Optimal and Learning Control for Robotics
Exercise Series 4

Bo Peng, New York University

Spring 2020

Question 1

Solution a) The value function V is a linear combination of parameter θ_V and basis functions $B(x)$:

$$\begin{aligned} V &= \theta_V^T B(x) \\ &= \sum_{i=1}^{N} \theta_{Vi} B_i(x) \end{aligned} \tag{1}$$

The derivative of θ_V is:

$$\nabla_{\theta_V} V = \begin{bmatrix} \frac{\partial V}{\partial \theta_{V1}} \\ \frac{\partial V}{\partial \theta_{V2}} \\ \vdots \\ \frac{\partial V}{\partial \theta_{VN}} \end{bmatrix} = \begin{bmatrix} B_1(x) \\ B_2(x) \\ \vdots \\ B_N(x) \end{bmatrix} = B(x) \tag{2}$$

b) The derivative of $\ln \pi(u|x, \theta_\pi)$ is:

$$\begin{aligned} \nabla_{\theta_\pi} \ln \pi(u|x, \theta_\pi) &= \nabla_{\theta_\pi} \ln \frac{\exp\{h(x, u, \theta_\pi)\}}{\sum_a \exp\{h(x, a, \theta_\pi)\}} \\ &= \nabla_{\theta_\pi} \ln \exp\{h(x, u, \theta_\pi)\} - \ln \sum_a \exp\{h(x, a, \theta_\pi)\} \\ &= \nabla_{\theta_\pi} h(x, u, \theta_\pi) - \ln \sum_a \exp\{h(x, a, \theta_\pi)\} \\ &= \nabla_{\theta_\pi} h(x, u, \theta_\pi) - \nabla_{\theta_\pi} \ln \sum_a \exp\{h(x, a, \theta_\pi)\} \end{aligned} \tag{3}$$

In this case, the preference function $h(x, u, \theta_\pi)$ is a linear combination of parameter θ_π and basis functions $\Psi(x, u)$:

$$\begin{aligned} h(x, u, \theta_\pi) &= \theta_\pi^T \Psi(x, u) \\ &= \sum_{i=1}^{M} \theta_{\pi i} \Psi_i(x, u) \end{aligned} \tag{4}$$

So the first part of derivative is:

$$\nabla_{\theta_\pi} h(x,u,\theta_\pi) = \nabla_{\theta_\pi} \theta_\pi^T \Psi(x,u) = \Psi(x,u) \tag{5}$$

Let $u = \sum_a \exp\{h(x,a,\theta_\pi)\}$, so the second part of derivative is:

$$\begin{aligned}
\nabla_{\theta_\pi} \ln \sum_a \exp\{h(x,a,\theta_\pi)\} &= \nabla_{\theta_\pi} \ln u \\
&= \frac{\partial \ln u}{\partial u} \cdot \frac{\partial u}{\partial \theta_\pi} \\
&= \frac{1}{u} \sum_a \frac{\partial \exp\{h(x,a,\theta_\pi)\}}{\partial \theta_\pi} \\
&= \frac{1}{u} \sum_a \frac{\partial \exp\{h(x,a,\theta_\pi)\}}{\partial h(x,a,\theta_\pi)} \cdot \frac{\partial h(x,a,\theta_\pi)}{\partial \theta_\pi} \\
&= \frac{1}{u} \sum_a \exp\{h(x,a,\theta_\pi)\} \cdot \Psi(x,a) \\
&= \sum_a \frac{\exp\{h(x,a,\theta_\pi)\}}{u} \cdot \Psi(x,a) \\
&= \sum_a \frac{\exp\{h(x,a,\theta_\pi)\}}{\sum_a \exp\{h(x,a,\theta_\pi)\}} \cdot \Psi(x,a) \\
&= \sum_a \pi(a|x,\theta_\pi) \cdot \Psi(x,a) \\
&= \Psi\pi
\end{aligned} \tag{6}$$

where,

$$\pi = \left[\pi(u_1|x,\theta_\pi), \quad \pi(u_2|x,\theta_\pi), \quad \ldots, \quad \pi(u_M|x,\theta_\pi)\right] \tag{7}$$

$$\Psi = \left[\Psi(x,u_1), \quad \Psi(x,u_2), \quad \ldots, \quad \Psi(x,u_M)\right] \tag{8}$$

suppose we have M possible control inputs.
Finally, add the 2 parts together to get the final derivative:

$$\nabla_{\theta_\pi} \ln \pi(u|x,\theta_\pi) = \Psi(x,u) - \Psi\pi \tag{9}$$

c) The python code for REINFORCE algorithm is shown here:

```python
class Reinforce:
    """
    An implementation of the reinforce algorithm (with or without baseline)
    """

    def __init__(self, model, cost, policy, discount_factor=0.99,
                 episode_length=100, policy_learning_rate = 0.000001):
        """
        the class constructor
        """
        self.model = model
        self.cost = cost
```

```python

        self.policy = policy

        self.discount_factor = discount_factor
        self.episode_length = episode_length

        self.policy_learning_rate = policy_learning_rate

    def iterate(self, num_iter=1):
        """
        the main loop
        """
        learning_progress = []

        # here we allocate some useful vectors
        x_traj = np.zeros([self.episode_length+1, self.model.num_states])
        u_traj = np.zeros([self.episode_length, 1])
        u_index = np.zeros([self.episode_length], dtype=np.int)
        cost_traj = np.zeros([self.episode_length])

        for i in range(num_iter):
            # generate an episode - start from 0
            x_traj[0,:] = np.zeros([2])
            # TO COMPLETE #
            # you can use the step function of self.model (i.e. the pendulum) to get the next state
            G = 0
            ## generate an episode
            for t in range(self.episode_length):
                x = x_traj[t,:]

                ## take one step and get cost
                idx, u = self.policy.sample(x)
                g = self.cost(x, u)

                ## update buffer
                x_traj[t+1,:] = self.model.step(x, u)
                u_traj[t] = u
                u_index[t] = idx
                cost_traj[t] = g

                G += g * self.discount_factor**t

            # now compute the returns Gt and update the policy parameters self.policy.theta through gradient descent
            # TO COMPLETE #
            for k in range(self.episode_length):
                alpha = np.array([self.discount_factor**i for i in range(self.episode_length-k)])
                Gt = alpha @ cost_traj[k:]

                x, idx = x_traj[k], u_index[k]
                dist, basis_fun = self.policy.get_distribution(x)
                dtheta = basis_fun[:, idx] - basis_fun @ dist
                self.policy.theta -= self.policy_learning_rate * self.discount_factor**k * Gt * dtheta

            # here we store the return at t=0 to get the learning progress
            print(f"Iteration: {i}, Discounted Cost: {G}")
            learning_progress.append(G)
```

```
        return learning_progress

    def get_Policy(self):
        """
        This helper function generate a 50x50 grid (theta x omega) with a policy (for display)
        the policy is computed as the expected control from pi
        we also compute a value function (to be used for the baseline part) - for now it is 0
        """
        n_discrete = 50
        pol = np.zeros([n_discrete,n_discrete])
        val = np.zeros([n_discrete,n_discrete])
        x_range = np.linspace(self.model.state_range[0,0], self.model.state_range[0,1], n_discrete)
        v_range = np.linspace(self.model.state_range[1,0], self.model.state_range[1,1], n_discrete)

        for i, x in enumerate(x_range):
            for j,v in enumerate(v_range):
                dist, basis = self.policy.get_distribution(np.array([x,v]))
                pol[i,j] = dist.dot(self.model.controls)
                # this can be used later to get a value estimate
                # val[i,j] = self.value.getValue(np.array([x,v]))[0]
        return pol, val
```

Listing 1: REINFORCE Algorithm

The hyper-parameters for training the inverted pendulum are: discounted factor $\alpha = 0.99$, episode length 100, order $p = 2$, learning rate $\gamma = 5 \times 10^{-8}$, iteration 10^5. The training results are shown below:

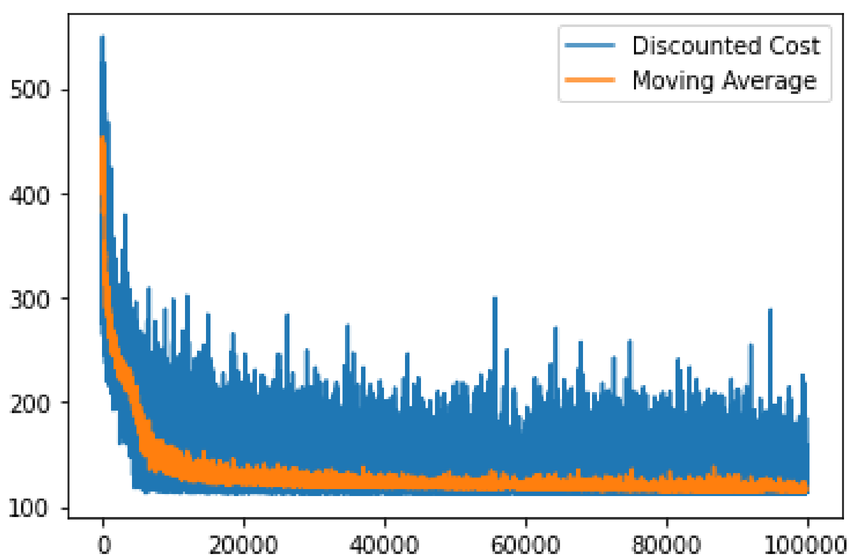

Figure 1: REINFORCE Training History

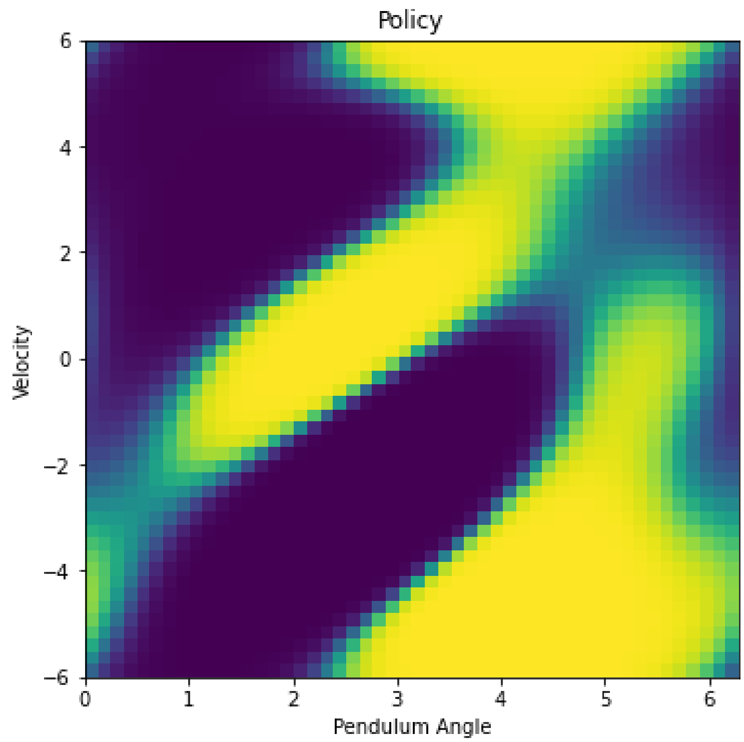

Figure 2: REINFORCE Policy

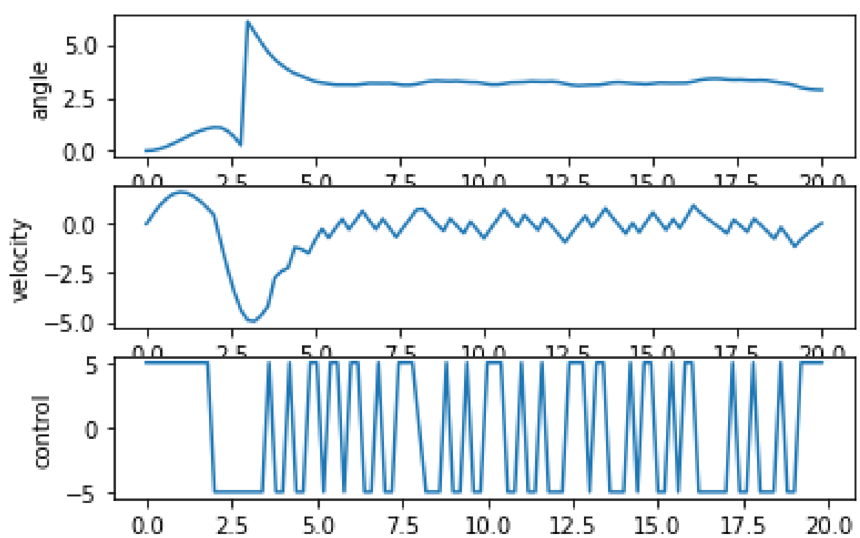

Figure 3: REINFORCE Trajectory

The animation of the trajectory can be found here. As is shown in Fig. 1, the discounted cost converges after 40,000 iterations of training and the learned policy could balance the pendulum appropriately.

c) The python code for REINFORCE algorithm with baseline is shown here:

```
class ReinforceBaseline:
    """
    An implementation of the reinforce algorithm with baseline.
    """

    def __init__(self, model, cost, policy, value, discount_factor=0.99,
                 episode_length=100, value_learning_rate=1e-3, policy_learning_rate = 0.000001):
        """
        the class constructor
        """
        self.model = model
        self.cost = cost

        self.policy = policy
        self.value = value

        self.discount_factor = discount_factor
        self.episode_length = episode_length

        self.policy_learning_rate = policy_learning_rate
        self.value_learning_rate = value_learning_rate

    def iterate(self, num_iter=1):
        """
        the main loop
        """
        learning_progress = []

        # here we allocate some useful vectors
        x_traj = np.zeros([self.episode_length+1, self.model.num_states])
        u_traj = np.zeros([self.episode_length, 1])
        u_index = np.zeros([self.episode_length], dtype=np.int)
        cost_traj = np.zeros([self.episode_length])

        for i in range(num_iter):
            # generate an episode - start from 0
            x_traj[0,:] = np.zeros([2])
            # TO COMPLETE #
            # you can use the step function of self.model (i.e. the pendulum) to get the next state
            G = 0
            ## generate an episode
            for t in range(self.episode_length):
                x = x_traj[t,:]

                ## take one step and get cost
                idx, u = self.policy.sample(x)
                g = self.cost(x, u)

                ## update buffer
                x_traj[t+1] = self.model.step(x, u)
                u_traj[t] = u
                u_index[t] = idx
                cost_traj[t] = g

```

```python
                G += g * self.discount_factor**t

            # now compute the returns Gt and update the policy parameters self.policy.theta through gradient descent
            # TO COMPLETE #
            for k in range(self.episode_length):
                x, idx = x_traj[k], u_index[k]
                dist, basis_fun = self.policy.get_distribution(x)

                alpha = np.array([self.discount_factor**i for i in range(self.episode_length-k)])
                Gt = alpha @ cost_traj[k:]
                V, v_basis = self.value.getValue(x)
                delta = Gt - V

                ## update value function
                dv = v_basis * delta
                self.value.theta += self.value_learning_rate * self.discount_factor**k * dv

                ## update policy
                dtheta = basis_fun[:, idx] - basis_fun @ dist
                self.policy.theta -= self.policy_learning_rate * self.discount_factor**k * delta * dtheta

            # here we store the return at t=0 to get the learning progress
            print(f"Iteration: {i}, Discounted Cost: {G}")
            learning_progress.append(G)

        return learning_progress

    def get_Policy(self):
        """
        This helper function generate a 50x50 grid (theta x omega) with a policy (for display)
        the policy is computed as the expected control from pi
        we also compute a value function (to be used for the baseline part) - for now it is 0
        """
        n_discrete = 50
        pol = np.zeros([n_discrete,n_discrete])
        val = np.zeros([n_discrete,n_discrete])
        x_range = np.linspace(self.model.state_range[0,0], self.model.state_range[0,1], n_discrete)
        v_range = np.linspace(self.model.state_range[1,0], self.model.state_range[1,1], n_discrete)

        for i, x in enumerate(x_range):
            for j,v in enumerate(v_range):
                dist, basis = self.policy.get_distribution(np.array([x,v]))
                pol[i,j] = dist.dot(self.model.controls)
                # this can be used later to get a value estimate
                val[i,j] = self.value.getValue(np.array([x,v]))[0]
        return pol, val
```

Listing 2: REINFORCE Algorithm with Baseline

The hyper-parameters for training the inverted pendulum are: discounted factor $\alpha = 0.99$, episode length 100, order $p = 2$, value learning rate $\gamma_V = 0.01$, policy learning rate $\gamma_\pi = 1\times10^{-6}$, iteration 10^5. The training results are shown below:

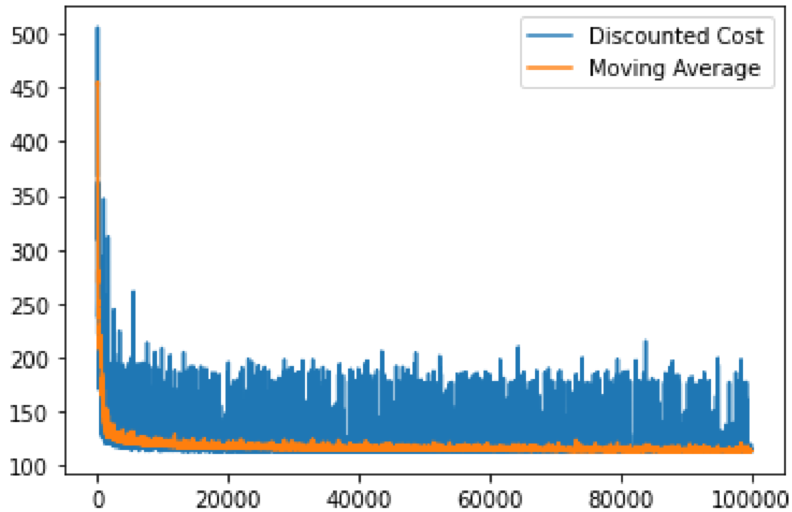

Figure 4: REINFORCE with Baseline Training History

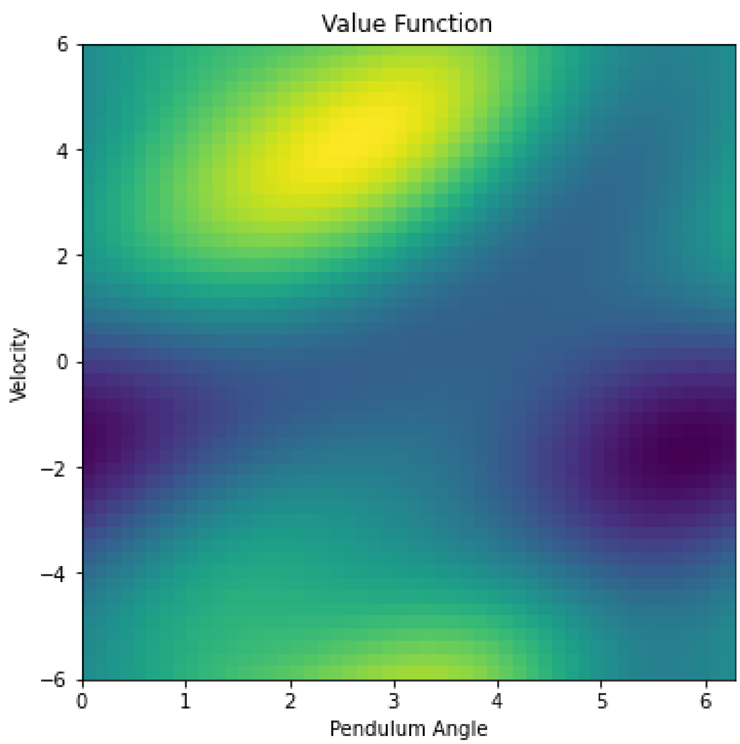

Figure 5: REINFORCE with Baseline Value Function

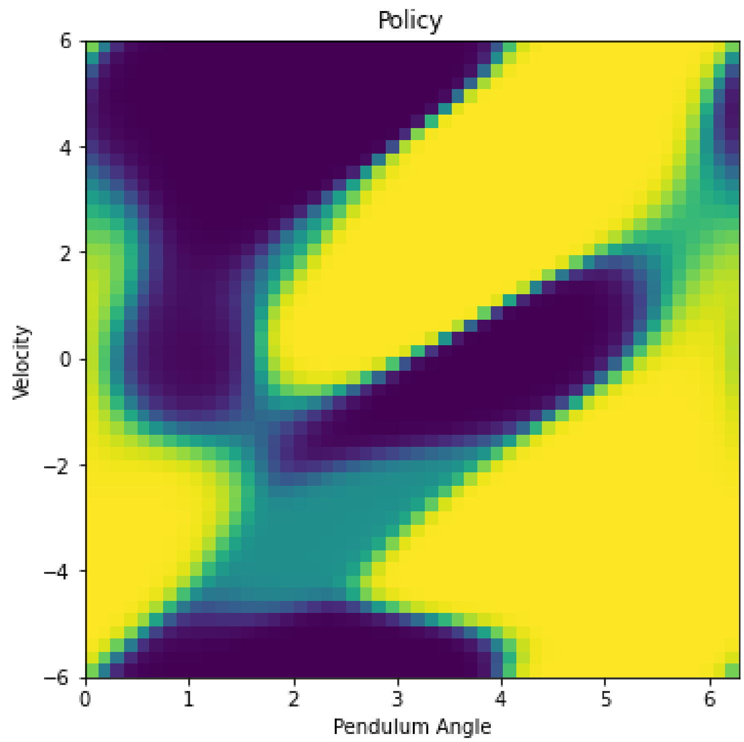

Figure 6: REINFORCE with Baseline Policy

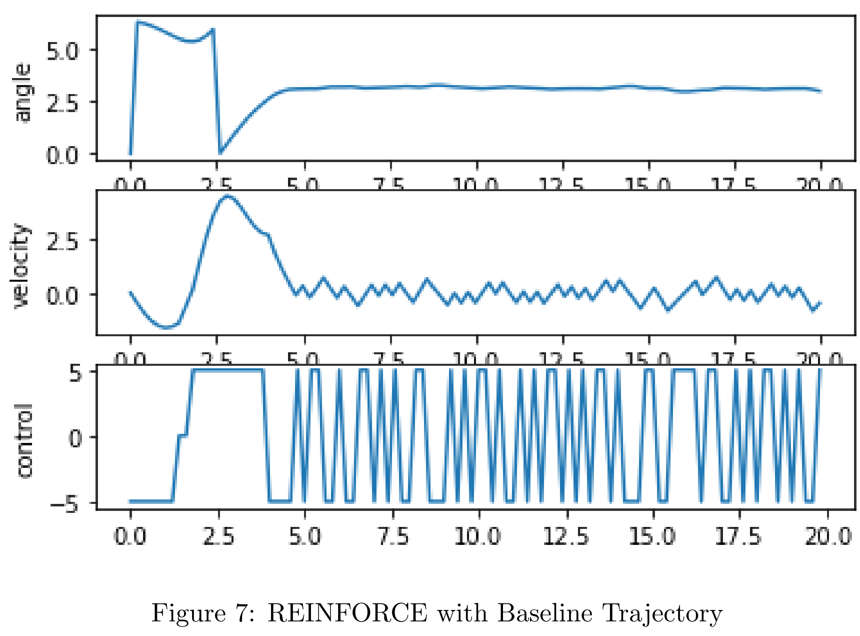

Figure 7: REINFORCE with Baseline Trajectory

The animation of the trajectory can be found here. As is shown in Fig.4, the training converges in about 40,000 steps, which is much faster than vanilla REINFORCE algorithm. And the system is stabilized with the learned policy.

e) Fig.1 and Fig.4 show that REINFORCE with baseline has a faster converge speed and it allows a larger learning rate for policy, which is useful for hyper-parameter tuning. In general, REINFORCE with baseline is easier to use compared to the vanilla algorithm.