}$$

We can then apply a learning algorithm to predict s_{t+1} as a function of s_t and a_t.

For example, one may choose to learn a linear model of the form

$$s_{t+1} = As_t + Ba_t, \tag{5}$$

using an algorithm similar to linear regression. Here, the parameters of the model are the matrices A and B, and we can estimate them using the data collected from our m trials, by picking

$$\arg\min_{A,B} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left(As_t^{(i)} + Ba_t^{(i)} \right) \right\|^2 .$$

(This corresponds to the maximum likelihood estimate of the parameters.)

Having learned A and B, one option is to build a **deterministic** model, in which given an input s_t and a_t, the output s_{t+1} is exactly determined.

[3] Open Dynamics Engine (http://www.ode.com) is one example of a free/open-source physics simulator that can be used to simulate systems like the inverted pendulum, and that has been a reasonably popular choice among RL researchers.

Specifically, we always compute s_{t+1} according to Equation (5). Alternatively, we may also build a **stochastic** model, in which s_{t+1} is a random function of the inputs, by modelling it as

$$s_{t+1} = As_t + Ba_t + \epsilon_t,$$

where here ϵ_t is a noise term, usually modeled as $\epsilon_t \sim \mathcal{N}(0, \Sigma)$. (The covariance matrix Σ can also be estimated from data in a straightforward way.)

Here, we've written the next-state s_{t+1} as a linear function of the current state and action; but of course, non-linear functions are also possible. Specifically, one can learn a model $s_{t+1} = A\phi_s(s_t) + B\phi_a(a_t)$, where ϕ_s and ϕ_a are some non-linear feature mappings of the states and actions. Alternatively, one can also use non-linear learning algorithms, such as locally weighted linear regression, to learn to estimate s_{t+1} as a function of s_t and a_t. These approaches can also be used to build either deterministic or stochastic simulators of an MDP.

4.2.2 Fitted value iteration

We now describe the **fitted value iteration** algorithm for approximating the value function of a continuous state MDP. In the sequel, we will assume that the problem has a continuous state space $S = \mathbb{R}^n$, but that the action space A is small and discrete.[4]

Recall that in value iteration, we would like to perform the update

$$\begin{aligned} V(s) &:= R(s) + \gamma \max_a \int_{s'} P_{sa}(s')V(s')ds' && (6) \\ &= R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')] && (7) \end{aligned}$$

(In Section 2, we had written the value iteration update with a summation $V(s) := R(s) + \gamma \max_a \sum_{s'} P_{sa}(s')V(s')$ rather than an integral over states; the new notation reflects that we are now working in continuous states rather than discrete states.)

The main idea of fitted value iteration is that we are going to approximately carry out this step, over a finite sample of states $s^{(1)}, \ldots, s^{(m)}$. Specifically, we will use a supervised learning algorithm—linear regression in our

[4] In practice, most MDPs have much smaller action spaces than state spaces. E.g., a car has a 6d state space, and a 2d action space (steering and velocity controls); the inverted pendulum has a 4d state space, and a 1d action space; a helicopter has a 12d state space, and a 4d action space. So, discretizing ths set of actions is usually less of a problem than discretizing the state space would have been.

description below—to approximate the value function as a linear or non-linear function of the states:

$$V(s) = \theta^T \phi(s).$$

Here, ϕ is some appropriate feature mapping of the states.

For each state s in our finite sample of m states, fitted value iteration will first compute a quantity $y^{(i)}$, which will be our approximation to $R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')]$ (the right hand side of Equation 7). Then, it will apply a supervised learning algorithm to try to get $V(s)$ close to $R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')]$ (or, in other words, to try to get $V(s)$ close to $y^{(i)}$).

In detail, the algorithm is as follows:

```
1. Randomly sample $m$ states $s^{(1)}, s^{(2)}, \ldots s^{(m)} \in S$.

2. Initialize $\theta := 0$.

3. Repeat {

       For $i = 1, \ldots, m$ {
           For each action $a \in A$ {
               Sample $s'_1, \ldots, s'_k \sim P_{s^{(i)}a}$ (using a model of the MDP).
               Set $q(a) = \frac{1}{k}\sum_{j=1}^{k} R(s^{(i)}) + \gamma V(s'_j)$
                   // Hence, $q(a)$ is an estimate of $R(s^{(i)}) + \gamma \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.
           }
           Set $y^{(i)} = \max_a q(a)$.
               // Hence, $y^{(i)}$ is an estimate of $R(s^{(i)}) + \gamma \max_a \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.
       }
       // In the original value iteration algorithm (over discrete states)
       // we updated the value function according to $V(s^{(i)}) := y^{(i)}$.
       // In this algorithm, we want $V(s^{(i)}) \approx y^{(i)}$, which we'll achieve
       // using supervised learning (linear regression).
       Set $\theta := \arg\min_\theta \frac{1}{2}\sum_{i=1}^{m} \left(\theta^T \phi(s^{(i)}) - y^{(i)}\right)^2$

   }
```

Above, we had written out fitted value iteration using linear regression as the algorithm to try to make $V(s^{(i)})$ close to $y^{(i)}$. That step of the algorithm is completely analogous to a standard supervised learning (regression) problem in which we have a training set $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$, and want to learn a function mapping from x to y; the only difference is that here s plays the role of x. Even though our description above used linear regression, clearly other regression algorithms (such as locally weighted linear regression) can also be used.

Unlike value iteration over a discrete set of states, fitted value iteration cannot be proved to always to converge. However, in practice, it often does converge (or approximately converge), and works well for many problems. Note also that if we are using a deterministic simulator/model of the MDP, then fitted value iteration can be simplified by setting $k = 1$ in the algorithm. This is because the expectation in Equation (7) becomes an expectation over a deterministic distribution, and so a single example is sufficient to exactly compute that expectation. Otherwise, in the algorithm above, we had to draw k samples, and average to try to approximate that expectation (see the definition of $q(a)$, in the algorithm pseudo-code).

Finally, fitted value iteration outputs V, which is an approximation to V^*. This implicitly defines our policy. Specifically, when our system is in some state s, and we need to choose an action, we would like to choose the action

$$\arg\max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')] \tag{8}$$

The process for computing/approximating this is similar to the inner-loop of fitted value iteration, where for each action, we sample $s'_1, \ldots, s'_k \sim P_{sa}$ to approximate the expectation. (And again, if the simulator is deterministic, we can set $k = 1$.)

In practice, there're often other ways to approximate this step as well. For example, one very common case is if the simulator is of the form $s_{t+1} = f(s_t, a_t) + \epsilon_t$, where f is some determinstic function of the states (such as $f(s_t, a_t) = As_t + Ba_t$), and ϵ is zero-mean Gaussian noise. In this case, we can pick the action given by

$$\arg\max_a V(f(s, a)).$$

In other words, here we are just setting $\epsilon_t = 0$ (i.e., ignoring the noise in the simulator), and setting $k = 1$. Equivalent, this can be derived from Equation (8) using the approximation

$$\begin{aligned} \mathrm{E}_{s'}[V(s')] &\approx V(\mathrm{E}_{s'}[s']) & (9) \\ &= V(f(s, a)), & (10) \end{aligned}$$

where here the expection is over the random $s' \sim P_{sa}$. So long as the noise terms ϵ_t are small, this will usually be a reasonable approximation.

However, for problems that don't lend themselves to such approximations, having to sample $k|A|$ states using the model, in order to approximate the expectation above, can be computationally expensive.