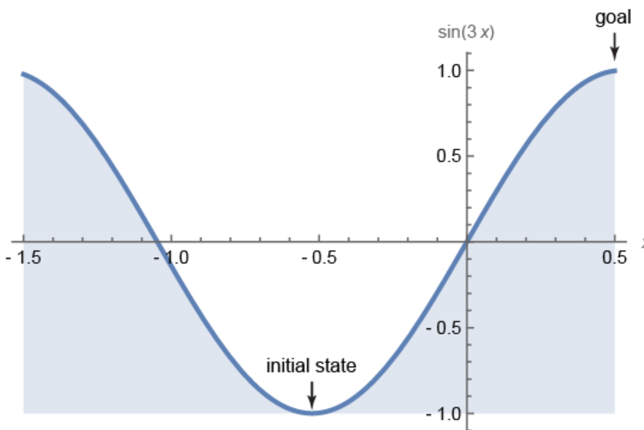
The mountain car problem via Q-Learning

This assignment builds on the mountain car problem, but instead of using the Bellman equations, we will use Q-learning.

To begin with, let's state our problem. We have a car that needs to climb a hill. When the car is sitting at the bottom of the hill, the engine is not strong enough to climb this hill. However, it could climb the hill if it went backwards, and then went forwards, thus using gravity to gain speed when it reached the bottom of the hill. Thus, the interesting idea is that the controller needs to figure out that it must move away from the goal in order to eventually reach the goal. Our objective is to see if the Bellman Equations can find this solution.

Below is a plot of the terrain. The state of the car, i.e., its position and velocity, is defined as $[x, \dot{x}]$, and the height of the hill is $\sin(3x)$. The car is starting near the bottom of the hill at state $[-0.5, 0]$, facing forward (toward increasing value of x).



The controller can produce two actions: $u = +1$, indicating move forward, and $u = -1$, indicating move backward. The car's engine will produce a force $F = 0.2u$. The car will respond by moving forward or backward based on the following state update equations:

$$x^{(n+1)} = x^{(n)} + \Delta t \dot{x}^{(n)}$$

$$\dot{x}^{(n+1)} = \dot{x}^{(n)} + \Delta t\left(-3g\cos\left(3x^{(n)}\right) + \frac{u^{(n)}f}{m} - \frac{k_f}{m}\dot{x}^{(n)}\right)$$

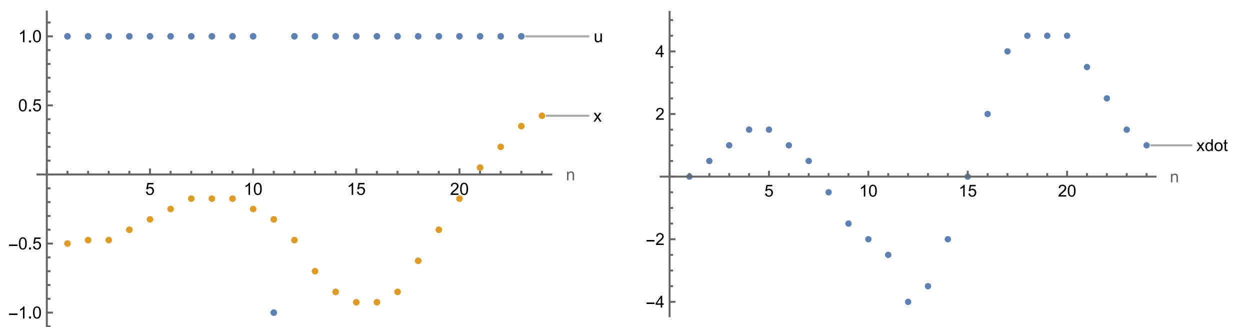
We have the following parameters: $\Delta t = 0.05$, $f = 0.2$, $g = 9.8$, $m = 0.02$, $k_f = 0.5m$. Thus, g is gravity and $\frac{k_f}{m}$ is the coefficient of friction.

In our last HW assignment we found the optimal action by using the Bellman equations to assign a value function as each time point to each state. However, the Bellman equations require us to know how our actions will affect our states. That is, we need to have an accurate model of the car. In Q-learning, we don't have this model, but we still need to find the optimal way to climb the hill.

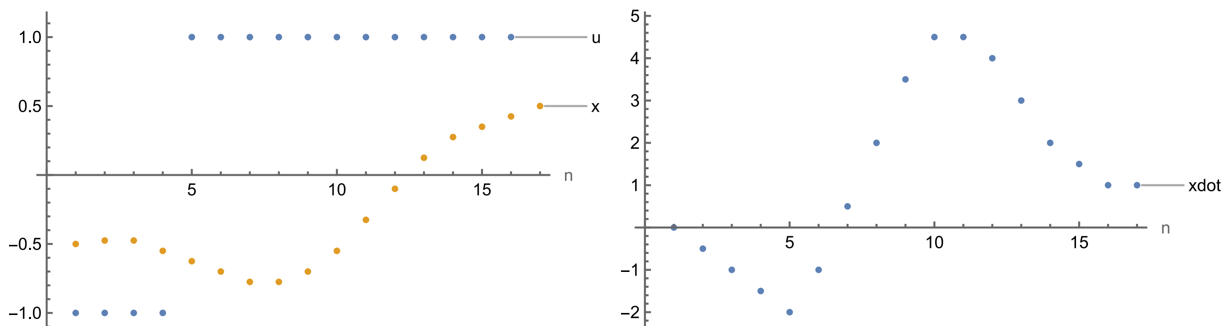
To implement Q-learning in an efficient way, we will need to limit the size of our state space so that it is much smaller than the state space that we implemented for the Bellman equations.

- Implement a state space in which x is bounded by -1.5 and +0.5, with 20 bins, and $\dot{x}$ is bounded by -5 and +5, with 20 bins. Thus, we have a 2D state space of size 20x20. Note that this is much smaller than the 150x100 size state space that we implemented for the Bellman equations.
- Write a function that will take the car from the real space to this grid world, a function that will take the car from the grid world to the real space, and a function that will compute the state transition (given an initial position and action u, compute the next position in the grid world). Note that the grid world is bounded so you want to take care so that if an action takes the car outside the boundary, you will reset its state to be at the edge of the boundary and within it.

To start out, let's try pushing forward for 10 steps (until the car stalls), then backwards for 1 step, and finally forward for 12 steps. You should find something like this:



This sequence of actions gets the car to the goal in around 24 steps. Let's see if Q-learning finds a better solution. Alternatively, we can try pushing backwards for 4 steps, then forwards. This approach gets us to the goal in around 17 steps.



Now let's find the solution that Q-learning will discover.

In this problem (as before), we incur no costs for performing actions, but we get rewarded based on the state of the car. The reward function depends only on the position of the car, not its velocity, and is described as follows:

$$r(x) = x$$

Thus, the closer we get to the top of the hill, the more reward we acquire. Our objective is to maximize the sum of reward until the final time point $T = 20$.

$$J = \sum_{n=1}^{T} r(x^{(n)})$$

- Initial conditions for the value function: make a value function of size 20x20 (size of the state space) for each of our two possible actions: +1, and -1. This function (table) will assign a value at each state for each possible action. Set all values to be zero, that is: $V(\boldsymbol{x}, u) = 0$.
- At $n = 1$, set initial state: $x^{(1)} = -0.5$ and $\dot{x}^{(1)} = 0$.
- Set Q-learning parameters: $\alpha = 0.1$, $\gamma = 0.95$.
- Pick the optimal action at our current state: we will use the value function to define probabilities of each possible action, thus allowing for exploration.

$$\Pr\left(u^{*(n)} = +1 | \boldsymbol{x}^{(n)}\right) = \frac{\exp(V(\boldsymbol{x}^{(n)}, +1))}{\exp(V(\boldsymbol{x}^{(n)}, +1)) + \exp\left(V(\boldsymbol{x}^{(n)}, -1)\right)}$$

$$\Pr\left(u^{*(n)} = -1 \middle| \boldsymbol{x}^{(n)}\right) = 1 - \Pr\left(u^{(n)} = +1 \middle| \boldsymbol{x}^{(n)}\right)$$

- Toss a coin with above probabilities to determine the optimal action at the current state.
- Perform the action $u^{*(n)}$, thus arriving in state $\boldsymbol{x}^{(n+1)}$ and receiving reward $r(\boldsymbol{x}^{(n+1)})$.
- Update the value function:

$$V\left(\boldsymbol{x}^{(n)}, u^{*(n)}\right) \leftarrow (1-\alpha)V\left(\boldsymbol{x}^{(n)}, u^{*(n)}\right) + \alpha\left(r\left(\boldsymbol{x}^{(n+1)}\right) + \gamma \max_{u} V\left(\boldsymbol{x}^{(n+1)}, u^{(n+1)}\right)\right)$$

  In the above equation, the last term is the maximum value that can be attained at state $\boldsymbol{x}^{(n+1)}$ across all actions.
- Continue for $n = 1 \ldots T$. Repeat 10,000 times to gradually improve the value function.
- At the end of training, plot u^*, x, and $\dot{x}$ as a function of n.